# Constraint Modelling Challenge 2005

*In conjunction with The Fifth Workshop on Modelling and Solving Problems with Constraints Held at IJCAI 2005, Edinburgh, Scotland, 31 July, 2005*

**Barbara M. Smith** and **Ian P. Gent**
Cork Constraint Computation Centre, University College Cork, Ireland
and School of Computer Science, University of St Andrews, Scotland

## 1 Introduction

The first Constraint Modelling Challenge was posed in May 2005; we challenged constraint programmers to solve a difficult optimization problem. There are a number of existing papers on the problem that we chose, but it had not previously been tackled using constraint programming, to our knowledge. In this paper, we attempt to present the thirteen submissions that we received, summarising the wide variety of ideas that the Challenge entrants used, and pointing out differences and similarities.

The statement of the problem follows:

> A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customer's order is started (i.e. the first product in the order is being made) a stack is created for that customer. [Each customer places exactly one order.] When all the products that a customer requires have been made, the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the maximum number of stacks that are in use simultaneously, i.e. the number of customer orders that are in simultaneous production, should be minimized.
>
> More formally: we are given a Boolean matrix in which the columns correspond to the products and each row corresponds to the order of a particular customer. The entry $c_{ij} = 1$ iff customer $i$ has ordered some quantity of product $j$ (the quantity ordered is irrelevant). The objective is to find a permutation of the products such that the maximum number of open orders at any point in the sequence is minimized: order $i$ is open at point $k$ in the production sequence if there is a product required in order $i$ that appears at or before position $k$ in the sequence and also a product that appears at or after position $k$ in the sequence.

The problem is one of a number of related sequencing problems discussed by Fink & Voss [4]. Another problem from this paper appears in CSPLib as prob039 (the rehearsal problem) and has previously been tackled using constraint programming. The rehearsal problem can be viewed as the open stacks problem with a different objective (to minimize the order spread, i.e. the total for all customers of the time that their order is in production). The different objective changes the character of the problem completely, however, and experience of solving the rehearsal problem cannot easily be transferred.

Fink and Voss cite several papers on the open stacks problem, using a variety of Operations Research techniques. Linhares and Yanasse [8] list several equivalent problems, including graph path-width.

The Challenge instances were provided by the groups entering the Challenge, and seem to present different kinds of difficulty, depending on their source. Three of the instances (SP2, SP3 and SP4) have not yet been solved optimally.

In the following sections, we first describe a number of preprocessing steps that could be used to simplify the instances, and a number of lower bounds on the number of open stacks that can be derived to assist in proving optimality. We then describe the different solution approaches that were tried in the Challenge entries. We do not discuss the performance of the models here; detailed results can be found in the individual submissions.

## 2 Preprocessing

There are a number of ways in which a given instance can be preprocessed to make it simpler to solve. Many or all of these have previously appeared in the literature on the open stacks problem, but here we mainly cite their use in the Challenge submissions.

The simplest reduction is to remove orders that require no products and products that do not appear in any order, although the Challenge instances should have been constructed so that there are no such orders or products.

Let $P$ be the set of products. Let $C(p)$ be the set of orders (or customers) requiring a product $p$ and $C(P')$ be the set of customers requiring a set of products $P' \subset P$.

If the orders requiring product $j$, $C(j)$, are a subset of those requiring product $k$, $C(k)$, then product $j$ can be removed from the problem; once an optimal sequence is found, product $j$ can be inserted next to product $k$ without affecting the maximum number of open stacks. Many Challenge entries remove dominated products in this way, e.g. [11; 13; 14]. Garcia de la Banda and Stuckey [5] and Miller *et al.* prove the correctness of this reduction; [5; 11] credit it to Becceneri *et al.* [2], who did not give a proof. Shaw and Laborie

[13] note that it is effective on many Challenge instances and, for one set of instances, removes on average half of the products, while in [5] it is calculated that 16% of products are removed across the whole set of instances. Beldiceanu and Carlsson [3] use a special case of this reduction in which two products required for exactly the same orders are merged.

Szymanek and Hennessy [15] recognise that if $C(j) \subseteq C(k)$, then it is safe to insist that product $j$ is sequenced before product $k$; however, they imposed this as a dominance constraint, and found that it did not reduce search.

If the set of products $P$ can be partitioned into two sets $P'$ and $P''$, such that $C(P') \cap C(P'') = \emptyset$, i.e. there are no orders requiring products in both $P'$ and $P''$, then the subproblems defined by $P'$ and $P''$ can be solved separately. Garcia de la Banda & Stuckey [5] note that this simplification was used in [19], and was useful for some of the Challenge instances. Simonis [14] found that the full decomposition was only useful for Challenge instances that were not difficult to solve anyway, but that a special case of *singleton products* was useful: a singleton product is required for only one order and that order requires no other products. Such a product can be scheduled at the beginning of the production sequence without affecting the maximum number of open stacks.

## 3 Lower Bounds

Finding a good lower bound on the maximum number of open stacks is useful in proving optimality: if a solution is found whose value is equal to the lower bound, the solution is known to be optimal and the search can terminate. This applies even to incomplete search methods, which cannot prove optimality in any other way, as well as to complete methods where it may be possible, though time-consuming, to show by exhaustive search that there is no solution with a better value.

The simplest lower bound is the maximum number of orders requiring a product. Other lower bounds are derived by considering the *co-demand graph* that has a node for each of the $m$ orders and an edge between nodes $i$ and $j$ iff there is a product required by both orders. Let $N_i$ be the set of neighbours of node $i$ in this graph.

Miller [9] states and proves a number of theorems on lower bounds based on the co-demand graph. The simplest is that if $\delta$ is the smallest degree of any node, then $\delta + 1$ is a lower bound on the maximum number of open stacks. Garcia de la Banda and Stuckey also use this bound, from [2] where it is not proved. Miller gives similar results that improve slightly on this bound. Another result from [9] calculates $\pi_{ij} = |(N_i \cup N_j) - i - j|$, $1 \leq i, j \leq m, i \neq j$, i.e. $\pi_{ij}$ is the number of orders that share a product with either $i$ or $j$ or both, excluding $i$ and $j$ themselves. If $\rho$ is the minimum value of $\pi_{ij}$ over all pairs of nodes $i$ and $j$, then $\rho + 1$ is a lower bound.

Garcia de la Banda and Stuckey [5] also use a number of bounds from [2] based on deriving *minors* by removing nodes or contracting edges in the co-demand graph.

Pesant [11] discusses the connection between the problem and a constrained graph colouring problem. A solution to the open stacks problem with $k$ stacks corresponds to a $k$-colouring of the co-demand graph, i.e. an allocation of one of $k$ colours to the vertices of the graph in such a way that any pair of nodes linked by an edge have different colours. The reverse is not necessarily true; Pesant shows that additional constraints are required to ensure that a $k$-colouring can correspond to a product sequence with at most $k$ open stacks. Hence, the chromatic number of the graph (the minimum number of colours) gives a lower bound on the minimum number of open stacks [13]. Although solving the graph colouring problem simply to compute a lower bound could be expensive, the size of any clique in the graph also gives a lower bound. Shaw and Laborie [13] use a greedy clique-finding algorithm to find a large clique in the graph; they found this bound useful in proving optimality. Pesant [11] notes that if the graph, after pre-processing to remove dominated products, is a clique, then the minimum number of stacks is the number of orders.

Simonis [14] finds lower bounds by solving small subproblems consisting of 3 to 5 products. The optimal solutions for these subproblems can be easily found by considering all permutations of the products. However, only subsets of the first few products are considered, where the products are ordered by a weight function (also used to derive the initial static variable ordering).

Baptiste [1] finds lower bounds by solving a small MIP model for each position in the sequence. For each position $j$, $1 \leq j \leq n$, the division of products into before and after $j$ that minimizes the number of stacks that are open at $j$ is found. The maximum of these optimal solutions gives a lower bound on the value of the optimal solution to the original problem.

## 4 Symmetry Breaking

As in many sequencing problems, reversing an optimal sequence of products gives another optimal sequence. Several authors add a constraint to break the symmetry. For instance, Shaw & Laborie [13] and Baptiste [1] choose one of the products requiring in the largest number of orders and constrain this product to appear in the first half of the production sequence. Although using very different models, Simonis [14] and Beldiceanu & Carlsson [3] both impose a fixed order on the first two variables selected, corresponding in both cases to two products.

## 5 Constraint Programming Models

### 5.1 A Basic Model

Possibly the most obvious viewpoint on which to base a constraint model has a variable for each product, whose values are the positions in the production sequence: $p_k = i$ iff product $k$ is scheduled in position $i$, $1 \leq i, k \leq n$. There is an allDifferent constraint on these variables, and other variables are introduced to allow the number of open stacks to be determined from the production sequence. Variables $f_j$ and $l_j$, for $1 \leq j \leq m$ can be defined as the first and last positions in the sequence where the stack for order $j$ is open. This can be expressed as the constraints $f_j = \min\{p_k | k \in P_j\}$ and $l_j = \max\{p_k | k \in P_j\}$, where $P_j$ is the set of products in order $j$. A boolean variable $o_{ij}$ indicates whether or not the stack for order $j$ is open during the $i$th time-slot

in the production sequence: $o_{ij} = (f_j \leq i \wedge l_j \geq i)$, $1 \leq i \leq n; 1 \leq j \leq m$.

The number of open orders during the $i$th time-slot is $\sum_{1 \leq j \leq m} o_{ij}$. Finally, the objective is the maximum value of this sum, over all values of $i$, $1 \leq i \leq n$.

### 5.2 A Dual Viewpoint

Since any production sequence can be seen as a permutation of the products, a dual viewpoint has variables representing the positions in the production sequence, whose values are the products, so that $s_i = k$ iff product $k$ is scheduled in position $i$, $1 \leq i, k \leq n$.

These two sets of variables can be linked by channelling constraints: $s_i = k$ iff $p_k = i$. Miller, Prosser & Unsworth [10] and Shaw & Laborie [13] use both sets of variables, linked by these channelling constraints; in [13], the *inverse* constraint in ILOG Solver is used to implement the channelling constraints efficiently.

The values assigned to the $s_i$ variables must be all different; the channelling constraints in fact obviate the need for an explicit allDifferent constraint on either set of variables to ensure correct solutions [7], but specifying such a constraint and maintaining generalized arc consistency can increase domain filtering. Shaw and Laborie [13] maintained GAC on the allDifferent constraint for that reason.

Miller, Prosser and Unsworth [10] use a search strategy that assigns values to the variables $s_1, s_2, ..., s_n$ in that order, i.e. the sequence is built up chronologically. They use a number of lower bounds, analogous to those derived in [9] that they use in preprocessing, but dependent on a partial assignment. They note, however, that the cost of calculating these bounds dynamically is very high. A probably more useful dynamic reduction is based on the observation that if the variables $s_1, ...s_i$ have been assigned, then any product that does not require a new stack can be scheduled next, without affecting whether or not the sequence can be completed optimally. Hence, all the products that are required only by customers whose orders have already been started should be sequenced next, in any order.

Shaw and Laborie [13] also describe this reduction, but finally used a different search strategy that they found more robust. The products are partitioned into two subsets, $P_1$ and $P_2$, such that $P_1$ will be sequenced before $P_2$. Each subset forms a subproblem that can be solved independently of the other, and it is solved by recursively subdividing into two subsets. If at any point either subproblem is insoluble (because the current upper bound on the maximum number of open stacks cannot be achieved) the search can backtrack. The search decisions assign a product to one subproblem or the other.

Hebrard, Hnich and Walsh [6] use a variant of the basic model with only the dual variables. They instead use special-purpose global constraints (one for each order) to link the variables $s_i$, $1 \leq i \leq n$, defining the position of each product in the production sequence, and the variables $o_{ij}$, $1 \leq i \leq n$; $1 \leq j \leq m$, indicating whether or not the stack for order $j$ is open during the $i$th time-slot in the production sequence. They describe the propagation of these global constraints.

They also use a heuristic to choose the variable (product) to occupy the position in the middle of the sequence; their reasoning is that the maximum number of open stacks tends to occur in the middle of the sequence, and hence that the products sequenced in the middle are the most important. For each product, they consider placing that product in the middle of the sequence (in position $m/2$) and finding the minimum number of open stacks at that point. To do this, they partition the other products into two sets, such that the number of orders required by a product in both sets is minimized. Any order that is required both by a product in the first half of the sequence and by a product in the second half of the sequence must have an open stack at position $m/2$. They find the 'Min-Intersect' partition of the remaining products for each product $j$, by solving a subsidiary constraint optimization problem. These solutions are then used as a heuristic to guide the search. The product which will give the minimum number of open stacks is positioned at $m_2$; this minimum number of open stacks at $m/2$ is also a lower bound on the optimal solution that can be found as a result of this (or any subsequent) choice of 'middle' product. The bound is further tightened by constraining the remaining products to be before or after $m/2$ according to the optimal partition, and finding the optimal sequence under those constraints. These bounds are then passed to the complete method.

### 5.3 Permuting the Customers

Rather than finding a permutation of the products, as in the basic model, Wilson and Petrie [17] solve the open stacks problem by finding an optimal permutation of the orders (customers). This idea was previously proposed by Yanasse [18]. If we consider the customer whose order is completed first, and the point in the sequence of products when that happens, every product that customer requires must be made before that point in the sequence. Further, there is no advantage to scheduling any other product before that point. So the maximum number of open stacks occurring while the first customer's stack is open occurs when the last product for the customer is made, and is equal to the number of products the customer requires. Reordering the number of products up to this point in the sequence makes no difference to the maximum number of open stacks in the whole sequence.

This idea can be extended to the remaining customers. A permutation of the customers can be defined from a permutation of the products, by considering the points in the sequence at which the last product for each customer is made, breaking ties for instance by the initial ordering of the customers. Wilson and Petrie also give a function mapping a permutation of the customers to a permutation of the products, and show that the maximum number of open stacks in this sequence is no worse than in any other product sequence corresponding to the same permutation of the customers. Hence, it is sufficient to consider permutations of the customers. Such a permutation defines the *customer elimination sequence*, i.e. it defines the order in which the stacks are closed.

(This idea is related to the dynamic reduction used by Miller et al. [10]: as the sequence of products is built up, any product that does not require opening a stack can be immediately added to the sequence. Hence, the next real decision is

effectively which stack to open next.)

Their CP model has $m$ variables, $r_1, ..., r_m$, constrained to be all different: $r_1$ is the customer whose order is fulfilled first, and so on. Constraints relate the successive neighbourhoods of customers in the co-demand graph to the sets of open stacks when customer $r_1$ is eliminated, when customer $r_2$ is eliminated and so on. (When customer $r_1$ is eliminated, there is a stack open for each of its neighbours.) From the sets of open stacks, the maximum number of open stacks can be found.

Wilson and Petrie eliminate some equivalent sequences of customers by using the idea of dominated customers. At any point in the search, adding a customer $i$ to the sequence as the next customer to be eliminated entails opening stacks for all the remaining customers who require products that are required by customer $i$, i.e. for all customers in $N_i$, the neighbourhood of $i$ in the co-demand graph, who are not in the neighbourhood of any customer already sequenced. Customer $i$ dominates customer $i'$ if the set of new stacks resulting from adding $i$ to the sequence is a strict subset of the set of new stacks resulting from adding $i'$, or the sets of new stacks are the same and $i < i'$. This also gives a variable ordering heuristic: choose the customer which will result in the smallest number of new open stacks.

They also discuss branching decisions based on whether two customers overlap, or if not, in which order they appear, and discuss implied constraints that can be derived.

Permuting the customers rather than the products clearly reduces problem size when the number of customers is less than the number of products; Wilson and Petrie also found that it gives good results generally.

### 5.4 Multiple Viewpoints

Szymanek and Hennessy [15] use a number of different viewpoints, including that of the basic model given earlier, linked by channelling constraints. The principal search variables in their model are boolean variables $cp_{kl}$, representing the relative position of the stacks for customers (orders) $k$ and $l$; $cp_{kl} = 0$ iff the stack for order $k$ is closed before the stack for order $l$ is opened. For any two orders $k, l$ that have a product in common, $cp_{kl} = cp_{lk} = 1$. The importance of these variables is that if $cp_{kl} = 0$, then orders $k$ and $l$ can potentially share a stack location.

Their model also views the problem as one of finding a permutation of the orders. Since the stacks for two or more orders may be started at the same time, if they require the same product, two different permutations of the orders may correspond to the same permutations of the products. Dominance rules are introduced to distinguish some of these cases; for instance, if $P_{j'} \subset P_j$, then order $j$ is considered to appear before $j'$, even if the first product in the sequence required by either of them is required by both. Szymanek and Hennessy give a global constraint that is used to link the $cp_{kl}$ variables, the permutation of the orders, and the objective. The search assigns the $cp_{kl}$ variables first, then the $p_j$ variables of the basic model (and then variables $f_j$ and $l_j$ of the basic model; but since the sequence has been fixed by the previous assignments, there are presumably no decisions to be made at this stage).

Although Szymanek and Hennessy have variables representing the sequence of customers as one viewpoint in their model, they do not construct the product sequence from the customer sequence, as Wilson and Petrie do, and so do not appear to get the same benefits.

### 5.5 A Scheduling Model

Beldiceanu and Carlsson [3] use the basic model, but view each order as a task requiring a resource (its stack) and use the *cumulative* constraint provided in SICStus Prolog to link the sets of variables $s_j, f_j, 1 \leq j \leq m$ to the objective, which on this view is the maximum number of resources in simultaneous use.

They use a static variable ordering, arranging the product variables in descending order of the number of customers requiring each product. The value ordering chooses values as close to the middle of the sequence as possible; hence, the search strategy tries to build up the sequence from the middle to the ends, and puts the products most in demand in the middle of the sequence. Beldiceanu and Carlsson comment that the weakness of the model is the poor propagation of the max amd min constraints, defining the variables $f_j$ and $l_j$. They suggest that this could be improved by developing a global constraint, with appropriate filtering algorithm, to constrain a variable to be the maximum or minimum of a set of variables that are all different.

Shaw and Laborie [13] also use a scheduling viewpoint, as well as the variables of the basic model and their duals. They define variables representing the start and end of each activity (order stack) and relate these to the variables $f_j$, $l_j$. Constraint-based scheduling algorithms are used to reason about the starts and ends of the activities and the resource usage, and these inferences can propagate to the other variables.

The scheduling viewpoint leads to some implied constraints that can be expressed in terms of the start and end variables. Beldiceanu and Carlsson [3] define the *duration* of the stack for order $j$, as the number of products it requires; this then gives a constraint on the start and end times of the activity. An equivalent constraint on $f_j$ and $l_j$ is given by [13; 14]: $l_j \geq f_j + |P_j| - 1, 1 \leq j \leq n$. Simonis [14] extends this to subsets of $P_j$.

Shaw and Laborie give further constraints of this kind. If two orders $j$ and $k$ share at least one product ($P_j \cap P_k \neq \emptyset$), then the corresponding activities must overlap for a duration at least $|P_j \cap P_k|$. Further, if $P_j \subset P_k$, then the activity corresponding to order $j$ can start no later and finish no earlier than the activity corresponding to order $k$. These constraints could be expressed either in terms of the start and finish variables in the scheduling model, or in terms of the $f_j$, $f_k$, $l_j$, $l_k$ in the basic model.

### 5.6 A Graph Colouring Model

As described earlier, the open stacks problem can be viewed as a constrainted graph colouring problem, and this can be useful in deriving lower bounds on the optimal solution. Shaw and Laborie [13] include a graph colouring viewpoint in their model in order to use this perspective on the problems dynamically during the search. The aim is to colour

the co-demand graph so that if two customers both have their stacks open at some point during the product sequence, their nodes in the graph are assigned different colours. Clearly, if two nodes are neighbours in the graph, they must be assigned different colours (there is at least one product that they both require). Shaw and Laborie introduce new variables, $h_j, 1 \leq j \leq m$, where $h_j$ represents the colour assigned to node (customer) $j$. If nodes $j, j'$ are neighbours in the co-demand graph, the constraint $h_j \neq h_{j'}$ is added at the start. Otherwise, the colour variables are related to the other variables of the problems by the constraints: if $f_j \leq l_{j'} \wedge f_{j'} \leq l_j$ then $h_j \neq h_{j'}$. The colours are interchangeable, and to reduce this symmetry, the large clique in the co-demand graph that is found during pre-processing, in order to provide a bound on the optimal value, is coloured at the start.

(Although using a very different approach, Truchet, Bourdon and Codognet [16], as described below, consider the possibility that two customers can possibly share the same stack, in a similar manner.)

Whereas Shaw and Laborie use the graph colouring perspective in conjunction with other viewpoints, Pesant [11] bases his model entirely on solving the constrained graph colouring problem. He finds initial upper and lower bounds on the objective value (the upper bound being one less than the number of customers, since it is easy to detect if the co-demand graph is a clique, and otherwise, the number of stacks required is less than the number of customers). He does a binary search to find the smallest value of $k$ for which a $k$-colouring that can correspond to an product sequence using $k$ stacks exists.

As in [13], the colour symmetries are partly broken by pre-colouring a large clique in the co-demand graph. Further symmetry-breaking constraints are also added, by ordering the sets of nodes assigned to each of the remaining colours.

Having found a $k$-colouring of the graph, it has to be converted, if possible, into a feasible product sequence. For each colour $i$, Pesant first searches for a sequence of the customers in the colour class $C_i$ that have been assigned to that colour; this is the order in which these customers will use their common stack location. This order in turn constrains the product sequence. The variables $p_j, 1 \leq j \leq n$ of the basic model are used, together with an allDifferent constraint on these variables. The search backtracks if it proves impossible to find a product sequence.

Unary constraints can be added on the variables $p_j$ corresponding to the products required by the customer currently being considered that uses stack location $i$. Since none of the customers assigned to colour $i$ can have any products in common, the number of products required by the customers using stack location $i$ previously, and the number required by the customers using this location subsequently, give lower and upper bounds on the position in the product sequence of any product required by this customer.

Pesant comments that the expertise that has been developed in solving graph colouring problems makes this a potentially fruitful approach to solving the open stacks problem.

### 5.7 Putting the products in order

Simonis [14] develops a model in which at any point during the search, the partial solution already built up indicates the order in which the products already considered will be sequenced but not their positions in the sequence. This is implemented by a real-valued variable $y_i$ for each product $i$; a complete solution can be translated into a production sequence by arranging the products in ascending order of the values assigned to the corresponding variables. The domains of the variables are an arbitrary real interval, with two sentinel values, say *start* and *end*, marking the start and end of the interval. Once values in this interval, say $v_1, v_2, ..., v_k$ with $v_1 < v_2 < ... < v_k$ have been assigned to a subset of the variables, the next variable considered is assigned a value in one of the sub-intervals defined by this assignment, i.e. in either $(start, v_1)$ or $(v_1, v_2)$ or ... or $(v_k, end)$ Hence, the new assignment represents a decision as to how the corresponding product should be placed in the sequence in relation to those already placed.

Simonis notes that the advantage of this model, in comparison with a model in which the variables represent the position in the sequence of each product, or the product to be placed in each position in the sequence, is that the branching factor of the search tree is small at the top of the tree. The first two variables are assigned arbitrarily, and the third has three possible choices. Eventually, the final product can go in one of $n + 1$ subintervals. In contrast, if the variables correspond to the positions in the sequence, for the first variable assigned there are $n$ choices, $n - 1$ choices for the second variable and so on.

The products are initially ordered by a weight function, where a product has a high weight if it is required by many customers, each of whom requires few other products. Simonis also uses an expensive shaving technique during search that combines a dynamic variable and value ordering heuristic with domain filtering. At each choice point, for every unassigned product variable and every sub-interval that it could be placed in, the increase in the cost function that would result from placing the product in that sub-interval is computed. This may remove some inconsistent values; thereafter the variable that causes the greatest increase in cost and the value that causes the smallest increase in cost is selected.

Simonis also uses a partial search (using *credit-based search*) to find good solutions quickly, combined with the complete search.

The values assigned to the $y_i$ variables are hard to interpret in this model; we might think of them as something like a time. Variables $f'_j, l'_j$ and so on are defined in a similar way to the basic model; for instance, $f'_j = \min\{y_k | k \in P_j\}$, and hence the objective can be defined using similar constraints.

It seems that this model would be similar to using the basic model with a search strategy that chooses the relative order of the products, rather than their positions in the sequence, by adding inequality constraints between them, so that at a choice point during search, the choice might be between $p_1 < p_2$ and $p_1 > p_2$. However, Simonis considered this search strategy and concluded that it did not allow the constraints to propagate well.

## 6 A Mixed Integer Programming Approach

Baptiste [1] gives a MIP formulation for the problem that is similar to the basic CP model given earlier, but uses 0/1 rather than integer variables. So instead of $p_k = i$ iff product $k$ is scheduled in position $i$, he has $x_{ki} = 1$ iff product $k$ is scheduled in position $i$. The constraints $\sum_{k=0}^{n} x_{ki} = 1$, $1 \leq i \leq n$ and $\sum_{i=0}^{n} x_{ki} = 1$, $1 \leq k \leq n$ ensure that exactly one product is scheduled in any position, and every product is scheduled somewhere; the latter constraint replaces the allDifferent constraint.

For every order $j$ and sequence position $i$, three 0/1 variables are defined. Instead of variables $f_j$ and $l_j$, variables $b_{ij}$, $a_{ij}$ are defined such that $b_{ij} = 1$ iff order $j$ starts production in position $i$ or before, and $a_{ij} = 1$ iff order $j$ ends production at position $i$ or after. The constraints defining these variables are: $b_{i+1,j} \geq b_{ij}$ and $a_{ij} \leq a_{i-1,j}$. With these variables, the variables $o_{ij}$ of the basic model can be defined by $o_{ij} = b_{ij} + a_{ij} - 1$. These variables are linked to the decision variables $x_{ki}$: for any order $k$ and product $j$ such that $c_{jk} = 1$, $x_{ki} \leq o_{ij}$, since if product $k$ is scheduled in position $i$ and order $j$ requires that product, then the stack for order $j$ is open at position $i$. Hence, the objective can be defined as before.

The formulation allows cuts analogous to the implied constraints derived from the scheduling model discussed earlier; for instance, if two orders $j$ and $j'$ are such that $P_{j'} \subseteq P_j$, i.e. all products required by $j'$ are also required by $j$, then $b_{ij} \leq b_{ij'}$ and $a_{ij} \geq a_{ij'}$, for all $i$, $1 \leq i \leq n$.

Baptiste comments that the weakness of the MIP model is that many symmetries (apart from the obvious reversal of the sequence) cannot easily be broken. He tried a secondary objective that found a solution with the minimum number of stacks that gave the lexicographically smallest sequence of products, to break other symmetries that arise from permuting products without changing the number of open stacks, but this was not successful.

## 7 Local Search

Two of the submissions use local search to solve the problems; a few other cases use local search in conjunction with complete search.

Prestwich [12] uses a similar model to Baptiste's MIP model, which can similarly be seen as a 0/1 version of the basic CP model described earlier. He proposes to increase the solution density of the model in order to improve the performance of local search, by adding "pseudo-solutions" in such a way that any pseudo-solution can be transformed into a true solution that is at least as good.

The pseudo-solutions are obtained by relaxing the constraints $\sum_{k=0}^{n} x_{ki} = 1$ and $\sum_{i=0}^{n} x_{ki} = 1$ to $\sum_{i=0}^{n} x_{ki} \geq 1$, for $1 \leq k \leq n$, i.e. each product can appear more than once in the sequence, and the number of product assigned to a position in the sequence can be 0, 1 or more. An alternative relaxation also has the constraints $\sum_{k=0}^{n} x_{ki} \geq 1$ i.e. every position in the sequence must have at least one product. Prestwich shows that a solution satisfying these constraints can be converted into a dominating solution representing a permutation of the products, and that there can be super-exponentially more solutions satisfying either set of relaxed constraints than there are to the original model. A local search algorithm related to WalkSAT is used with all three models, and the relaxed models are shown to be much faster than the original model on an artificial instance for which optimal pseudo-solutions exist. On Challenge instances, the second relaxation is found to give best results.

Truchet, Bourdon and Codognet [16] use an idea reminiscent of one of the viewpoints in [15]: to reduce the maximum number of open stacks, it is important to ensure that as far as possible, two orders that have no products in common can share a stacking space. (Of course, two orders that do have a product in common must both have their stacks open when that product is made and so cannot share a stacking space.) They define a relation $Sep(j, j') = (P_j \cap P_{j'} = \emptyset)$: $Sep(j, j')$ is true for two products $j, j'$ if they can potentially be separated i.e. can share a stacking space. They define a surrogate for the objective, $g$, defined to be the number of pairs of orders, $j, j'$ such that (in terms of the basic model) $f_j > l_{j'} \vee f_{j'} > l_j$, i.e. the stack for order $j$ opens after the stack for order $j'$ closes, or v.v. They show that finding a permutation that maximizes $g$ is equivalent to finding a permutation that minimizes the maximum number of open stacks.

The advantage of defining the new objective function is that $g$ has a greater range of values than the original objective and is more suitable for use as an error or cost function in local search.

Truchet *et al.* use a local search method (Adaptive Search implemented for permutation problems) to find improved configurations by identifying a variable that can be considered (partly) responsible for the poor quality of the current configuration, and trying another value in this variable's domain.

Shaw and Laborie [13] combine the constraint model described earlier with local search, although only for the larger Challenge instances. Whenever a new (and therefore better) solution is found, Large Neighbourhood Search is used to try to improve it further. A sub-sequence of the product sequence is selected at random and LNS attempts to reassign the products in these positions using a smaller number of open stacks.

## 8 A Model Checking Approach

Miller [9] solves the open stacks problem by viewing a product sequence with at most $M$ open stacks as a violation of a safety property; if such a violation is found, model checking provides a counter-example that can be converted into a solution to the open stacks problem with at most $M$ open stacks.

Because proving that no solution exists for a given $M$ is very difficult or in some cases impossible with this approach, Miller derives a number of lower bounds on the value of the optimal solution; if a solution can be found with value equal to the lower bound, then this is sufficient proof that no better solution exists. These bounds are also used by Miller, Prosser and Unsworth [10].

Miller also gives a heuristic method for constructing solutions, which she reports often find the optimal solutions immediately, and otherwise gives a good starting point. This

is based on constructing primarily a sequence of *customers* from which the product sequence can be derived, in such a way as to keep the number of open stacks low. This approach is reminiscent of the complete method used by Wilson and Petrie [17].

Miller comments that an advantage of the model checking system used (SPIN) is the backtracking that it provides. In this case, if at some point during search, the first $i$ products in the sequence have been chosen, and more than $M$ stacks have been opened, the search will immediately backtrack. Further, if this sub-sequence used at most $M$ open stacks, but the search has previously extended a sub-sequence consisting of these $i$ products in a different order, then the search must previously have failed because at some later point it was impossible to use no more than $M$ stacks. Hence, the current sub-sequence should again fail. This is recognised by SPIN, because the state of the system at this point is identical to the previously visited state. The notion of identical states appears in a different guise in the approach described in the next section.

## 9 A Dynamic Programming Approach

Garcia de la Banda and Stuckey [5] use a dynamic programming formulation to solve the open stacks problem. They point out that if a subset $P'$ of the products has been scheduled at the beginning of the sequence, with the last product in this sub-sequnce being $p$, the customers that have open stacks at this point are those who ordered product $p$, as well as those who ordered any other product in $p$ as well as any product that has not been sequenced. This does not depend on the order in which the previous products, $P' - \{p\}$, are scheduled, nor on the order of the remaining products, $P - P'$.

If the minimum number of stacks required to schedule the set of products $S$, given that the products in $P - S$ have already been sequenced, is $stacks_P(S)$ and the set of customers with open stacks at that point is $A(p, P - S - p)$, where $p$ is the last product sequenced in $P - S$, Garcia de la Banda & Stuckey give the basic dynamic programming recursion:

$$stacks_P(S) = \min_{p \in S} \max\{A(p, S - \{p\}), stacks_P(S - \{p\})\}$$

They suggest that this is potentially a much more efficient approach than, say, the basic CP model given earlier, because rather than implicitly considering every permutation of the products, it is only necessary to consider the subsets of $P$.

They use a number of lower bounds on the optimal value, some of which are discussed earlier. They find an optimal solution by either trying every possible value from the lower to the upper bound, or by doing a binary search in this range (as does Pesant [11]).

They comment that although their approach does not use constraint programming, it is equivalent to a constraint programming approach in which states visited are memoized.

## 10 Summary and Conclusions

A number of key ideas emerge from the the approaches adopted in the various submissions to the Challenge. We list here some of the ideas that appear in different submissions, in various guises:

- Instances can often be reduced by preprocessing them, to remove products or customers that cannot affect the solution. Lower bounds on the optimal solution can also be calculated in various ways ahead of search; this appears to be crucial in proving optimality for many instances.
- Although the problem is ostensibly one of finding a permutation of the products, in fact, focussing on the sequence of the customers can be a more fruitful way to consider the problem. This has been done either by specifically sequencing the customers, or by recognising that while sequencing the products, the real decisions are only those that involve opening a new stack.
- If building up the product sequence chronologically, rearranging the products before time $p$ cannot affect the optimal arrangement of the products after $p$, and v.v. This observation has been recognised in various ways, from a dynamic programming approach, to various ways of splitting the sequence and dealing with the two parts separately.
- Two orders can share the same stack location only if they have no products in common; the products that they each require must also be separated in the product sequence. Since it is only by sharing stack locations that the number of open stacks can be reduced, some Challenge submissions have focussed on orders that can potentially be separated in this way as the key to minimizing the number of open stacks.

The Challenge entrants had only limited time (around six weeks) to devise models to solve the open stacks problem. Given that, we were gratified to receive so many excellent entries. The variety of approaches and the number of interesting ideas in the submissions is impressive. The Challenge has conclusively shown that constraint programming is a fruitful approach to solving the open stacks problem. We plan to submit the problem, with the Challenge instances, to CSPLib and hope to see further development of the models described here.

## 11 Thanks, and Advice to Future Organisers

We especially thank Patrick Prosser for proposing the Modelling Challenge as a part of the 2005 Workshop. It was also his idea that it be a challenge, not a competition or evaluation. We think this is very important: we do not feel that constraint programming is at a point where winners can be determined purely by cpu time given the large variety of tools and techniques used.

We thank the organising and programme committees of the IJCAI 05 Workshop on Modelling and Solving Problems with Constraints, and most especially Zeynep Kiziltan for her extensive help. We thank Toby Walsh for agreeing to enter the results of the Challenge into CSPLib. For other help in various ways we would like to thank Ian Miguel, Sylvain Soliman, and members of the CP Pod research group. We would like to thank the UK's Symmetry and Search network, which sponsored the prize for best paper.

Of course we very much thank all the entrants to the Challenge, and especially those who also submitted instances during the first phase.

We hope that modelling challenges can be run in the future, perhaps annually. We do not think it appropriate to keep the challenge in our hands, so we thought it might be useful to offer advice to future organisers.

The first point is to emphasise that we always intended the challenge to allow non-constraint based approaches, and a number of such entries appeared. We think this was very beneficial to the Challenge and for comparing constraints with other approaches, and it should be preserved. The name of the Challenge (i.e. *Constraint* modelling) worried some entrants who checked with us before submitting, but it is hard to see how to rename the Challenge which is about constraints and comparison with other approaches.

The second is that there are certain points in the process that we perhaps needed to put more work into. First among those would have been the selection and distribution of the Challenge instances. Some work double checking them would have been useful, and also solving them with a simple solver to filter out very easy ones. The obvious reason we did not do this is pressure of time.

A third point is that the two phase approach to the Challenge seemed to work well. In the first phase entrants were able to submit their own instances for use in the second phase. In fact we used all instances that were submitted (barring one omitted by error), and they were pleasingly diverse. We didn't concern ourselves if instances were hand crafted to be easy for a particular solver and hard for others, since all other entrants had the chance to construct instances like this too.[1]

We do hope that this report is useful to others interested in this problem, and again hope this will be repeated in future years: but it does take some time to complete, so be warned!

## References


[1] P. Baptiste. Simple MIP Formulations to Minimize the Maximum Number of Open Stacks. IJCAI05 Constraint Modelling Challenge entry.

[2] J. C. Becceneri, H. H. Yanasse, and N. Y. Soma. A method for solving the minimization of the maximum number of open stacks problem within a cutting process. *Comput. Oper. Res.*, 31(14):2315–2332, 2004.

[3] N. Beldiceanu and M. Carlsson. A *cumulative* Model for a Pattern Sequencing Problem. IJCAI05 Constraint Modelling Challenge entry.

[4] A. Fink and S. Voss. Applications of modern heuristic search methods to pattern sequencing problems . *Computers & Operations Research*, 26:17–34, 1999.

[5] M. Garcia de la Banda and P. J. Stuckey. Dynamic Programming to Minimize the Maximum Number of Open Stacks. IJCAI05 Constraint Modelling Challenge entry.

[6] E. Hebrard, B. Hnich, and T. Walsh. Partition with Minimal Intersection. IJCAI05 Constraint Modelling Challenge entry.

[7] B. Hnich, B. M. Smith, and T. Walsh. Dual Models of Permutation and Injection Problems. *JAIR*, 21:357–391, 2004.

[8] A. Linhares and H. H. Yanasse. Connections between cutting-pattern sequencing, vlsi design, and flexible machines. *Comput. Oper. Res.*, 29(12):1759–1772, 2002.

[9] A. Miller. Improved lower bounds for solving the minimal open stacks problem. IJCAI05 Constraint Modelling Challenge entry.

[10] A. Miller, P. Prosser, and C. Unsworth. A Constraint Model and a Reduction Operator for the Minimising Open Stacks Problem. IJCAI05 Constraint Modelling Challenge entry.

[11] G. Pesant. Trying Hard to Solve the Simultaneously Open Stacks Problem with CP. IJCAI05 Constraint Modelling Challenge entry.

[12] S. Prestwich. Open Stack Minimisation by Local Search and Reverse Dominance Reasoning. IJCAI05 Constraint Modelling Challenge entry.

[13] P. Shaw and P. Laborie. A Constraint Programming Approach to the Min-Stack Problem. IJCAI05 Constraint Modelling Challenge entry.

[14] H. Simonis. Modelling Challenge: Benchmark Results. IJCAI05 Constraint Modelling Challenge entry.

[15] R. Szymanek and M. Hennessy. Modelling Challenge – Open Stack Problem. IJCAI05 Constraint Modelling Challenge entry.

[16] C. Truchet, J. Bourdon, and P. Codognet. Tearing customers apart for solving PSP-SOS. IJCAI05 Constraint Modelling Challenge entry.

[17] N. Wilson and K. Petrie. Using Customer Elimination Orderings to Minimise the Maximum Number of Open Stacks. IJCAI05 Constraint Modelling Challenge entry.

[18] H. H. Yanasse. On a pattern sequencing problem to minimize the maximum number of open stacks. *European Journal of Operational Research*, 100:454–463, 1997.

[19] B. J. Yuen and K. V. Richardson. Establishing the optimality of sequencing heuristics for cutting stock problems. *European Journal of Operational Research*, 84:590–598, 1995.


[1] However, note that we required source code to be entered under a promise of confidentiality, meaning that we could if necessary check for entries which had a database of known instances with solutions, which we would have been unhappy with.

# Simple MIP Formulations to Minimize the Maximum Number of Open Stacks

**Philippe Baptiste**
Ecole Polytechnique
Laboratoire d'Informatique LIX, CNRS
F-91128 Palaiseau
Philippe.Baptiste@polytechnique.fr

## Abstract

We consider a manufacturing scheduling problem in which the sequence of products to be manufactured has to be determined so as to minimize the number of customer orders that have been started and are waiting to be completed. We describe a straightforward local search method together with a powerful lower bound based on a MIP formulation. Finally we introduce another MIP with specific cuts that allows us to optimally solve medium size instances. Experimental results are reported.

## 1 Introduction

We consider the manufacturing scheduling problem as described in the 2005 Constraint Modeling Challenge webpages.

"A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customer's order is started (*i.e.*, the first product in the order has been made) a stack is created for that customer. When all the products that a customer requires have been made, the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the number of stacks that are in use simultaneously, *i.e.*, the number of customer orders that are in simultaneous production, should be minimized."

We use the following notation. The Boolean matrix $K$ is used to identify the products required by customers. The entry $K_{cp}$ is 1 if and only if customer $c$ requires product $p$. $m$ and $n$ respectively denote the total number of customers and the total number of products. Customers are numbered from 0 to $m-1$ and products are numbered from 0 to $n-1$. We say that a customer $c$ starts (respectively ends) at $t$ if and only if the first (resp. last) product required by $c$ is sequenced in position $t$.

We refer to the web pages of the challenge `www.dcs.st-and.ac.uk/ ipg/challenge/index.html` for a complete description of the problem and for a brief bibliography.

## 2 Lower Bound

The most basic lower bound is the maximum over all products $p$ of $\sum_{c=0}^{m-1} K_{cp}$. To improve this bound, we introduce $lb(t)$ a MIP based bound associated to $t \in \{0, ..., n-1\}$. Our lower bound is then the maximum over all $t$ of $lb(t)$.

Informally speaking $lb(t)$ corresponds to the problem of deciding which products are sequenced before/after position $t$ in the sequence. Indeed, a product $p$ is either sequenced before or after position $t$. Let then $x_p \in \{0,1\}$ denote the binary variable corresponding to this alternative ($x_p = 1$ if and only if $p$ is sequenced strictly before $t$, $x_p = 0$ otherwise). As exactly $t$ products must be sequenced in position $0, ..., t-1$, we have $\sum_{p=0}^{n} x_p = t$.

Now let us introduce, for each customer $c \in \{0, ..., m-1\}$, three binary variables $b_c, a_c, i_c \in \{0,1\}$.

- $b_c$ equals one if and only if customer $c$ starts before $t$.
- $a_c$ equals one if and only if customer $c$ ends after or at $t$.
- $i_c$ equals one if and only if customer $c$ starts before $t$ and ends after or at $t$ ("$i$" stands for In process).

We can link these 3 variables as follows: $i_c \geq b_c + a_c - 1$ since (1) $i_c$ equals 1 if the customer starts before and ends after $t$ and (2) a customer must start before $t$ or end after $t$ hence $b_c + a_c$ is never 0 in a solution. Our objective is to minimize the number of customers for which $i_c = 1$ so, the objective of the MIP is exactly $\sum_{c=0}^{m-1} i_c$. It now remains to link the customer variables to the product variables: $\forall c \in \{0, ..., m-1\}, (\sum_{p=0}^{n-1} K_{cp}) b_c \geq \sum_{p=0}^{n-1} K_{cp} x_p$ and $(\sum_{p=0}^{n-1} K_{cp}) a_c \geq \sum_{p=0}^{n-1} K_{cp}(1 - x_p)$. In the above equation, $(\sum_{p=0}^{n-1} K_{cp})$ plays the role of a big "$M$" (this is the smallest possible one). Altogether, this leads to:

$$\min \sum_{c=0}^{m-1} i_c$$

$$\begin{cases} \sum_{p=0}^{n-1} x_p = t \\ \forall c \in \{0, ..., m-1\}, i_c \geq b_c + a_c - 1 \\ \forall c \in \{0, ..., m-1\}, (\sum_{p=0}^{n-1} K_{cp}) b_c \geq \sum_{p=0}^{n-1} K_{cp} x_p \\ \forall c \in \{0, ..., m-1\}, (\sum_{p=0}^{n-1} K_{cp}) a_c \geq \sum_{p=0}^{n-1} K_{cp}(1 - x_p) \\ \forall p \in \{0, ..., n-1\}, x_p \in \{0,1\} \\ \forall c \in \{0, ..., m-1\}, a_c, b_c, i_c \in \{0,1\} \end{cases}$$

Computing $lb$ should be easy since the MIP contains $O(n+m)$ variables and $O(n+m)$ constraints. Experimental results show that this is true for most of the instances.

## 3 Upper Bound

We use a very simple local search method to compute an upper bound of the optimal solution. As this procedure is not the major contribution of the paper, we mention it briefly.

Through out the local search, our criterion is a lexicographical combination of the number of open stacks and of the sum of the staring times minus the completion times. The secondary criterion is extremely useful to guide the search towards promising regions. The local search is based on a random insertion "move" (remove a product in the sequence and insert it somewhere else). Such a move is always accepted if the objective function is improved. Based on a probability that decreases over time (like in simulated annealing), the move is also accepted if it does not deteriorate the objective function too much. The total number of iterations is exactly 50000.

## 4 MIP Formulation

We first describe a basic model and we then introduce some cuts to improve the search for the solution.

### 4.1 Basic Model

We use the following variables:

- *Product Assignment.* For each product $p$ and each sequence position $t$ ($0 \leq t < n$), $x_{pt}$ is the binary assignment variable that equals 1 if and only if $p$ is sequenced in position $t$.
- *Customer Variables.* For each customer $c$ and each sequence position $t$, we have three binary variables $s_{ct}, e_{ct}, i_{ct} \in \{0,1\}$ that equal 1 if and only if the customer respectively starts before or at $t$, ends after or at $t$, or is in process at $t$.
- *Stack Variable.* The variable $\sigma \in \{0, m\}$ represents the number of stacks simultaneously open in the solution.

The objective is to minimize $\sigma$. We now describe the constraints that ensure we reach an optimal solution.

- *One product at a time.* For all sequence position $t \in \{0, ..., n-1\}$, we have exactly one product, *i.e.*, $\sum_{p=0}^{n-1} x_{pt} = 1$.
- *Products are sequenced.* All products are sequence somewhere, *i.e.*, $\forall p \in \{0, ..., n-1\}, \sum_t x_{pt} = 1$.
- *Bounding the number of open customers.* At any position $t$, the number of customers in process is not greater than $\sigma$, *i.e.*, $\forall t \in \{0, ..., n-1\}, \sum_{c=0}^{m-1} i_{ct} \leq \sigma$
- *Start and end Variables.* Since $s_{ct}$ equals 1 iff, $c$ starts before or at $t$, we have $\forall c \in \{0, ..., m-1\}, \forall t \in \{0, ..., n-1\}, s_{ct} \geq s_{c\,t-1}$. For the same reason, $\forall c \in \{0, ..., m-1\}, \forall t \in \{0, ..., n-1\}, e_{ct} \leq e_{c\,t-1}$. Finally, $i_{ct} = s_{ct} + e_{ct} - 1$.
- *Linking customers and products.* For any customer $c$ and product $p$ such that $K_{cp} = 1$, we have $x_{pt} \leq i_{ct}$.

Altogether this leads to the following MIP.

$$\min \sigma$$
$$\begin{cases} \forall t \in \{0, ..., n-1\}, \sum_{p=0}^{n-1} x_{pt} = 1 \\ \forall p \in \{0, ..., n-1\}, \sum_{p=0}^{n-1} x_{p} = 1 \\ \forall t \in \{0, ..., n-1\}, \sum_{c=0}^{m-1} i_{ct} \leq \sigma \\ \forall c \in \{0, ..., m-1\}, \forall t \in \{0, ..., n-1\}, \\ \quad s_{ct} \geq s_{c\,t-1} \\ \forall c \in \{0, ..., m-1\}, \forall t \in \{0, ..., n-1\}, \\ \quad e_{ct} \leq e_{c\,t-1} \\ \forall c \in \{0, ..., m-1\}, \forall t \in \{0, ..., n-1\}, \\ \quad i_{ct} = s_{ct} + e_{ct} - 1 \\ \forall c \in \{0, ..., m-1\}, \forall p \in \{0, ..., n-1\} \ \ s.t., \ \ K_{cp} = 1, \\ \quad \forall t \in \{0, ..., n-1\}, x_{pt} \leq i_{ct} \\ \forall p \in \{0, ..., n-1\}, \forall t \in \{0, ..., n-1\}, x_{pt} \in \{0,1\} \\ \forall c \in \{0, ..., m-1\}, \forall t \in \{0, ..., n-1\}, s_{ct}, e_{ct}, i_{ct} \in \{0,1\} \end{cases}$$

### 4.2 Some nice properties of the MIP

Our MIP contains $O(n(m+n))$ binary variables and $O(n(m+n))$ constraints. We believe this is rather low and can lead to small search trees. The nice property of the MIP is that when the sequence of products is known (*i.e.*, when $x_{pt}$ variables are fixed), the remaining problem does not require any branching so, the last line of the MIP can be replaced by

$$\forall c \in \{0, ..., m-1\}, \forall t \in \{0, ..., n-1\}, s_{ct}, e_{ct}, \mathbf{i_{ct}} \in [\mathbf{0}, \mathbf{1}].$$

Interestingly, we could also do the opposite. Indeed, when $s_{ct}, e_{ct}, i_{ct}$ values are fixed, it is easy to see that the remaining problem is a pure assignment problem. So it does not require any branching. Hence, we could alternatively replace $x_{pt} \in \{0,1\}$ by $\mathbf{x_{pt}} \in [\mathbf{0}, \mathbf{1}]$.

In practice, this does not prove to be very efficient and it is much better to state to the MIP that all variables are indeed binary. However, this shows that our formulation is relatively tight.

### 4.3 Cuts

First, we try to "tighten" $e$ and $s$ variables. Consider a customer $c$. It requires $q = \sum_{p=0}^{n-1} K_{cp}$ products hence $e_{c\,q-1} \geq 1$ and $\forall t \geq q, e_{ct} \geq 1 - s_{c\,t-q}$ For the same reason, $s_{c\,n-q} \geq 1$ and $\forall t < n-q, s_{ct} \geq 1 - e_{c\,t+q}$.

Second, we add a redundant constraints on "aggregated" customers. Consider two customers $c$ and $c'$ and let $q$ denote the number of products required by $c$ or by $c'$. There are $q$ products sequenced before the end of $c$ or of $c'$ hence, $e_{c\,q-1} + e_{c'\,q-1} \geq 1$ and for the same reason, $s_{c\,n-q} + s_{c'\,n-q} \geq 1$.

Third, we add a constraint of "included" customers. Consider two customers $c$ and $c'$ such that all products required by $c'$ are also required by $c$. Then, for any $t$, we have $s_{ct} \geq s_{c't}$ and $e_{ct} \geq e_{c't}$.

Finally, we add a simple constraint to break symmetry. To do so, we chose a product required by a maximum number of customers and we constrain it to be sequenced in the first half of the sequence.

### 4.4 Things that do not work!

We have tried to use the lower and upper bounds computed in Sections 2 and 3 to tighten $\sigma$. The outcome of the resulting

MIP is either infeasibility (in this case the initial upper bound is optimal) or an optimal solution. Surprisingly, the behavior of this new MIP is much worse than the initial one.

We have also tried to replace the constraints $\forall c \in \{0,...,m-1\}, \forall p \quad s.t., \quad K_{cp} = 1, \forall t \in \{0,...,n-1\}, x_{pt} \leq i_{ct}$ by $\forall c \in \{0,...,m-1\}, \forall t \in \{0,...,n-1\}, \sum_{p:K_{cp}=1} x_{pt} \leq i_{ct}$. This formulation more compact and should be more efficient. Preliminary tests have shown that it increases the average number of nodes by more than 50!

Finally, we have implemented a rather complex MIP formulation to look for an optimal solution that lexicographically minimize the sequence of products. This MIP is also using a quadratic number of variables and constraints and allows to break many symmetries (much more than in the initial MIP) because of the secondary objective function. Unfortunately, this does not work either.

### 4.5 Experimental Results

All experiments were run on PC Dell Latitude D600 running XP. Cplex 9.0 has been used to solve the MIPs. We have followed the guidelines of the Challenge to report experiments. The values required in the tables are well suited to CP approaches but are less relevant to MIP approaches. Indeed, *it is impossible to distinguish the time spent to find an optimal solution from the time spent for the proof itself*. For this reason, we have slightly modified the tables. In the following, the "search effort" always denote the number of nodes in MIP search trees.

We first report our results on a small set of instances with various sizes Miller19, GP1, GP2, GP3, GP4, GP5, GP6, GP7, GP8, NWRS1, NWRS2, NWRS3, NWRS4, NWRS5, NWRS6, NWRS7, NWRS8, SP1, SP2, SP3 and SP4. As both the lower bound computation (Section 2) and the search for an optimal solution are based on MIP (Section 4) they might require a large amount of CPU time. So, we have decided to ignore instances GP5, GP6, GP7, GP8, SP3 and SP4. We first ran our MIP on the remaining instances with a time limit of 1200.0 seconds. Within this time limit, the optimal solution has been found (and proven) for GP1, GP4, NWRS1, NWRS2, NWRS3, NWRS4 and NWRS5. For the remaining instances, we ran our small MIP that provides a lower bound and our simple local search algorithms. The cpu time and the number of nodes reported are then related to this small MIP only.

We have also ran our algorithms on the instances clustered by size (Table 2). Given the huge number of instances, the search for an optimal solution was stopped after 300 seconds. The lower bound (Section 2) and the Upper bound (Section 3) have been computed for all instances that could not be solved within 300 seconds. We have added two columns to the table:

- "Av. Gap" provides the average relative gap between the upper and the lower bound
- "Av. Runtime" provides the average runtime over all instances (both solved and unsolved).

## 5 Conclusion

The major weakness of this approach is that we are not able to break many symmetries. It seems that adding cuts to remove symmetries is both complex and ineffective for the MIP approach. We believe that this strange behavior should be deeply investigated.

| Instance | Best objective value found | Best lower bound found | Proved optimal ? | Total Runtime | Total search effort |
|---|---|---|---|---|---|
| Miller19 | 13 | 12 | No | 26.64 | 9543 |
| GP1 | 45 | 45 | Yes | 686.81 | 183 |
| GP2 | 41 | 40 | No | 922.77 | 27975 |
| GP3 | 41 | 40 | No | 1169.25 | 27055 |
| GP4 | 30 | 30 | Yes | 252.5 | 0 |
| GP5 | n.a | n.a | n.a | n.a | n.a |
| GP6 | n.a | n.a | n.a | n.a | n.a |
| GP7 | n.a | n.a | n.a | n.a | n.a |
| GP8 | n.a | n.a | n.a | n.a | n.a |
| NWRS1 | 3 | 3 | Yes | 0.14 | 0 |
| NWRS2 | 4 | 4 | Yes | 0.28 | 0 |
| NWRS3 | 7 | 7 | Yes | 3.31 | 24 |
| NWRS4 | 7 | 7 | Yes | 10.36 | 193 |
| NWRS5 | 12 | 12 | Yes | 204.18 | 1106 |
| NWRS6 | 12 | 10 | No | 8.03 | 589 |
| NWRS7 | 10 | 6 | No | 21.97 | 569 |
| NWRS8 | 16 | 13 | No | 39.16 | 3231 |
| SP1 | 9 | 6 | No | 5.87 | 153 |
| SP2 | 20 | 16 | No | 99.28 | 25987 |
| SP3 | n.a | n.a | n.a | n.a | n.a |
| SP4 | n.a | n.a | n.a | n.a | n.a |

Table 1: Individual results (Ph. Baptiste)

| File | % solved optimally within the cutoff limit | Mean best value found | Total runtime per instance | | | Search effort per instance to find optimal solution | | | Total search effort per instance | | | Av. Gap % | Av. Runtime |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | mean | median | max | mean | median | max | mean | median | max | | |
| problem_10_10 | 100.00 | 8.03 | 1.90 | 1.62 | 9.07 | n.a. | n.a. | n.a. | 53.14 | 0.00 | 0.00 | 0.00 | 1.90 |
| problem_10_20 | 99.09 | 8.92 | 8.89 | 3.57 | 253.83 | n.a. | n.a. | n.a. | 1212.42 | 0.00 | 0.00 | 0.19 | 11.57 |
| problem_15_15 | 95.82 | 12.87 | 36.43 | 11.64 | 303.30 | n.a. | n.a. | n.a. | 3473.67 | 0.00 | 0.00 | 0.82 | 47.58 |
| problem_15_30 | 11.36 | 14.02 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 8.71 | 6.23 |
| problem_20_10 | 99.27 | 15.88 | 20.58 | 9.22 | 275.10 | n.a. | n.a. | n.a. | 1646.77 | 0.00 | 0.00 | 0.20 | 22.63 |
| problem_20_20 | 1.82 | 17.97 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 8.45 | 6.46 |
| problem_30_10 | 0.73 | 23.95 | 3.51 | 3.54 | 3.76 | n.a. | n.a. | n.a. | 182.00 | 0.00 | 0.00 | 9.70 | 3.62 |
| problem_30_15 | 0.91 | 25.97 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 7.01 | 8.60 |
| problem_30_30 | 3.64 | 28.34 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 5.02 | 48.24 |
| problem_40_20 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. |
| shaw | 0.00 | 13.62 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 11.20 | 5.39 |
| wbop_10_10 | 100.00 | 6.75 | 5.79 | 3.60 | 28.78 | n.a. | n.a. | n.a. | 1144.15 | 0.00 | 0.00 | 0.00 | 5.79 |
| wbop_10_20 | 60.00 | 8.07 | 113.93 | 87.26 | 298.45 | n.a. | n.a. | n.a. | 7064.63 | 0.00 | 0.00 | 7.99 | 189.64 |
| wbop_10_30 | 0.00 | 8.57 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 13.47 | 4.93 |
| wbop_15_15 | 70.00 | 10.37 | 101.09 | 86.27 | 273.07 | n.a. | n.a. | n.a. | 6708.90 | 0.00 | 0.00 | 4.91 | 161.77 |
| wbop_15_30 | 0.00 | 12.20 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 10.71 | 10.33 |
| wbop_20_10 | 92.50 | 14.28 | 43.63 | 21.18 | 295.90 | n.a. | n.a. | n.a. | 4110.78 | 0.00 | 0.00 | 1.75 | 63.05 |
| wbop_20_20 | 0.00 | 14.87 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 11.05 | 7.18 |
| wbop_30_10 | 0.00 | 22.48 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 11.63 | 3.80 |
| wbop_30_15 | 0.00 | 22.38 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 9.93 | 8.30 |
| wbop_30_30 | 0.00 | 24.00 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 8.69 | 122.86 |
| wbo_10_10 | 100.00 | 5.92 | 3.75 | 3.10 | 17.83 | n.a. | n.a. | n.a. | 392.15 | 0.00 | 0.00 | 0.00 | 3.75 |
| wbo_10_20 | 62.50 | 7.35 | 111.25 | 77.52 | 296.77 | n.a. | n.a. | n.a. | 6771.96 | 0.00 | 0.00 | 7.15 | 183.25 |
| wbo_10_30 | 0.00 | 8.22 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 16.30 | 4.64 |
| wbo_15_15 | 63.33 | 9.37 | 98.14 | 80.83 | 292.84 | n.a. | n.a. | n.a. | 4293.39 | 0.00 | 0.00 | 6.53 | 222.58 |
| wbo_15_30 | 0.00 | 11.63 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 12.74 | 8.76 |
| wbo_20_10 | 98.57 | 12.90 | 39.10 | 22.94 | 245.60 | n.a. | n.a. | n.a. | 2436.54 | 0.00 | 0.00 | 0.54 | 42.86 |
| wbo_20_20 | 0.00 | 13.70 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 14.40 | 5.79 |
| wbo_30_10 | 1.00 | 20.05 | 3.77 | 3.77 | 3.77 | n.a. | n.a. | n.a. | 526.00 | 0.00 | 0.00 | 11.97 | 3.60 |
| wbo_30_15 | 0.00 | 20.96 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 10.44 | 6.56 |
| wbo_30_30 | 0.00 | 22.67 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 10.73 | 49.57 |
| wbp_10_10 | 100.00 | 7.28 | 2.04 | 1.71 | 5.19 | n.a. | n.a. | n.a. | 123.20 | 0.00 | 0.00 | 0.00 | 2.04 |
| wbp_10_20 | 92.86 | 8.71 | 23.42 | 5.67 | 300.91 | n.a. | n.a. | n.a. | 3889.37 | 0.00 | 0.00 | 1.26 | 43.39 |
| wbp_10_30 | 0.00 | 9.31 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 16.19 | 3.80 |
| wbp_15_15 | 86.67 | 11.05 | 56.93 | 32.36 | 201.00 | n.a. | n.a. | n.a. | 6958.87 | 0.00 | 0.00 | 2.30 | 89.76 |
| wbp_15_30 | 0.00 | 13.12 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 11.50 | 6.65 |
| wbp_20_10 | 97.50 | 15.13 | 27.22 | 19.59 | 253.82 | n.a. | n.a. | n.a. | 3399.41 | 0.00 | 0.00 | 0.83 | 34.10 |
| wbp_20_20 | 0.00 | 15.41 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 16.40 | 6.06 |
| wbp_30_10 | 0.00 | 23.18 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 13.79 | 3.79 |
| wbp_30_15 | 0.00 | 22.98 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 13.15 | 7.37 |
| wbp_30_30 | 0.00 | 24.54 | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | n.a. | 11.21 | 76.86 |

Table 2: Aggregate results (Ph. Baptiste)

# A *cumulative* Model for a Pattern Sequencing Problem

Nicolas Beldiceanu[1] and Mats Carlsson[2]

[1] LINA FRE CNRS 2729, École des Mines de Nantes, FR-44307 Nantes Cedex 3, France.
Nicolas.Beldiceanu@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden.
Mats.Carlsson@sics.se

**Abstract.** This note presents a constraint model for the pattern sequencing problem proposed at the first constraint modelling challenge at IJCAI. This model is based on a *cumulative* constraint. We get a model with a linear number of variables and constraints according to the number of products and customers. Results with SICStus Prolog are reported on the benchmark suite provided by the organizers of the challenge.

## 1 Problem Description

Given a 0-1 matrix $M$ in which each column $j$ $(1 \leq j \leq p)$ corresponds to a product required by the customers and each row $i$ $(1 \leq i \leq c)$ corresponds to the order of a particular customer[1], the objective is to find a permutation of the products such that the maximum number of open orders at any point in the sequence is minimized. Order $i$ is *open* at point $k$ in the production sequence if there is a product required in order $i$ that appears at or before position $k$ in the sequence and also a product that appears at or after position $k$ in the sequence.

## 2 Contribution and Model

Given a $p \cdot c$ 0-1 matrix $M$, our contribution is a compact model for the pattern sequencing problem. We came up with a model involving $p$ variables in $[1, p]$, $3 \cdot c$ variables taking their values in $[1, p]$, 2 variables taking their values in $[0, c]$, and one *cumulative*, one *alldifferent*, one arithmetic as well as $c$ *minimum* and *maximum* constraints. We first provide an example of the pattern sequencing problem and recall the definition of the *cumulative* constraint. We then present our model and illustrate it on the example initially introduced.

Consider the matrix depicted by part (A1) of Fig. 1. Part (B1) gives its corresponding *cumulated* matrix obtained by setting to 1 each 0 which is both preceded and followed by a 1. The cost 3 of this solution corresponds to the maximum number of 1 in the *cumulated* matrix. But observe that we can get a lower cost by permutting the fourth and the last columns. The corresponding matrix is depicted by part (A2) of Fig. 1. Finally,

[1] The entry $c_{ij} = 1$ iff customer $i$ has ordered some quantity of product $j$.

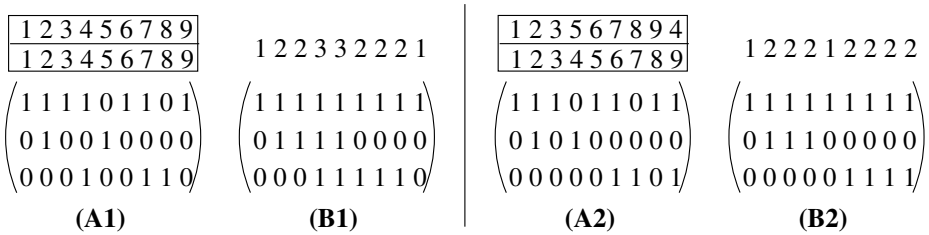



**Fig. 1.** A first matrix (A1) and its corresponding cumulated matrix (B1). A second matrix (A2) where we permute two columns of (A1) and it corresponding cumulated matrix (B2).

part (B2) shows its corresponding *cumulated* matrix from which we conclude that we have a solution of cost 2. Before presenting our model we shortly recall the definition of the *cumulative* constraint.

Given a set of tasks, where each task has an origin, a duration, an end and a resource consumption, the *cumulative* constraint enforces that at each point in time, the cumulated height of the set of tasks that overlap that point, does not exceed a given fixed limit. It also imposes for each task the fact that the end is the sum of the origin and of the duration of that task.

As depicted by Fig. 2, the key idea of our model is to associate to each row (i.e. customer) $i$ of the *cumulated* matrix a stack task which start at the first 1 on row $i$ and ends just after the last 1 of row $i$. Then the cost of a solution is simply the maximum height on the corresponding cumulated profile.

For each column $j$ of the 0-1 matrix initially given we create a variable $V_j$ ranging from 1 to the number of columns $p$. The value of $V_j$ gives the position of column $j$ in a solution. Since $V_1, V_2, \ldots, V_p$ must be assigned to distinct positions we first have an *alldifferent*$([V_1, V_2, \ldots, V_p])$ constraint.

For each row (i.e. customer) $i$ of the 0-1 matrix initially given we create 3 variables $O_i$, $D_i$ and $E_i$ which respectively correspond to the *stack opening time*, the *stack open duration* and the *stack closing time* of customer $i$. Thus $O_i + D_i = E_i$.

We create a *minimum*$(O_i, [V_{i,1}, V_{i,2}, \ldots, V_{i,k_i}])$ and a *maximum*$(E_i - 1, [V_{i,1}, V_{i,2}, \ldots, V_{i,k_i}])$ constraints for linking the opening time $O_i$ and the closing time $E_i$ with those permutation variables which correspond to those columns of the 0-1 matrix initially given having a 1 on row $i$. The minimum of the stack open duration $D_i$ is set to the number of 1 on row $i$ of the initial 0-1 matrix.

Finally we put all the stack tasks in a *cumulative* constraint, telling that each stack task uses one unit of the ressource during all it execution. Since we want to have the same model for different limits on the number of open stacks we create one extra dummy task which starts at 1, ends at $p + 1$ and with a height $H$ in $[0, c]$. We link $H$ with the number of open stacks $Cost$ by the constraint $H + Cost = c$.

## 3 Additional Constraints and Enumeration

**Symmetry breaking** $V_a < V_b$, where $V_a$ and $V_b$ are the first two variables in the static order.

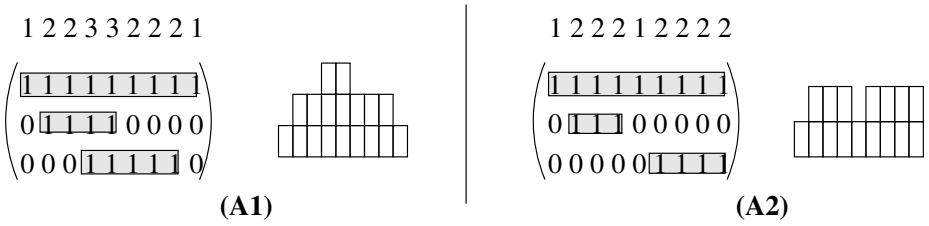



**Fig. 2.** Stack tasks associated to each row (i.e. customer) and corresponding cumulated profile for the two matrices of the previous example.

**Subset row** Suppose the 1s in row $i$ form a subset of the 1s in row $j$. Then we have $D_i \leq D_j, O_i \geq O_j, E_i \leq E_j$.

**Dominance constraints** First, we give two ways of transforming a valid solution $S'$ to another valid solution $S''$ with the same or lower cost. We say that $S''$ dominates $S'$. Then, we express constraints that suppress solutions $S'$, taking care not to suppress any solutions $S''$. The transformations apply to columns $i, j$ and row $q$ such that the set of 1s in column $i$ equals the set of 1s in column $j$, except $M[q, i] = 0$ and $M[q, j] = 1$.

*Transformation 1*: Suppose $S'$ is a solution with $O_q < V_i < V_j$. We obtain $S''$ by swapping columns $i$ and $j$. There are two cases: a) If $j$ is the last column with a 1 in row $q$, then $E_q$ and $D_q$ will decrease. b) Otherwise, $E_q$ and $D_q$ remain unchanged. The other rows are unaffected. The maximum peak cannot increase; it might decrease.

*Transformation 2*: Suppose $S'$ is a solution with $E_q > V_i > V_j$. We obtain $S''$ by swapping columns $i$ and $j$. There are two cases: a) If $j$ is the first column with a 1 in row $q$, then $O_q$ will increase and $D_q$ will decrease. b) Otherwise, $O_q$ and $D_q$ remain unchanged. The other rows are unaffected. The maximum peak cannot increase; it might decrease.

We don't want to apply Transformation 2 in case (b), as the obtained $S''$ would match the precondition of Transformation 1, leading to an infinite sequence of transformations. Hence we add the conjunct $V_j = O_q$ to the precondition of Transformation 2. We obtain the following dominance constraints, which suppress solutions $S'$: $\neg(O_q < V_i \wedge V_i < V_j)$ and $\neg(E_q > V_i \wedge V_i > V_j \wedge O_q = V_j)$.

**Enumeration** We use the following strategy for the enumeration:

- Identical columns are merged.
- Variable choice: Columns are ordered by decreasing total number of 1s.
- Value choice: From the middle to both extremities. For example, for 15 columns we get the following ordering of the values: 8, 9, 7, 10, 6, 11, 5, 12, 4, 13, 3, 14, 2, 15, 1. The intuition being that columns with higher number of 1s should be placed nearer the middle of the schedule. This value choice heuristics was more successful than simpler variants for the most difficult instances.

**Weakness of the Model** It seems that the propagation related to the *minimum* and *maximum* constraints is rather weak since it does not directly takes into account the fact that the permutation variables should take distinct values. One idea for the future could be to create the constraint (and a filtering algorithm achieving arc-consistency)

*min_max*(MIN, MAX, $[V_1, V_2, \ldots, V_n]$) where MIN is the minimum value of $V_1, V_2, \ldots, V_n$, MAX is the maximum value of $V_1, V_2, \ldots, V_n$, and $V_1, V_2, \ldots, V_n$ are all different.

## 4 Results

We run our solver with a 15 seconds time cutoff limit on a 3GHz Pentium 4 with SICStus Prolog on all the instances provided by the organizers of the challenge. The next table summarizes our results. Columns A, B, C, D, E, F, G, H, I, J of the table presented next page respectively provide

- A: The percentage of solved instances to optimality within the cutoff limit of 15 seconds,
- B: The mean best value found for all instances,
- C: The mean total time for finding the optimal solution and for proving optimality for those instances for which we could prove optimality,
- D: The median total time for all instances,
- E: The maximum time for all instances,
- F: The mean total number of backtracks for finding the optimal solution and for proving optimality for those instances for which we could prove optimality,
- G: The median total number of backtracks for all instances,
- H: The maximum number of backtracks for all instances,
- I: The total number of instances,
- J: The number of instances solved to optimality.

All reported times are expressed in milliseconds. A -1 in columns C and F indicates that we could not solve to optimality any instance of a group corresponding to the first column.

| File | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| Harvey_wbo_10_10 | 97.50 | 5.92 | 765.38 | 200 | 15000 | 5879.79 | 1466.0 | 46452 | 40 | 39 |
| Harvey_wbo_10_20 | 2.50 | 8.22 | 4050.00 | 15000 | 15000 | 20840.00 | 116235.5 | 20840 | 40 | 1 |
| Harvey_wbo_10_30 | 0.00 | 8.87 | -1.00 | 15000 | 15000 | -1.00 | 112653.5 | 0 | 40 | 0 |
| Harvey_wbo_15_15 | 11.67 | 10.12 | 1340.00 | 15000 | 15000 | 7869.57 | 86656.5 | 23970 | 60 | 7 |
| Harvey_wbo_15_30 | 0.00 | 13.37 | -1.00 | 15000 | 15000 | -1.00 | 86195.5 | 0 | 60 | 0 |
| Harvey_wbo_20_10 | 71.43 | 12.96 | 5396.40 | 7150 | 15000 | 28902.16 | 36668.0 | 74038 | 70 | 50 |
| Harvey_wbo_20_20 | 1.12 | 15.68 | 4370.00 | 15000 | 15000 | 19625.00 | 65653.5 | 19625 | 90 | 1 |
| Harvey_wbo_30_10 | 10.00 | 20.77 | 9797.00 | 15000 | 15000 | 38710.00 | 46435.5 | 50062 | 100 | 10 |
| Harvey_wbo_30_15 | 0.00 | 23.16 | -1.00 | 15000 | 15000 | -1.00 | 45319.5 | 0 | 120 | 0 |
| Harvey_wbo_30_30 | 0.00 | 25.96 | -1.00 | 15000 | 15000 | -1.00 | 46454.0 | 0 | 140 | 0 |
| Harvey_wbop_10_10 | 100.00 | 6.75 | 3167.25 | 1435 | 13740 | 24003.47 | 13824.0 | 100746 | 40 | 40 |
| Harvey_wbop_10_20 | 0.00 | 8.80 | -1.00 | 15000 | 15000 | -1.00 | 107799.5 | 0 | 40 | 0 |
| Harvey_wbop_10_30 | 0.00 | 9.42 | -1.00 | 15000 | 15000 | -1.00 | 111199.5 | 0 | 40 | 0 |
| Harvey_wbop_15_15 | 3.33 | 11.42 | 775.00 | 15000 | 15000 | 4134.00 | 85696.5 | 5389 | 60 | 2 |
| Harvey_wbop_15_30 | 0.00 | 13.93 | -1.00 | 15000 | 15000 | -1.00 | 88745.0 | 0 | 60 | 0 |
| Harvey_wbop_20_10 | 45.00 | 14.57 | 6716.11 | 15000 | 15000 | 36210.50 | 60876.0 | 73901 | 40 | 18 |
| Harvey_wbop_20_20 | 0.00 | 17.21 | -1.00 | 15000 | 15000 | -1.00 | 62415.0 | 0 | 90 | 0 |
| Harvey_wbop_30_10 | 10.00 | 23.32 | 11875.00 | 15000 | 15000 | 54780.25 | 41399.0 | 69201 | 40 | 4 |
| Harvey_wbop_30_15 | 0.00 | 24.55 | -1.00 | 15000 | 15000 | -1.00 | 41844.5 | 0 | 60 | 0 |
| Harvey_wbop_30_30 | 0.00 | 27.53 | -1.00 | 15000 | 15000 | -1.00 | 49053.5 | 0 | 140 | 0 |
| Harvey_wbp_10_10 | 100.00 | 7.27 | 211.50 | 85 | 1480 | 1239.10 | 463.5 | 9987 | 40 | 40 |
| Harvey_wbp_10_20 | 42.86 | 8.91 | 3376.33 | 15000 | 15000 | 12292.03 | 45963.5 | 59432 | 70 | 30 |
| Harvey_wbp_10_30 | 21.00 | 9.49 | 3299.52 | 15000 | 15000 | 11614.19 | 36407.0 | 75663 | 100 | 21 |
| Harvey_wbp_15_15 | 23.33 | 11.48 | 2631.43 | 15000 | 15000 | 12081.57 | 60864.0 | 27010 | 60 | 14 |
| Harvey_wbp_15_30 | 5.00 | 13.94 | 4828.33 | 15000 | 15000 | 11137.83 | 41508.5 | 34109 | 120 | 6 |
| Harvey_wbp_20_10 | 90.00 | 15.12 | 3403.33 | 3470 | 15000 | 16412.83 | 16051.5 | 48931 | 40 | 36 |
| Harvey_wbp_20_20 | 2.22 | 17.05 | 13390.00 | 15000 | 15000 | 138564.00 | 62686.0 | 140004 | 90 | 2 |
| Harvey_wbp_30_10 | 40.00 | 23.57 | 6535.62 | 15000 | 15000 | 27417.44 | 43974.0 | 56478 | 40 | 16 |
| Harvey_wbp_30_15 | 0.00 | 24.73 | -1.00 | 15000 | 15000 | -1.00 | 55268.0 | 0 | 60 | 0 |
| Harvey_wbp_30_30 | 0.00 | 27.07 | -1.00 | 15000 | 15000 | -1.00 | 47983.5 | 0 | 140 | 0 |
| Simonis_problem_10_10 | 100.00 | 8.03 | 95.53 | 20 | 2590 | 408.95 | 16.0 | 13481 | 550 | 550 |
| Simonis_problem_10_20 | 67.27 | 9.03 | 1722.89 | 1500 | 15000 | 5532.77 | 3441.0 | 79736 | 550 | 370 |
| Simonis_problem_15_15 | 58.00 | 13.06 | 2153.20 | 6475 | 15000 | 9169.26 | 25392.5 | 89755 | 550 | 319 |
| Simonis_problem_15_30 | 37.28 | 14.38 | 1229.27 | 15000 | 15000 | 2411.34 | 27862.0 | 33803 | 220 | 82 |
| Simonis_problem_20_10 | 99.09 | 15.88 | 1660.86 | 625 | 15000 | 7599.73 | 3089.5 | 76284 | 550 | 545 |
| Simonis_problem_20_20 | 30.90 | 18.55 | 1251.76 | 15000 | 15000 | 3901.53 | 41040.5 | 52262 | 220 | 68 |
| Simonis_problem_30_10 | 79.45 | 24.09 | 3919.72 | 4330 | 15000 | 14941.28 | 16701.0 | 62324 | 550 | 437 |
| Simonis_problem_30_15 | 24.09 | 26.84 | 2144.90 | 15000 | 15000 | 10059.94 | 49759.0 | 96676 | 220 | 53 |
| Simonis_problem_30_30 | 21.82 | 29.18 | 132.92 | 15000 | 15000 | 288.96 | 39811.0 | 4893 | 110 | 24 |
| Simonis_problem_40_20 | 16.36 | 37.83 | 395.00 | 15000 | 15000 | 1172.44 | 34265.0 | 11890 | 110 | 18 |
| Miller_Miller19 | 0.00 | 20.00 | -1.00 | 6900 | 15000 | -1.00 | 6581.0 | 0 | 1 | 0 |
| Wilson_gp50by50 | 25.00 | 46.50 | 6900.00 | 15000 | 15000 | 6581.00 | 23722.5 | 6581 | 4 | 1 |
| Wilson_gp100by100 | 0.00 | 95.75 | -1.00 | 15000 | 15000 | -1.00 | 15260.5 | 0 | 4 | 0 |
| Wilson_nwrsSmaller4 | 50.00 | 7.75 | 7155.00 | 15000 | 15000 | 17922.00 | 19043.0 | 35105 | 4 | 2 |
| Wilson_nrwsLarger4 | 0.00 | 20.75 | -1.00 | 15000 | 15000 | -1.00 | 24236.5 | 0 | 4 | 0 |
| Wilson_sp4 | 0.00 | 46.50 | -1.00 | 15000 | 15000 | -1.00 | 15390.0 | 0 | 4 | 0 |
| Shaw_ShawInstances | 0.00 | 16.28 | -1.00 | 15000 | 15000 | -1.00 | 39345.0 | 0 | 25 | 0 |

# 5 Appendix: source code

```
solve(Matrix, Ps, NbStacks) :-
problem(Matrix, Ps, PsL, NbStacks, ValOrder), min_max(1000, NbStacks, [], PsL, Ps, ValOrder).

min_max(NbStacks0, NbStacks, PsL0, PsL, Ps, ValOrder) :-
NbStacks #< NbStacks0,
findall(f(PsL,Ps,NbStacks), label(PsL,NbStacks,ValOrder,PsL0), [f(PsL1,Ps1,NbS1)]), !,
format('new incumbent Ps=~q Cost=~q\n', [Ps1,NbS1]),
min_max(NbS1, NbStacks, PsL1, PsL, Ps, ValOrder).
min_max(NbStacks, NbStacks, PsL, PsL, _, _).

label(Ps, NbStacks, ValOrder, Oracle) :- middle_out_labeling(Ps, ValOrder, Oracle), indomain(NbStacks), !.

middle_out_labeling([], _, _).
middle_out_labeling([X|Xs], Order1, []) :- !,
select(X, Order1, Order2),
middle_out_labeling(Xs, Order2, []).
middle_out_labeling([X|Xs], Order1, [O|Oracle1]) :-
suffixchk(Order1, [O|AfterO]),
(   X=O, Oracle2 = Oracle1
;   member(X, AfterO), Oracle2 = []
),
selectchk(X, Order1, Order2), middle_out_labeling(Xs, Order2, Oracle2).

split(N, N, []) --> !.
split(I, N, [X|Xs]) --> [X], {J is I+1}, split(J, N, Xs).

problem(Matrix, PsOrig, Ps4, NbStacks, ValOrder) :-
nonzero_rows(Matrix, Rows1),
length(Rows1, NbO), transpose(Rows1, RowsT1),
length(RowsT1, NbCols), length(PsOrig, NbCols),
keys_and_values(RowsT2, RowsT1, PsOrig), !,
keysort(RowsT2, RowsT3), keyclumps(RowsT3, RowsT4),
merge_clumps(RowsT4, RowsT5, Ps1), transpose(RowsT5, Rows5),
% Rows5  is the original matrix, lex-sorted by column, identical columns removed.
% PsOrig are the permutation variables corresponding to the original matrix.
% Ps1    are the permutation variables corresponding to Rows5.
length(Ps1, NbP), domain(Ps1, 1, NbP),
Mid is (NbP+2)>>1, Mid1 is Mid-1, NbP1 is NbP+1,
for(Mid, 1, NbP1, L1, []), for(Mid1, -1, 0, L2, []), splice(L1, L2, ValOrder, []),
all_distinct(Ps1, [on(minmax),consistency(bound)]), NbP1 is NbP+1,
orders_tasks(Rows5, Ps1, NbP1, Tasks, [task(1,NbP,NbP1,H,1)]),
tag_by_sum(RowsT5, Ps1, Ps2), % order Ps by decreasing #1s
keysort(Ps2, Ps3), keys_and_values(Ps3, _, Ps4), Ps3 = [(_-LB)-P1,_-P2|_],
P1 #> P2,                       % symmetries, V. 2
NbStacks in LB..NbO, NbStacks + H #= NbO, % lower bound
cumulatives(Tasks, [machine(1,NbO)], [bound(upper)]),
dominance(RowsT5, Ps1, Tasks), row_subsets(Rows5, 0, Pairs, []), post_subsets(Pairs, Tasks),
% redundant_cumulatives(0, NbP, RowsT5, Ps1, Tasks),
true.

nonzero_rows([], []).
nonzero_rows([R|Rs1], [R|Rs2]) :- memberchk(1, R), !, nonzero_rows(Rs1, Rs2).
nonzero_rows([_|Rs1], Rs2) :- nonzero_rows(Rs1, Rs2).

for(End, _, End) --> !.
for(Cur, Step, End) --> [Cur], {Next is Cur+Step}, for(Next, Step, End).

splice([], []) --> [].
splice([A], []) --> !, [A].
splice([A|As], [B|Bs]) --> [A,B], splice(As, Bs).

row_subsets([], _) --> [].
row_subsets([Row|Rows], I) --> {J is I+1}, row_subsets(Rows, Row, J, J), row_subsets(Rows, J).

row_subsets([], _, _, _) --> [].
row_subsets([Row2|Rows], Row1, I1, I2) -->
{J2 is I2+1}, ({subset_01(Row1, Row2, 0, _)} -> [I1-J2] ; []), row_subsets(Rows, Row1, I1, J2).

post_subsets([], _).
post_subsets([I-J|Pairs], Tasks) :-
nth1(I, Tasks, task(O1,D1,E1,_,_)), nth1(J, Tasks, task(O2,D2,E2,_,_)),
D1 #=< D2, O1 #>= O2, E1 #=< E2, post_subsets(Pairs, Tasks).

redundant_cumulatives(NbP, NbP, _, _, _).
redundant_cumulatives(I, NbP, Cols, Ps, Tasks1) :-
J is I+1, nth1(J, Cols, Col), sumlist(Col, H),
ps_of_supersets(Cols, Ps, Col, Pxs, []), length(Pxs, W),
(   W=:=1 -> true
;   H=<1 -> true
;   min_and_max1(Pxs, Org, End),
    Dur in W..1000,
    redundant_cumulatives_tasks(Col, Tasks1, Tasks2, [task(Org,Dur,End,H,1)]),
    length(Col, NbO),
    cumulatives(Tasks2, [machine(1,NbO)], [bound(upper)]),
    % format('redundant cumulatives, column=~d #rows=~d #columns=~d\n', [J,H,W]),
    true
),
redundant_cumulatives(J, NbP, Cols, Ps, Tasks1).
```

```
ps_of_supersets([], [], _) --> [].
ps_of_supersets([Col2|Cols], [P|Ps], Col1) -->
({subset_01(Col1, Col2, 0, _)} -> [P] ; []), ps_of_supersets(Cols, Ps, Col1).

redundant_cumulatives_tasks([], [T]) --> [T].
redundant_cumulatives_tasks([0|Col], [T|Ts]) --> !, [T], redundant_cumulatives_tasks(Col, Ts).
redundant_cumulatives_tasks([1|Col], [_|Ts]) --> redundant_cumulatives_tasks(Col, Ts).

% Dominance constraints for any two columns i,j such that
% orders(i) + q = orders(j):
% NOT(First(q)<Pi<Pj)
% NOT(First(q)=Pj<Pi<Last(q))
dominance(RowsT5, Ps1, Tasks) :-
dominance_items(RowsT5, Ps1, Tasks, Items, []), dominance_constraints(Items).

dominance_items([], [], _) --> [].
dominance_items([Coli|Cols], [Pi|Ps], Tasks) --> dominance_items(Cols, Ps, Coli, Pi, Tasks).

dominance_items([], [], _, _, _) --> [].
dominance_items([Colj|Cols], [Pj|Ps], Coli, Pi, Tasks) -->
(   {subset_01(Coli, Colj, 0, [Q])}
->  {nth1(Q, Tasks, Task)},
    [item(Pi,Pj,Task)]
;   []
), dominance_items(Cols, Ps, Coli, Pi, Tasks).

subset_01([], [], _, []).
subset_01([X|L1], [X|L2], I, L3) :- !, J is I+1, subset_01(L1, L2, J, L3).
subset_01([0|L1], [1|L2], I, [J|L3]) :- J is I+1, subset_01(L1, L2, J, L3).

dominance_constraints([]).
dominance_constraints([item(Pi,Pj,task(O,_,E,_,_))|Items]) :-
Pi #=< Pj #=> O #>= Pi,  Pi #>= Pj #/\ O #= Pj #=> E #=< Pi,
dominance_constraints(Items).

merge_clumps([], [], []).
merge_clumps([Clump|Clumps], [Col|Cols], [P|Ps]) :-
Clump = [Col-P|R], merge_vars(R, P), merge_clumps(Clumps, Cols, Ps).

merge_vars([], _).
merge_vars([_-X|Xs], X) :- merge_vars(Xs, X).

tag_by_sum([], [], []).
tag_by_sum([Col|Cols], [P|Ps1], [(W-Sum)-P|Ps2]) :-
sumlist(Col, Sum), W is -Sum, tag_by_sum(Cols, Ps1, Ps2).

tag_by_weighted_sum([], [], [], _).
tag_by_weighted_sum([Col|Cols], [P|Ps1], [(W-Sum)-P|Ps2], Weights) :-
sumlist(Col, Sum), weighted_sum(Col, Weights, 0, WSum), W is -WSum,
tag_by_weighted_sum(Cols, Ps1, Ps2, Weights).

weighted_sum([], [], S, S).
weighted_sum([X|Xs], [Y|Ys], S1, S3) :- S2 is S1 + X*Y, weighted_sum(Xs, Ys, S2, S3).

rows_sums([], []).
rows_sums([Row|Rows], [Sum|Sums]) :- sumlist(Row, Sum), rows_sums(Rows, Sums).

orders_tasks([], _, _) --> [].
orders_tasks([O1|Os], Ps, NbPl) --> [task(Org,Dur,End,1,1)],
{order_task(O1, Ps, Org, End)}, {sumlist(O1, LB)}, {Dur in LB..NbPl},
orders_tasks(Os, Ps, NbPl).

order_task(Bs, Ps, Org, End) :- order_ps(Bs, Ps, Ps1, []), min_and_max1(Ps1, Org, End).

order_ps([], []) --> [].
order_ps([0|Bs], [_|Ps]) --> !, order_ps(Bs, Ps).
order_ps([1|Bs], [P|Ps]) --> [P], order_ps(Bs, Ps).

suffixchk(List, List) :- !.
suffixchk([_|List], Suffix) :- suffixchk(List, Suffix).

min_and_max1(Ps1, Org, End) :- min(Ps1, Org), max1(Ps1, End).

min([X], X) :- !.
min(Row, Min) :- element(_, Row, Min), ge_each(Row, Min).

max1([Max], Max1) :- !, Max+1 #= Max1.
max1(Row, Max1) :- element(_, Row, Max), le_each(Row, Max), Max+1 #= Max1.

ge_each([], _).
ge_each([X|Xs], Min) :- X #>= Min, ge_each(Xs, Min).

le_each([], _).
le_each([X|Xs], Max) :- X #=< Max, le_each(Xs, Max).
```

# Constraint Modelling Challenge 2005
# A dynamic programming approach

Thierry Benoist
Bouygues e-lab, 1 av. Eugène Freyssinet,
78061 St Quentin en Yvelines Cedex, France
tbenoist@bouygues.com

## The problem

*A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customer's order is started (i.e. the first product in the order has been made) a stack is created for that customer. When all the products that a customer requires have been made, the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the number of stacks that are in use simultaneously i.e. the number of customer orders that are in simultaneous production should be minimized.*
*More formally: we are given a Boolean matrix in which the columns correspond to the products required by the customers and each row corresponds to the order of a particular customer. The entry c_ij = 1 iff customer i has ordered some quantity of product j (the quantity ordered is irrelevant). The objective is to find a permutation of the products such that the maximum number of open orders at any point in the sequence is minimized: order i is open at point k in the production sequence if there is a product required in order i that appears at or before position k in the sequence and also a product that appears at or after position k in the sequence*

We have $n$ customers (indexed by i) and $m$ products (indexed by j). We have to find a permutation of {1,…$m$}. Let $\sigma_j$ be the position of product j in the sequence.
P(i) is the set of products of customer i
C(j) is the set of customers ordering product j

## Dynamic Programming

Given a solution (permutation) $\sigma$, the number of open commands at position $t$ only depends on the product $j$ attached to position $t$ and on the set of products $S_t$ attached to previous positions (up to $t$-1). Indeed open commands are C(j) $\cup$ O($S_t$) with O($S_t$)={i $\in$ [1..$n$] , P(i)$\cap S_t \notin$ {$\emptyset$,P(i)}}, in other words commands containing product $j$ (namely $C(j)$) and commands with some products in $S_t$ but not all (namely O($S_t$)). It is important to note that the actual permutation of $S_t$ has no impact on the number of open commands at position $t$.
Therefore we can design a dynamic program with $2^m$ states, corresponding to all possible subsets of [1..$m$]. Any state $S$ can be reached from $|S|$ different states[1] with one product less. If we denote by $f(S)$ the objective value corresponding to the best permutation of products of S, then S can be recursively written as:

$$\begin{cases} f(S) = \min_{j \in S} \left( \max\left( f(S - \{j\}), \left| C(j) \cup O(S - \{j\}) \right| \right) \right. \\ f(\emptyset) = 0 \end{cases},$$

and $f([1..m])$ is the optimal solution of the problem.

## Complexity

Space complexity is $2^m$. More precisely, for problems of size n<64, m=30, we need 6x$10^9$ bits in memory.
The computation algorithm reads as follows:

For S in subsets ([1..$m$]) // $2^m$ subsets
  Compute O(k), scanning all commands and intersecting their set of product with k
  Then for all products $j$ (at most $m$, $m/2$ in average), compute the union of O(k) and P(j)
  Then get the cardinality of this set and update $f(S \cup \{j\})$ if necessary

The time complexity of this algorithm is O($nm2^m$). However since space complexity limits the size of tractable instances (with this approach) it is intersecting to focus on the case $n$ and $m < 64$ since in most programming languages it allows performing intersections, union and even cardinality[2] operations in constant time. For such instance the number of operations is around $(n+m)2^m$. In practice computations

[1] |X| representing the cardinality of set X
[2] Having precomputed the number of bits of all 256x256 bitvectors of size ≤ 16

times are <1ms when m=10, around 1s when m=20, around 30mn when m=30. Larger problem cannot be solved with this approach.

**Future work**

This dynamic program may help solving larger instances. For instance it could explore in one second nodes of depth *m*-20 in a chronological branch and bound [Benoist & Cambazard, *ongoing work*]. Or it could optimize any window of 20 columns in a local search approach…

**Table 1. Aggregate results (Thierry Benoist)**

- Java Program on PC 2.4Ghz, 1Go RAM.
- One run per instance.
- No measure of search effort (no backtrack)
- Max runtime 2 seconds.

All sets of instances with m ≤ 20 are solved. For results on problems with m=30 see table 2.

| File | % solved optimally within the cutoff limit | Mean best value found | Total runtime per instance in **milliseconds** | | |
|---|---|---|---|---|---|
| | | | mean | median | max |
| problem_10_10.dat | 100% | 8.0309 | 0.56909 | 0 | 31 |
| problem_10_20.dat | 100% | 8.9218 | 830.309 | 828 | 1031 |
| problem_15_15.dat | 100% | 12.869 | 24.4364 | 16 | 78 |
| problem_15_30.dat | | | | | |
| problem_20_10.dat | 100% | 15.878 | 0.87636 | 0 | 16 |
| problem_20_20.dat | 100% | 17.973 | 993.545 | 985 | 1219 |
| problem_30_10.dat | 100% | 23.953 | 0.98727 | 0 | 31 |
| problem_30_15.dat | 100% | 25.968 | 31.1818 | 31 | 63 |
| problem_30_30.dat | | | | | |
| problem_40_20.dat | 100% | 6.4727 | 1296.17 | 1297 | 1469 |
| ShawInstances.txt | 100% | 13.68 | 992.52 | 985 | 1047 |
| wbo_10_10.txt | 100% | 5.925 | 2.79487 | 0 | 16 |
| wbo_10_20.txt | 100% | 7.35 | 835.175 | 829 | 860 |
| wbo_10_30.txt | | | | | |
| wbo_15_15.txt | 100% | 9.35 | 24.75 | 31 | 47 |
| wbo_15_30.txt | | | | | |
| wbo_20_10.txt | 100% | 12.9 | 0 | 0 | 0 |
| wbo_20_20.txt | 100% | 13.689 | 993.189 | 985 | 1016 |
| wbo_30_10.txt | 100% | 20.05 | 0.48 | 0 | 16 |
| wbo_30_15.txt | 100% | 20.958 | 31.675 | 31 | 62 |
| wbo_30_30.txt | | | | | |
| wbop_10_10.txt | 100% | 6.75 | 0.8 | 0 | 16 |
| wbop_10_20.txt | 100% | 8.075 | 838.625 | 828 | 937 |
| wbop_10_30.txt | | | | | |
| wbop_15_15.txt | 100% | 10.367 | 24.4833 | 16 | 62 |
| wbop_15_30.txt | | | | | |
| wbop_20_10.txt | 100% | 14.275 | 0.775 | 0 | 16 |
| wbop_20_20.txt | 100% | 14.867 | 1009.69 | 1000 | 1110 |
| wbop_30_10.txt | 100% | 22.475 | 0.775 | 0 | 16 |
| wbop_30_15.txt | 100% | 22.383 | 33.8333 | 31 | 63 |
| wbop_30_30.txt | | | | | |
| wbp_10_10.txt | 100% | 7.275 | 1.95 | 0 | 47 |
| wbp_10_20.txt | 100% | 8.7143 | 830.8 | 828 | 875 |
| wbp_10_30.txt | | | | | |
| wbp_15_15.txt | 100% | 11.05 | 24.95 | 16 | 62 |
| wbp_15_30.txt | | | | | |
| wbp_20_10.txt | 100% | 15.125 | 2.675 | 0 | 31 |
| wbp_20_20.txt | 100% | 15.411 | 1009.36 | 1000 | 1578 |
| wbp_30_10.txt | 100% | 23.175 | 1.175 | 0 | 16 |
| wbp_30_15.txt | 100% | 22.983 | 33.3167 | 31 | 63 |
| wbp_30_30.txt | | | | | |

**Table 2. Individual results (Thierry Benoist)**

Same program as in section 1. Complete search method

| Instance | Best objective value found | Proved optimal? | Runtime | Search effort to find optimal solution | Total search effort |
|---|---|---|---|---|---|
| Miller19 | | | | | |
| GP1 | | | | | |
| GP2 | | | | | |
| GP3 | | | | | |
| GP4 | | | | | |
| GP5 | | | | | |
| GP6 | | | | | |
| GP7 | | | | | |
| GP8 | | | | | |
| NWRS1 | 3 | Yes | 0.7s | | |
| NWRS2 | 4 | Yes | 0.8s | | |
| NWRS3 | 7 | Yes | 31s | | |
| NWRS4 | 7 | Yes | 30s | | |
| NWRS5 | 12 | Yes | 24mn | | |
| NWRS6 | 12 | Yes | 21mn | | |
| NWRS7 | | | | | |
| NWRS8 | | | | | |
| SP1 | 9 | Yes | 36s | | |
| SP2 | | | | | |
| SP3 | | | | | |
| SP4 | | | | | |

# Dynamic Programming to Minimize the Maximum Number of Open Stacks

**M. Garcia de la Banda**
School. of Comp. Sci. & Soft. Eng.
Monash University, 6800, Australia
mbanda@csse.monash.edu.au

**P. J. Stuckey**
NICTA Victoria Laboratory
Dept. of Comp. Sci. & Soft. Eng.
University of Melbourne, 3010, Australia
pjs@cs.mu.oz.au

## Abstract

We argue that a complete method for the Open Stacks problem should be based on dynamic programming. Starting from a call based dynamic program, we show a number of ways to improve the dynamic programming search, preprocess the problem to simplify it, and to determine lower and upper bounds. We then explore a number of search strategies for reducing the search space. The final dynamic programming solution is, we believe, highly effective.

## 1 Introduction

The Open Stacks problem can be phrased as follows: Let $P$ be a set of products, $C$ a set of customers, and let us assume that the products ordered by customer $c_i \in C$ are placed in stack $i$ satisfying $\forall c_i, c_j \in C, c_i \neq c_j : i \neq j$. Customer $c_i$ is active (or stack $i$ is open) from the time the first product ordered by $c_i$ is built until the last product ordered by $c_i$ is built. The Minimization of Open Stacks Problem (MSOP) [3] aims at finding an order for building the products in $P$ which minimises the maximum number of customers active (or of open stacks) at any time.

## 2 Dynamic Programming Formulation

The MSOP problem is naturally expressible in a dynamic programming formulation. Let $c(p)$ be the set of customers ordering product $p \in P$, and $c(S) = \cup_{p \in S} c(p)$ be the set of customers ordering products from set $S \subseteq P$. Assume that product $p$ is built immediately before any product from set $A \subset P$, and after any other remaining product ($P - A - \{p\}$). Then, the set of active customers at the time $p$ is built are

$$a(p, A) = c(p) \cup (c(A) \cap c(P - A - \{p\}))$$

i.e., those who ordered $p$, plus those whose orders include some products scheduled before $p$ and some scheduled after. Crucially, $a(p, A)$ does not depend on any particular order of the products in A or $P - A - \{p\}$. Let $stacks_P(S)$ be the minimum number of stacks required to schedule the set of products $S$ assuming that those in $P - S$ are scheduled earlier. Dynamic programming can be used to define $stacks_P(S)$ as:

$$stacks_P(S) = \min_{p \in S} \max\{a(p, S - \{p\}), stacks_P(S - \{p\})\}$$

Dynamic programming is so effective for this problem because it reduces the raw search space from $|P|!$ to $2^{|P|}$, since we only need to investigate minimum stacks for each subset of $P$. Note also that dynamic programming is completely equivalent to a constraint logic programming approach with memoing. Although our implementation does not use a CP system, it certainly can be considered a CP approach.

The following code illustrates our $A^\star$ call based dynamic programming algorithm, which improves over a naive dynamic programming formulation by taking into account lower $L$ and upper bounds $U$:

```
stacks(S, L, U)
  if (S = ∅) return 0
  if (stack[S]) return stack[S]
  min := U + 1
  T := S
  while (min > L and T ≠ ∅)
    p := index min{a(p, S − {p}) | p ∈ T}
    T := T − {p}
    if (a(p, S − {p}) ≥ min) break
    sp := max(a(p, S − {p}), stacks(S − {p}, L, U))
    if (sp < min) min := sp
  stack[S] := min
  if (min > U) FAIL := FAIL ∪ {S}
  else SUCCESS := SUCCESS ∪ {S}
  return min
```

The algorithm starts by checking whether $S$ is empty, in which case 0 stacks are needed. Otherwise, it checks whether the minimum number of stacks for $S$ has already been computed (and stored in $stack[S]$), in which case it returns the previously stored result (code shown in light grey). If not, the algorithm basically computes in $sp$ the value $\max(a(p, S - \{p\}), \mathsf{stacks}(S - \{p\}), L, U)$ for each $p \in S$, and updates the current minimum in $min$ if required. Note, however, that this computation is avoided (thanks to the **break**) for products whose active set of customers is greater or equal than the current minimum $min$, since they cannot improve on the current solution. As a result, the order in which the products in $S$ are tried will affect the amount of work performed by the algorithm. The simple heuristic embedded in our algorithm selects the product $p$ which would have the least active

customers if scheduled immediately. The loops also stops as soon as the current solution equals the lower bound, since we are only interested in finding one best solution.

The dark grey code stores in $SUCCESS$ the sets which resulted in finding a solution within the bounds, and in $FAIL$ those which did not. We will make use of these later.

Calling stacks$(P, L, U)$ returns the minimal number of stacks required to schedule the products $P$ assuming a lower bound $L$ and upper bound $U$. Extracting the optimal solution found from $stack[]$ is straightforward, and standard for dynamic programming.

We can improve the code above by noticing that when computing $stacks_P(S)$, the open stacks initially are given by $o(S) = c(P - S) \cap c(S)$. If we have a product $p \in S$ where $c(p) \subseteq o(S)$, then there must be a solution to $stacks_P(S)$ which starts with $p$.

**Lemma 1** *If there exists $p \in S$ where $c(p) \subseteq o(S)$ then there is an optimal order for $stacks_P(S)$ beginning with $p$.*

**Example 1** Consider the following open stacks problem

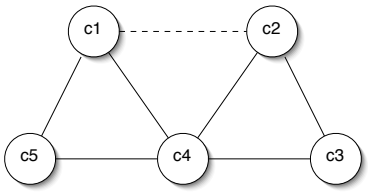
| | p1 | p2 | p3 | p4 | p5 | p6 | p7 |
|---|---|---|---|---|---|---|---|
| c1 | X | . | . | . | X | . | X |
| c2 | X | . | . | X | . | . | . |
| c3 | . | X | . | X | . | X | . |
| c4 | . | . | X | X | . | X | X |
| c5 | . | . | X | . | X | . | . |

Consider scheduling $S = \{p1, p2, p3, p4, p6\}$ after $\{p5, p7\}$ have been scheduled. Then, $o(S) = \{c1, c4, c5\}$ and an optimal schedule can begin with $p3$ since $c(p3) \subseteq o(S)$. □

We can improve the search further using the following argument about the minimal number of stacks required. Define the *customer graph* $G = (V, E)$ for an open stacks problem as: $V = c(P)$ and $E = \{(c_1, c_2) \mid \exists p \in P, \{c_1, c_2\} \subseteq c(P)\}$. That is, nodes represent customers, and nodes are adjacent if they order the same product. Let $d_G(c)$ be the degree of node $c$ in $G$.

**Lemma 2** *The minimal number of stacks required for set of products $S$ is at least $b(S) = |o(S)| + \min\{d_{G'}(c) \mid c \in c(S), G' = (V, E - \{(c_1, c_2)|\{c_1, c_2\} \subseteq c(P - S)\})\}$.*

**Example 2** The customer graph for Example 1 is




Consider scheduling the set $S = \{p2, p3, p4, p5, p6, p7\}$ after $\{p1\}$. The open stacks are $o(S) = \{c1, c2\}$. The reduced customer graph removes the dashed arc between $c1$ and $c2$. The remaining degrees are: 2,2,2,4,2 respectively. $b(S) = |\{c1, c2\}| + 2 = 4$. This is a lower bound on a schedule for $S$, since closing any customer requires at least this many open stacks. □

We can use this to improve the $A^\star$ algorithm above. We replace the calculation $a(p, A)$ with

$$a'(p, A) = \max\{a(p, A), b(A)\}$$

which gives an improved lower bound on the future number of stacks required.

## 3 Preprocessing

Our methodology attempts to simplify the problem by applying two preprocessing steps to the initial $P$. The first step removes from $P$ any product $p'$ such that $c(p') \subseteq c(p)$ for some $p$ appearing in the reduced problem. Solving the reduced problem gives an optimal value for $P$, and optimal solutions to the reduced problem can be extended to give optimal solutions to $P$ by simply placing each $p'$ immediately after the $p$ that subsumed it.

This was also noted (although not proved) in Becceneri *et al*. [1]. We can prove it using Lemma 1. Simply note that if $c(p') \subseteq c(p)$ then any order for $S$ including $p'$ but not including $p$ must have $c(p') \subseteq o(S)$. Because the problem is the same when considering the reverse order, the same holds for orders with $p'$ before $p$.

**Example 3** Consider the open stacks problem from Example 1. Since $c(p2) \subseteq c(p4)$ and $c(p6) \subseteq c(p4)$, the two products can be removed. Inserting them after $p4$ in an optimal order for the reduced set of products, gives an optimal order for the original problem.

| | p5 | p7 | p3 | p1 | p4 | p2 | p6 |
|---|---|---|---|---|---|---|---|
| c1 | X | X | – | X | . | . | . |
| c2 | . | . | . | X | X | . | . |
| c3 | . | . | . | . | X | X | X |
| c4 | . | X | X | – | X | – | X |
| c5 | X | – | X | . | . | . | . |

□

Our second preprocessing step is more obvious: if $P$ can be partitioned into two sets $P = P_1 \cup P_2$ such that $c(P_1) \cap c(P_2) = \emptyset$, then we can independently order $P_1$ followed by $P_2$. This is noted by Yeun and Richardson [3]. We thought this was too unrealistic to occur, but it does occur in several benchmarks, including some of the mildy difficult ones.

Becceneri *et al* [1] reference techniques for handling tree like sub-graphs of the customer graph independently, but the paper is not available (and is in Portugese!).

## 4 Bounds

Our $A^\star$ algorithm uses both upper and lower bounds to reduce the number of subsets visited. Trivial lower and upper bounds are $L = \max\{|c(p)| \mid p \in P\}$ and $U = |C|$.

Our approach tries to improve the lower bound by analysing the customer graph.

**Lemma 3** *If $Q \subseteq C$ is clique in the customer graph $G$, the minimal number of open stacks is at least $|Q|$*

We independently determined this lower bound before finding [1] where they explain a more general approach to calculating lower bounds. They introduce the following lower bound without proof.

**Lemma 4** *If $d = 1 + \min\{d_G(c) \mid c \in C\}$ then $d$ is a lower bound on the open stacks for the problem.*

A *minor* of an open stacks problem can be obtained by either removing an entire customer $c \in C$ (replacing $c(p)$ by $c'(p) = c(p) - \{c\}$), or merging two adjacent customers $(c_1, c_2) \in E$ (replacing each product order $c(p)$ by $c'(p) = (c(p) - \{c_2\}) \cup \{c_1\}$ if $c_2 \in c(p)$, or by $c'(p) = c(p)$ otherwise). These operations correspond to edge contraction or node elimination from the customer graph $G_c$.

**Lemma 5** *Let $m$ be the minimal open stacks for a problem defined by $c(p)$, and $m'$ be the minimal open stacks for the problem defined by $c'(p)$ where $c'$ is a minor of $c$. Then $m' \leq m$.*

With these two lemmas we apply any number of minor steps and use the size of the minimum degree node + 1, as a lower bound for the original problem. Note that we can stop when the remaining customer graph is a clique.

Becceneri *et al.* [1] define a heuristic arc contraction approach (HAC) based on this, but provide no proof of correctness. We built an implementation of Becceneri *et al.*'s algorithm and a greedy clique finder (that doesn't do contractions but tries starting from each products set of customers).

In order to improve the upper bound we run a number of greedy heuristics of the following general form.

heuristic$(S)$
  $min$ := 0
  **while** $(S \neq \emptyset)$
    heuristically select $p$
    $S := S - \{p\}$
    **if** $(a(p, S - \{p\}) > min)$ $min := a(p, S - \{p\})$
  **return** $min$

We experimented with eleven heuristics, with the five most successful over all benchmark instances being:
(1) Yuen's hueristic 3 [2]: selects at each stage the product $p$ whose intersection of customers with previous active stacks minus the number of new stacks is maximized

$$\text{index} \max_{p \in S} |c(p) \cap c(P - S)| - |c(p) - c(P - S)|.$$

(2) minimizes the number of active stacks and breaks ties in favor of products that close greater number of stacks (are the last product in those stacks)

$$\text{index} \min_{p \in S} (a(p, S - \{p\}), -|c(p) - c(S - \{p\})|).$$

(3) minimizes the number of active stacks except (a) all active stack numbers less than the current $min$ are considered equivalent, and (b) ties are again broken in favor of products that close more stacks.

$$\text{index} \min_{p \in S} (\max(min, a(p, S - \{p\})), -|c(p) - c(S - \{p\})|).$$

(4) minimizes the number of active stacks and breaks ties by maximizing a cost given by $\Sigma_{c \in c(p)} 2^{-|n(S,c)|}$ where $n(S, c)$ is the number of products $p' \in S$ for which customer $c$ appears in $c(p')$. This effectively assigns to each customer with $m$ ordered products a cost of (almost) 1 split amongst its products as follows: $2^{-m}$ for the first scheduled product, $2^{-m+1}$ for the second, ..., $2^{-2}$ for the second last, and $2^{-1}$ for the last product.

$$\text{index} \min_{p \in S} (a(p, S - \{p\}), -\Sigma_{c \in c(p)} 2^{-|n(S,c)|})$$

(5) minimizes the maximum of the number of active stacks required using the improved formula $a'(p, A)$.

$$\text{index} \min_{p \in S} a'(p, S - \{p\})$$

We also implemented the minimal cost node heuristic (which we'll denote (6)) of Becceneri *et al* [1] which does not follow the general greedy format since it selects arcs (not products) in the customer graph to determine a product order.

## 5 Search Strategies

Our $A^\star$ program is particularly effective when called with $L = U = n$, where it only explores schedules which use exactly $n$ active stacks. This is related to the fact that the problem is *fixed parameter tractable*. This immediately suggests an extended search procedure where we successively try each possible value from the lower to the upper bound:

stepwise$(L, U)$
  **for** $try := L$ to $U$
    $FAIL := \emptyset$
    $min :=$ stacks$(P, try, try)$
    **if** $(min \leq try)$ **return** $min$
    **for** $(S \in FAIL)$ $stack[S] := 0$

Note that if stacks returns a number different from $U+1$, this is the optimal value and can be reused in later computations. We need to reset the memoed values for FAILed sets (when $stack[S] > U$), since they must have failed because the upper bound was too low.

We can improve upon this search using binary search. The following code

binarychop$(L, U)$
  $gmin := U + 1$
  **while** $(L \leq U)$
    $FAIL := SUCCESS := \emptyset$
    $try := (L + U)$ $div$ 2
    $min :=$ stacks$(P, try, try)$
    **if** $(min \leq try)$
      $gmin := min$
      $U := min - 1$
      **for** $(S \in FAIL \cup SUCCESS)$ $stack[S] := 0$
    **else**
      $L := try + 1$
      **for** $(S \in FAIL)$ $stack[S] := 0$
  **return** $gmin$

repeatedly tries the midpoint of the current range. If successful, it tries values below it after removing all $stack[S]$ computations performed for stacks$(P, try, try)$ (but not those previously calculated and used by this computation), since they could be too high or too low. If unsuccessful, it tries values above it, after removing FAILed stored values.

Finally, we noted that often the most expensive stack number to try was the stack number below the optimal, and those above the optimal were usually easier than those below. This

motivated a backwards stepwise approach where the possible stack numbers are tried in decreasing order:

```
backwards(L, U)
  try := U
  while (try ≥ L)
    FAIL := SUCCESS := ∅
    min := stacks(P, try, try)
    if (min > try) return gmin
    gmin := min
    try := min − 1
    for (S ∈ FAIL ∪ SUCCESS) stack[S] := 0
```

This has another advantage: we can stop at any time with a (non-optimal) solution. Note that since stacks can return a value less than $try$ we do not just decrease the $try$ by one each time, but to under the minimum we last found.

## 6 Experimental Results

In this section we briefly describe the effect of the preprocessing approaches, lower and upper bounds approaches and searching approaches.

We compare on all benchmarks except the most difficult: SP2, SP3, SP4, which none of our versions can finish in time. In order to compare the different search approaches we show the total number of calls to stacks to optimally solve each instance (except SP2, SP3, SP4) for each search strategy with all optimizations enabled, and then backwards with some optimizations disabled individually. The appendix shows the results for all benchmarks.

| Search method | Total calls to stacks | Total time (secs) |
|---|---|---|
| $A^\star$ | 56,231,534 | 1386 |
| stepwise | 29,887,854 | 880 |
| binarychop | 25,351,370 | 715 |
| backwards | 21,271,366 | 572 |
| backwards −definite | 33,992,526 | 1,260 |
| backwards $-a'(p, A)$ | 169,638,021 | 441 |
| backwards −redundant | 30,166,640 | 850 |
| backwards −red −def | 70,759,348 | 2231 |
| backwards −partition | 21,275,426 | 573 |
| backwards −upper | 21,298,294 | 570 |
| backwards −lower | 21,452,365 | 575 |

The dynamic programming code is written in C, with no great tuning or clever data structures, and many runtime flags to allow us to compare the different versions easily.

First of all the definite choice optimization of Lemma 1 is highly beneficial. The total number of calls to stacks reduces by 1/3 but the time halves since we avoid search for the best possible candidate.

The improved search offered by the use of $a'(p, A)$ instead of $a(p, A)$ is massive. The search reduces by an order of magnitude. But because we havent attempted a very clever implementation of $a'(p, A)$ execution is slower, since using $a(p, A)$ we can have a very tight inner loop.

Removing redundant products $p'$ where $c(p') \subseteq c(p)$ for another product $p$ is an important first step. Over the benchmark suite we remove 16305 redundant products out of 101385 total products, a 16% reduction in size on average. Given that each extra product could in the worst case double the search space, this is vital. This is masked by using the definite choice optimization, if both are removed the program fails to solve NWRS8 which has 20 redundant products out of 60.

There are 113 instances where the products are separable (which surprised us somewhat), with 2.42 separate parts on average. In most cases the result of separating is not much better than not, since the separable partitions are usually tiny singletons. But there are examples such as Warwick_1711 where the search space reduces from 1853 calls to stacks to 183, even though the separable parts are size 1, 1 and 5 out of 29 nonredundant products.

The effect of the upper bound heuristics are not too great once we use backwards. They improve the number of sets in 884 cases, but the percentage improvement is tiny overall (0.0012%) since they do not improve any of the really hard benchmarks by more that a tiny fraction. Comparatively, the heuristics rank in the order (1) to (6) (worst to best). Of 5964 partitions of products for 5803 problems, the following table shows the number of times each heuristic returned the (equal) *best* answer of all heuristics, the *unique* best answer (bettered all others), the number of times the answer was the *optimal* answer to the problem (of 5803), and the total *sum* of the heuristic results is shown.

| heur | (1) | (2) | (3) | (4) | (5) | (6) |
|---|---|---|---|---|---|---|
| best | 3046 | 3596 | 3615 | 3840 | 5073 | 5446 |
| unique | 29 | 1 | 7 | 43 | 159 | 514 |
| optimal | 2733 | 3231 | 3248 | 3458 | 4608 | 4986 |
| sum | 98167 | 96798 | 96769 | 96462 | 94592 | 94093 |

Although the lower bounds approaches are very successful at finding good lower bounds, the only time they can improve the backwards approach is when the lower bound is the optimal. While this occurs frequently it does not occur on the hard benchmarks so there is little benefit. The HAC heuristic is never improved by the clique approach. The clique lower bound gives the optimal answer in 2718 benchmarks of 5803, while the HAC approach gives the optimal on 3380.

While the lower and upper bounds are not that useful for backwards, this is certainly not the case for $A^\star$, stepwise or binarychop. Similarly, without using $a'(p, A)$ the lower and upper bounds are much more important.

## References


[1] J.C. Becceneri, H.H. Yannasse, and N.Y. Soma. A method for solving the minimization of the maximum number of open stacks problem within a cutting process. *Computers & Operations Research*, 31:2315–2332, 2004.

[2] B.J. Yuen. Improved heuristics for sequencing cutting patterns. *European Journal of Operational Research*, 87:57–64, 1995.

[3] B.J. Yuen and K.V. Richardson. Establishing the optimality of sequencing heuristics for cutting stock problems. *European Journal of Operational Research*, 84:590–598, 1995.

# A Appendix

All experiments were run on a Pentium IV 3.4Ghz with 2GB RAM running Linux Fedora Core 3. The dynamic programming software was written in C, compiled with gcc 3.4.2 using -O3. The runs are performed using all $A^\star$ improvements, all preprocessing steps, all lower and upper bounds heurisitics (using the best value found), and the backwards stepwise search approach. In fact the backwards stepwise approach is fairly insensitive to upper and lower bounds unless the lower bound is the optimal which can save substantial computation.

Since the program is deterministic the search results are the same on each run of a benchmark. The timing results for each suite are aggregates over 10 runs of each individual benchmark in the suite. For the individuals the time shown is the average over ten runs of the individual benchmark. The time calculated is sum of user and system time given by `getrusage`, it accords well with wall clock times for these CPU intensive programs. For the problems that take significant time we observed around 10% variation in timings across different runs of the same benchmark.

The measure of search effort is the number of calls to stacks that do not immediately return, because of cache hit or $S = \emptyset$. This is the same as the number of non-symmetric calls to `labeling` if we consider this as a constraint programming approach with memoing. Note that since we use the backwards stepwise approach, between each successive stack number tried we empty the cache, so the total number of calls is just the sum of the calls made for each stack number. The maximum search effort per instance was set at $2^{25} = 33554432$ calls to stacks.

Note that we always run the dynamic programming search even if the calculated lower and upper bounds agree (in which case we know we have the optimal solution already)

The software found the opimal solutions for all problems except SP2, SP3 and SP4 which hit the search limit. It finds a solution of size 19 for SP2 using 25785 calls to stacks before hitting the limit trying 18 stacks. The runtime shown for SP2, SP3 and SP4 is the time to find the best solution. The best lower bounds we have for SP2, SP3, and SP4 calculated using lower bound heuristics and using stepwise are 18, 15 and 22 respectively.

| File | %solved | mean | Total runtime per instance (ms) | | | Search to find optimal | | | Total search effort | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | best | mean | med. | max | mean | med. | max | mean | med. | max |
| problem_10_10 | 100% | 8.03 | 0.09 | 0 | 2 | 7.01 | 7 | 23 | 8.21 | 7 | 86 |
| problem_10_20 | 100% | 8.92 | 0.13 | 0 | 4 | 10.45 | 10 | 46 | 15.66 | 11 | 419 |
| problem_15_15 | 100% | 12.87 | 0.37 | 0 | 7 | 13.75 | 13 | 89 | 39.19 | 14 | 584 |
| problem_15_30 | 100% | 14.02 | 2.49 | 1 | 43 | 25.17 | 22 | 487 | 260.09 | 23 | 6031 |
| problem_20_10 | 100% | 15.88 | 0.52 | 0 | 4 | 12.99 | 10 | 92 | 36.96 | 28 | 179 |
| problem_20_20 | 100% | 17.97 | 5.83 | 1 | 86 | 29.31 | 19 | 561 | 428.12 | 20 | 6634 |
| problem_30_10 | 100% | 23.95 | 1.63 | 1 | 9 | 17.59 | 10 | 178 | 57.62 | 52 | 280 |
| problem_30_15 | 100% | 25.97 | 7.48 | 3 | 46 | 35.45 | 15 | 384 | 282.63 | 113 | 1634 |
| problem_30_30 | 100% | 28.32 | 718.36 | 3 | 10074 | 999.80 | 30 | 64954 | 31473.85 | 30 | 379396 |
| problem_40_20 | 100% | 36.38 | 96.89 | 8 | 607 | 90.69 | 20 | 1329 | 2454.55 | 147 | 14757 |
| Shaw_20_20 | 100% | 13.68 | 12.43 | 11 | 42 | 45.80 | 19 | 474 | 812.76 | 667 | 3020 |
| wbo_10_10 | 100% | 5.92 | 0.14 | 0 | 1 | 9.82 | 10 | 11 | 14.00 | 10 | 60 |
| wbo_10_20 | 100% | 7.35 | 0.33 | 0 | 6 | 20.27 | 19 | 57 | 47.98 | 19 | 629 |
| wbo_10_30 | 100% | 8.20 | 1.27 | 0 | 21 | 26.32 | 27 | 30 | 147.53 | 28 | 1621 |
| wbo_15_15 | 100% | 9.35 | 1.11 | 1 | 7 | 15.57 | 15 | 31 | 103.60 | 71 | 579 |
| wbo_15_30 | 100% | 11.58 | 28.34 | 2 | 213 | 85.82 | 30 | 1936 | 2496.88 | 30 | 17724 |
| wbo_20_10 | 100% | 12.90 | 0.51 | 0 | 3 | 11.74 | 10 | 25 | 40.26 | 40 | 96 |
| wbo_20_20 | 100% | 13.69 | 8.72 | 7 | 42 | 38.48 | 20 | 338 | 540.33 | 363 | 2894 |
| wbo_30_10 | 100% | 20.05 | 1.78 | 2 | 5 | 14.52 | 10 | 70 | 60.79 | 58 | 117 |
| wbo_30_15 | 100% | 20.96 | 8.08 | 7 | 30 | 28.54 | 15 | 155 | 280.28 | 253 | 859 |
| wbo_30_30 | 100% | 22.56 | 1108.10 | 306 | 8686 | 608.63 | 30 | 14230 | 41707.21 | 16392 | 319162 |
| wbop_10_10 | 100% | 6.75 | 0.10 | 0 | 1 | 9.82 | 10 | 10 | 14.22 | 10 | 42 |
| wbop_10_20 | 100% | 8.07 | 0.56 | 0 | 8 | 21.02 | 19 | 105 | 69.78 | 20 | 715 |
| wbop_10_30 | 100% | 8.55 | 1.07 | 1 | 25 | 28.98 | 28 | 70 | 113.22 | 29 | 2464 |
| wbop_15_15 | 100% | 10.37 | 0.77 | 0 | 6 | 15.25 | 15 | 32 | 71.92 | 15 | 313 |
| wbop_15_30 | 100% | 12.15 | 18.63 | 2 | 197 | 162.52 | 30 | 2890 | 1593.28 | 30 | 18177 |
| wbop_20_10 | 100% | 14.28 | 0.49 | 0 | 3 | 11.82 | 10 | 28 | 32.30 | 25 | 85 |
| wbop_20_20 | 100% | 14.87 | 7.99 | 2 | 58 | 40.24 | 20 | 659 | 473.01 | 20 | 3428 |
| wbop_30_10 | 100% | 22.48 | 1.21 | 1 | 5 | 10.78 | 10 | 22 | 39.48 | 39 | 83 |
| wbop_30_15 | 100% | 22.38 | 7.50 | 5 | 38 | 25.13 | 15 | 166 | 249.48 | 156 | 1070 |
| wbop_30_30 | 100% | 23.84 | 986.12 | 87 | 8770 | 1113.31 | 30 | 35735 | 31250.81 | 2973 | 300677 |
| wbp_10_10 | 100% | 7.28 | 0.08 | 0 | 1 | 7.60 | 7 | 18 | 11.68 | 8 | 70 |
| wbp_10_20 | 100% | 8.71 | 0.17 | 0 | 2 | 11.93 | 12 | 36 | 25.07 | 13 | 330 |
| wbp_10_30 | 100% | 9.31 | 0.20 | 0 | 2 | 14.03 | 14 | 21 | 25.43 | 15 | 268 |
| wbp_15_15 | 100% | 11.05 | 0.56 | 0 | 6 | 13.82 | 13 | 54 | 59.75 | 15 | 509 |
| wbp_15_30 | 100% | 13.09 | 5.02 | 1 | 60 | 28.75 | 23 | 308 | 539.16 | 26 | 6580 |
| wbp_20_10 | 100% | 15.12 | 0.52 | 0 | 3 | 11.97 | 10 | 45 | 41.12 | 40 | 96 |
| wbp_20_20 | 100% | 15.41 | 7.26 | 2 | 83 | 50.58 | 19 | 1000 | 468.31 | 31 | 5136 |
| wbp_30_10 | 100% | 23.18 | 2.03 | 2 | 8 | 23.68 | 10 | 100 | 73.33 | 66 | 185 |
| wbp_30_15 | 100% | 22.98 | 10.56 | 7 | 45 | 57.48 | 15 | 651 | 377.62 | 270 | 1348 |
| wbp_30_30 | 100% | 24.46 | 1203.78 | 5 | 17097 | 2071.78 | 30 | 132556 | 45069.34 | 61 | 765944 |

Table 1: Aggregate results: Garcia de la Banda and Stuckey

| Instance | best value found | Proved optimal? | Runtime (ms) | Search to find optimal | Total search effort |
|---|---|---|---|---|---|
| Miller_20_40 | 13 | ✔ | 610 | 40 | 39656 |
| GP1 | 45 | ✔ | 8.4 | 42 | 42 |
| GP2 | 40 | ✔ | 11.2 | 48 | 48 |
| GP3 | 40 | ✔ | 12.6 | 50 | 50 |
| GP4 | 30 | ✔ | 10.5 | 37 | 37 |
| GP5 | 95 | ✔ | 84.8 | 208 | 208 |
| GP6 | 75 | ✔ | 138.0 | 100 | 100 |
| GP7 | 75 | ✔ | 118.8 | 99 | 99 |
| GP8 | 60 | ✔ | 174.3 | 96 | 96 |
| NWRS1 | 3 | ✔ | 0.2 | 8 | 8 |
| NWRS2 | 4 | ✔ | 0.1 | 11 | 11 |
| NWRS3 | 7 | ✔ | 0.0 | 13 | 13 |
| NWRS4 | 7 | ✔ | 0.1 | 15 | 15 |
| NWRS5 | 12 | ✔ | 1.4 | 20 | 20 |
| NWRS6 | 12 | ✔ | 1.0 | 23 | 23 |
| NWRS7 | 10 | ✔ | 3.0 | 32 | 32 |
| NWRS8 | 16 | ✔ | 2118.6 | 40 | 86869 |
| SP1 | 9 | ✔ | 26.4 | 17 | 1269 |
| SP2 | 19 | ✘ | 1650 (?) | 25785 (?) | — |
| SP3 | 36 | ✘ | 1 hour (?) | 949523 (?) | — |
| SP4 | 56 | ✘ | 4 hours (?) | 3447816 (?) | — |

Table 2: Individual results: Garcia de la Banda and Stuckey

## B Proofs

**Lemma 1** *If there exists $p \in S$ where $c(p) \subseteq o(S)$ then there is an optimal order for $stacks_P(S)$ beginning with $p$.*

**Proof:** Take any optimal order $\Pi_1 p' \Pi_2 p \Pi_3$ of $S$. Consider the order $p\Pi_1 p' \Pi_2 \Pi_3$. We show that the active stacks for each product can only decrease. Consider products in $\Pi_3$ have the same active sets since the set of products before and after is an unchanged. Now consider any product $p'$ (as a general representative of those products before $p$ in the original order). In the original order $ap' = c(P - \Pi_2 - \{p\} - \Pi_3) \cap c(\{p'\} \cup Pi_2 \cup \{p\} \cup \Pi_3)$. In the new order $ap'' = c(P - \Pi_2 - \Pi_3) \cap c(Pi_2 \cup \{p\} \cup \Pi_3)$. Now $c(P - \Pi_2 - \{p\} - \Pi_3) = c(P - \Pi_2 - \Pi_3)$ since $c(p) \subseteq c(P - S) \subseteq c(P - \Pi_2 - \{p\} - \Pi_3)$. and $c(Pi_2 \cup \{p\} \cup \Pi_3) \subseteq c(\{p'\} \cup Pi_2 \cup \{p\} \cup \Pi_3)$. Hence $ap'' \subseteq ap'$. We also have to examine the stacks for $p$. In the new order $a(p, S - \{p\}) \subseteq o(S)$ and $o(S)$ is a lower bound on the number of stacks in any order. Hence order $p\Pi_1 p' \Pi_2 \Pi_3$ has a minimal number of stacks. □

**Lemma 2** *The minimal number of stacks required for set of products $S$ is at least $b(S) = |o(S)| + \min\{d_{G'}(c) \mid c \in c(S), G' = (V, E - \{(c_1, c_2) | \{c_1, c_2\} \subseteq c(P - S)\})\}$.*

**Proof:** At the beginning of $S$ the remaining customers $c(S)$. We argue about the minimal number of stacks required to close any open customer $c \in c(S)$. In order to close a customer $c$ we need to have open stacks for the customer and all customers $c'$ adjacent in the customer graph (since $c$ and $c'$ share some product $p$ which needs to be completed before we can close $c$). The reduced customer graph $G'$ removes edges from $G$ which correspond to customer-customer dependencies which may already have been completed (since they may only occur in products in $P - S$). In order to close any customer $c$, we need to open at least $d_{G'}(c)$ new customers (since these edges only connect to unopened customers). Hence the bound holds. □

**Lemma 3** *If $Q \subseteq C$ is clique in the customer graph $G$, the minimal number of open stacks is at least $|Q|$*

**Proof:** Assume to the contrary. Eliminating all the customers except those in $Q$ (i.e., replacing $c(p)$ by $c(p) \cap Q$) gives a problem which clearly is a lower bound on the original problem. Imagine we have a schedule on this reduced problem with $|Q| - 1$ stacks. Then clearly there must be one customer $c_1$ which becomes inactive before another customer $c_2$ becomes active. Contradiction since then the product where $c_1$ and $c_2$ are jointly required cannot be scheduled. □

**Lemma 4** *If $d = 1 + \min\{d_G(c) \mid c \in C\}$ then $d$ is a lower bound on the open stacks for the problem.*

**Proof:** In order to close a stack we need to have active a customer, and all its neighbours in the customer graph. Since the customer shares at least one product with each of these. In order to close the first stack we need to have at least $d$ active customers. □

**Lemma 5** *Let $m$ be the minimal open stacks for a problem defined by $c(p)$, and $m'$ be the minimal open stacks for the problem defined by $c'(p)$ where $c'$ is a minor of $c$. Then $m' \leq m$.*

**Proof:** Take an optimal order $\Pi$ for $c$. Now since $(c_1, c_2) \in E$ there is a product that shares $c_1$ and $c_2$ and hence their open lifetimes intersect. Consider the same order $\Pi$ for $c'$. Since $c_2$ is replaced by $c_1$, the lifetime of $c_1$ is now exactly the union of the lifetimes of $c_1$ and $c_2$ for $c$. Hence the number of open stacks $m''$ given by $\Pi$ is such that $m'' \leq m$. The minimal number for $c'$ is $m' \leq m'' \leq m$. □

# Partition with Minimal Intersection

**Emmanuel Hebrard**
NICTA and UNSW
Sydney, Austalia
ehebrard@cse.unsw.edu.au

**Brahim Hnich**
University College Cork
Ireland
b.hnich@4c.ucc.ie

**Toby Walsh**
NICTA and UNSW
Sydney, Austalia
tw@cse.unsw.edu.au

## 1 Introduction

Our approach relies on interleaving a complete and a heuristic method. We therefore introduce two models: The first one is a complete model that channels the permutation of the products with a matrix of Boolean variables standing for open stack at a given time and for a given customer. We detail the propagation algorithm for the channelling constraint. The second model is an approximation approach, the idea is that given a product $p$ that we schedule at the middle rank ($m/2$) in the ordering, we can approximate the best *permutation* by the best *partition* of products around $p$. This relies on the observation that having many non-open stacks is harder to achieve at mid-schedule than at the start or the end. When there is a high enough demand, then the number of open stacks tends to increase and then decrease only once, the rank $m/2$ is therefore critical. The second observation is that once a product is chosen for the middle rank, the number of open stacks at the time this product is manufactured depends only on the partition of the other products *before* or *after*, and not on the actual permutation. Moreover, if we solve this problem for each product then the best number of stack closed at rank $m/2$ is a lower bound for the whole problem. We therefore solve one such problem for each product, and from this preprocessing we get a lower bound and also an approximate solution. We also found that, once each one of these partition problems is solved, giving this partition as starting point for the complete method is usually a good improvement. When the preprocessing is completed, then the upper and lower bounds are handed to the complete method (when they are not already equal) to hopefully prove optimality.

## 2 Complete Model

We refer to the matrix containing the data as $demand$. Then for any permutation of the columns (products) of $demand$, we can construct a matrix $openOrders$ that represents which orders are open for which products. For instance, let the following matrices represent $demand \Rightarrow openOrders$ for two possible permutations. The first permutation of the products has 4 orders opened at once for product 2, whilst the second permutation has never more than 2 open orders at the same time:

| | $P_1$ | $P_2$ | $P_3$ | | | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|---|---|---|---|---|
| $C_1$ | 0 | 1 | 0 | | $C_1$ | 0 | 1 | 0 |
| $C_2$ | 1 | 0 | 1 | $\Rightarrow$ | $C_2$ | 1 | 1 | 1 |
| $C_3$ | 1 | 0 | 1 | | $C_3$ | 1 | 1 | 1 |
| $C_4$ | 0 | 1 | 0 | | $C_4$ | 0 | 1 | 0 |
| | $P_1$ | $P_3$ | $P_2$ | | | $P_1$ | $P_3$ | $P_2$ |
| $C_1$ | 0 | 0 | 1 | | $C_1$ | 0 | 0 | 1 |
| $C_2$ | 1 | 1 | 0 | $\Rightarrow$ | $C_2$ | 1 | 1 | 0 |
| $C_3$ | 1 | 1 | 0 | | $C_3$ | 1 | 1 | 0 |
| $C_4$ | 0 | 0 | 1 | | $C_4$ | 0 | 0 | 1 |

Suppose that we have $n$ customers and $m$ products. We declare an array *permutation* of $m$ variables, ranging in $[1..m]$ with an `alldiff` constraint. The semantic of $permutation[i] = j$ is that product $i$ comes $j^{th}$ in the ordering. We also declare a matrix *openOrders* of $n \times m$ 0/1 variables. The semantic of these variables corresponds to the second and fourth matrices (*open orders*) in the example above. *openOrders*$[i, j] = 1$ iff there exists $k, l$ such that $permutation[k] \leq j \leq permutation[l]$ and customer $i$ demands products $k$ and $l$. Then we minimise the maximum sum on the rows of $openOrders$, that is $max_i(\sum_j openOrders[i, j])$.

The most important part of this model is the way $permutation$ and $openOrders$ are channelled. We devised a global constraint for that purpose.

**Definition 1** SEQUENCECHANNEL$([B_1, \ldots, B_n], X_1, \ldots, X_k) \Leftrightarrow$
$$\texttt{alldiff}(X_1, \ldots, X_k) \wedge$$
$$\forall 1 \leq i \leq k,\ B_{X_i} = 1 \wedge$$
$$\forall i \leq j \leq k,\ (B_i = 1 \wedge B_k = 1) \Rightarrow B_j = 1$$

We need one SEQUENCECHANNEL constraint for each customer. The Boolean variables $[B_1, \ldots, B_n]$ correspond to one row of $openOrders$, and the integer variables $X_1, \ldots, X_k$ to those variables in $permutation$ such that the corresponding product is demanded by the customer we consider. For instance, on the small example we have:

- SEQUENCECHANNEL$(openOrders[1], P_2)$
- SEQUENCECHANNEL$(openOrders[2], P_1, P_3)$
- SEQUENCECHANNEL$(openOrders[3], P_1, P_3)$
- SEQUENCECHANNEL$(openOrders[4], P_2)$

Now we give some rules to propagate SEQUENCECHANNEL$([B_1, \ldots, B_n], X_1, \ldots, X_k)$. Let

$\alpha_0$ (resp. $\beta_0$) be the smallest (resp. greatest) index of a Boolean variable that contains the value 1, and $\alpha_1$ (resp. $\beta_1$) be the smallest (resp. greatest) index of a Boolean variable that does not contain the value 0. The propagation rules that we present here first compute the values of $\alpha_0, \beta_0, \alpha_1, \beta_1$, then propagate to the integer variables and conversely, use the result to refine $\alpha_0, \beta_0, \alpha_1, \beta_1$ and prune the boolean variables:

**Propagation Resulting from $\alpha_0, \beta_0$:** The first observation is that if there exists $i$ such that $B_i = 0$ and $i - \alpha_0 < k$ then we have $\alpha_0 = i + 1$. This is because there is not enough room for all the 1's between $\alpha_0$ and $i$, and they must be all consecutive. Of course, the same reasoning can be applied to $\beta_0$.

Then we look at the $X_i$'s. We can prune the bounds of any $X_i$ to $[\alpha_0..\beta_0]$. When this is done we enforce:

$$\alpha_0 = min_i(min(P_i)) \;\&\; \beta_0 = max_i(max(P_i))$$

We have three cases:

1. $\beta_0 - \alpha_0 + 1 < k$: we fail.
2. $\beta_0 - \alpha_0 + 1 = k$: then we can set $\alpha_1$ and $\beta_1$ to $\alpha_0$ and $\beta_0$ respectively.
3. Otherwise: For all $i$ such that $X_i$ is ground, we enforce:

$$\alpha_1 = min(\alpha_1, i) \;\&\; \beta_1 = max(\beta_1, i)$$

Moreover, if the total number of values in the domains of the $X_i$'s is equal to $k$, then we have:

$$\alpha_1 = \alpha_0 \;\&\; \beta_1 = \beta_0$$

**Propagation Resulting from $\alpha_1, \beta_1$:** We have the following inequalities:

$$\alpha_1 \leq min_i(max(P_i)) \;\&\; \beta_1 \geq max_i(min(P_i))$$

Since some indexes lower than or equal to $min_i(max(P_i))$ (resp. greater than or equal to $max_i(min(P_i))$) will be set to one.

$$\alpha_1 \leq \beta_0 - k + 1 \;\&\; \beta_1 \geq \alpha_0 + k - 1$$

Since there will be at least $k$ 1's.

Finally we have $B_i = 0$ for all $1 \leq i \leq \alpha_0$ or $n \geq i \geq \beta_0$ and $B_i = 1$ for all $\alpha_1 \leq i \leq \beta_1$.

## 3 Heuristic model

The basic idea here is that the product that we choose to put in the middle of the ordering is the most constrained. It is easy to see that when the first product is manufactured, we need no more stacks than the demand for that product. This is also the case for the last product. Most of the time, the number of open stacks will follows a regular distribution with a unique peak (or plateau). Moreover, given a product that we choose to be manufactured in the "middle" of the sequence, only the *partition* of the other products before or after the rank $m/2$ is important to know how many stacks will be necessary at time $m/2$. To understand why this method gives good results, it is important to notice that for a stack to be closed at time $m/2$, it must be closed for either $0..m/2$ or $m/2..m$. Therefore, consider a product manufactured at time $m/2$, and a partition that ensure $k$ closed stacks at that time. This ensure at least $km/2$ closed stacks (0's in $openOrders$) for any permutation that respects this partition. The problem that we want to solve can be formulated as follows:

*Given a product $p$, what is the minimum number of open stacks at time $m/2$ if we schedule $p$ to rank $m/2$ ?*

The exact rank of other columns does not matter, this value depends only on how we partition the columns, either *before* or *after* $j$. Indeed, for any $i$, if we schedule product $j$ to rank $m/2$, then we have $openOrders[i][m/2] = 1$ iff either $demand[i][j] = 1$ or there exists $k, l$ such that $demand[i][k] = 1$ and $demand[i][l] = 1$ and the products $k$ and $l$ are scheduled on both sides of $m/2$, and the exact ranking of $k$ and $l$ are not important. We give an equivalent definition of the problem using set notations:

**Definition 2** MININTERSECTPARTITION*: given $m$ sets $s_1, \ldots s_m$, we must partition those sets into 2 groups $G_1 = \{s_{11}, \ldots s_{1m/2}\}$, $G_2 = \{s_{21}, \ldots s_{2m/2}\}$ such that $|G_1| = |G_2|$ and the size of the intersection between the union of all sets within each group is minimised:*

$$min(|\bigcup_{s_i \in G_1} \cap \bigcup_{s_j \in G_2}|)$$

Indeed, given a product $j$ that is scheduled to the rank $m/2$, we can represent each product $k$ as the set of customers $i$ that demand $k$ and does not demand $j$ ($s_k = \{i \mid demand[i,k] = 1 \wedge demand[i,j] = 0\}$). Now if two of those sets share a "customer" $i$ and are not put into the same group (before/after), then the stack for product $j$ will be open at rank $m/2$.

We give a constraint program to solve this problem:

1. A set of at most $n$ Boolean variables $Z_1, \ldots Z_d$, one for each initial 0 in the column $j$ of $demand$.
2. $m-1$ Boolean variables $P_1, \ldots P_{m-1}$, one for each column (product) apart form $j$. We pose $P_k = 0$ iff $P_k$ goes *before* the rank $i$ (given to the column $j$).
3. We post a sum constraint $\sum_{k \in [1..m-1]} P_k = m/2$, to enforce the partition.
4. For each columns $c_1, c_2$ and row $r$ such that $demand[r, c_1] = 1$, $demand[r, j] = 0$ and $demand[r, c_2] = 1$, we post a constraint to enforce that if these two columns are partitioned in different sides, then $Z_r$ should equal 1:

$$(P_{c_1} \oplus P_{c_2}) \Rightarrow Z_r$$

5. We minimise the sum $\sum_{k \in [1..d_j]} Z_k$, i.e., the number of "preserved" 0's (or intersections between columns).

This problem is in practice small (at most $n + m - 1$ Boolean variables) and easy to solve for instances up to a certain size. However, it is NP-hard, and for really large instances (say for instance 50 products and customers), solving this partition problem to optimality is difficult. Here is a sketch of a proof of NP-hardness:

**Proof:** We reduce MAX-2SAT to the problem of partitioning the collection of sets $\{S_1, \ldots, S_n\}$ into two collections $G_1, G_2$ of equal size such that $|\bigcup_{S_i \in G_1} \cap \bigcup_{S_j \in G_2}|$ is minimum.

Let $\phi$ be a 2SAT formula with $n$ atoms and $m$ binary clauses. We introduce two sets $S_i$ and $\bar{S}_i$ for every atom $x_i$. For every pair of literals $x_i, x_j$ there are 3 possibilities:

1. They are opposites ($x_j = \bar{x_i}$), then we do nothing.
2. There is a clause $c_k = (\bar{x_i}, \bar{x_j})$, then we introduce $n+1$ elements $e_{k1}, \ldots e_{k(n+1)}$ in both $S_i$ and $S_j$.
3. Otherwise, for the $k^{th}$ such pair, we introduce $n+2$ elements $f_{m+k1}, \ldots f_{m+k(n+2)}$ in both $S_i$ and $S_j$.

We first show that partitioning $S_i$ and $\bar{S}_i$ in the same collection is never optimal. Consider such a solution, since the two groups have equal size, there exists $j$ such that $S_j$ and $\bar{S}_j$ are in the second collection. Now if we swap $S_i$ and $S_j$, we will remove at least $2n(n+1)$ intersections, and add at most $2(n-1)(n+2)$. It thus always is an improvement.

As a consequence, any optimal solution corresponds to a valid 2SAT assignment, moreover, since allowed combinations are less penalised it corresponds to an optimal 2SAT solution.

### 3.1 Algorithm

In this section we detail how we combined these two approaches.

- We solve $m$ times the problem MININTERSECTPARTITION, once for each product.

  We collect the intersection size and keep it iff it is the lowest so far.

  Then we consider the partition, and we represent the problem using the complete model, but modified as follows: We set $permutation[j] = m/2$, and for each $i \neq j$, we add $permutation[i] < m/2$ or $permutation[i] > m/2$ according to the partition computed earlier.

  We collect the value returned by the complete method and keep it iff it is the lowest so far.
- Then we pass the bounds to the complete method, for completeness.

## 4 Implementation and Discussion

We implemented these models using `HalCSP`[1]. The heuristic model we actually used for most instances is a slight variation on the one introduced earlier. The only difference is that instead of choosing one product, we chose a pair of products to put at rank $m/2-1$ and $m/2$. The benefits both on lower and upper bounds usually compensated the fact that $O(n^2)$ problems were to be solved instead of $O(n)$. We usually imposed no cutoff when solving MININTERSECTPARTITION, but we imposed time cutoff for the complete method both for solving "partitioned" problems and the "total" one. The value of the cutoff varies according to the instances. It is worthwhile to note that MININTERSECTPARTITION being usually small and involving only Boolean variables, the number of backtracks can be huge for a given duration. On the other hand, the complete model makes much less backtracks for similar problem size and duration. For instance, on `wbo_30_30.txt` we will have 75,000 backtracks per second in one case and 900 in the other. Therefore the search effort column should be taken with great care, as the search effort corresponding to the complete method is underestimated in comparison with that of the heuristic model. All the experiments were run on desktop with an Intel Pentium 4 CPU 3.20GHz and 1Gb of memory. The "K", "M" and "G" in the tables correspond respectively to "Thousands", "Millions" and "Billions" of Backtracks.

[1] available at http://www.cse.unsw.edu.au/ ehebrard/codef.htm

## 5 Appendix

| File | solved | value | Total cpu time (s) | | | Best solution backtracks | | | Total backtracks | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | mean | mean | median | max | mean | median | max | mean | median | max |
| problem_10_10.dat | 100% | 8.03 | 0.6 | 0.0 | 41.7 | 1K | 118 | 38K | 7K | 564 | 511K |
| problem_10_20.dat | 64% | 8.74 | 32.7 | 6.0 | 606.4 | 1M | 22K | 38M | 1M | 116K | 38M |
| problem_15_15.dat | | | | | | | | | | | |
| problem_15_30.dat | | | | | | | | | | | |
| problem_20_10.dat | 99% | 15.88 | 1.1 | 0.2 | 32.1 | 2K | 1K | 104K | 7K | 2K | 128K |
| problem_20_20.dat | 60% | 17.98 | 76.0 | 33.7 | 662.0 | 3M | 40K | 49M | 3M | 203K | 49M |
| problem_30_10.dat | 100% | 23.99 | 1.5 | 0.4 | 38.1 | 3K | 1K | 101K | 7K | 3K | 132K |
| problem_30_15.dat | 53% | 25.98 | 17.1 | 7.9 | 128.1 | 100K | 4K | 3M | 140K | 57K | 3M |
| problem_30_30.dat | 20% | 28.47 | 365.4 | 289.6 | 4590.0 | 27M | 253K | 763M | 27M | 2M | 763M |
| problem_40_20.dat | 30% | 36.43 | 54.0 | 35.0 | 352.8 | 4M | 43K | 90M | 4M | 199K | 90M |
| ShawInstances.txt | 44% | 13.76 | 79.9 | 41.0 | 519.1 | 43K | 24K | 347K | 172K | 81K | 1M |
| wbo_10_10.txt | 100% | 5.92 | 0.1 | 0.1 | 0.2 | 484 | 340 | 1K | 983 | 926 | 2K |
| wbo_10_20.txt | 97% | 7.35 | 8.0 | 4.1 | 59.0 | 7K | 3K | 33K | 33K | 26K | 215K |
| wbo_10_30.txt | 77% | 8.20 | 138.3 | 126.1 | 723.2 | 144K | 64K | 1M | 327K | 246K | 1M |
| wbo_15_15.txt | 80% | 9.35 | 13.8 | 5.1 | 103.1 | 8K | 851 | 85K | 35K | 11K | 434K |
| wbo_15_30.txt | 11% | 11.65 | 169.5 | 92.6 | 828.4 | 180K | 25K | 1M | 482K | 488K | 1M |
| wbo_20_10.txt | 100% | 12.90 | 0.3 | 0.2 | 2.5 | 1K | 988 | 25K | 3K | 2K | 28K |
| wbo_20_20.txt | 65% | 13.74 | 35.3 | 24.7 | 307.1 | 25K | 10K | 449K | 91K | 78K | 449K |
| wbo_30_10.txt | 100% | 20.05 | 1.6 | 0.7 | 9.0 | 8K | 3K | 58K | 10K | 5K | 58K |
| wbo_30_15.txt | 59% | 20.99 | 37.1 | 31.4 | 238.8 | 14K | 11K | 144K | 52K | 24K | 489K |
| wbo_30_30.txt | 2% | 22.73 | 401.4 | 348.1 | 1135.4 | 625K | 110K | 16M | 3M | 1M | 20M |
| wbop_10_10.txt | 100% | 6.75 | 0.1 | 0.1 | 0.3 | 767 | 724 | 1K | 1K | 1K | 4K |
| wbop_10_20.txt | 95% | 8.07 | 10.7 | 3.8 | 118.2 | 9K | 8K | 28K | 41K | 26K | 389K |
| wbop_10_30.txt | 62% | 8.57 | 62.4 | 55.6 | 149.1 | 64K | 48K | 347K | 200K | 155K | 539K |
| wbop_15_15.txt | 90% | 10.37 | 11.7 | 1.1 | 229.4 | 7K | 1K | 124K | 35K | 4K | 871K |
| wbop_15_30.txt | 22% | 11.20 | 92.7 | 80.0 | 279.9 | 140K | 39K | 1M | 697K | 705K | 1M |
| wbop_20_10.txt | 100% | 14.47 | 0.1 | 0.1 | 0.4 | 1K | 1K | 3K | 1K | 1K | 3K |
| wbop_20_20.txt | 85% | 14.88 | 22.2 | 17.5 | 168.6 | 41K | 22K | 397K | 146K | 108K | 556K |
| wbop_30_10.txt | 100% | 22.50 | 0.5 | 0.4 | 1.6 | 4K | 4K | 10K | 5K | 5K | 10K |
| wbop_30_15.txt | 73% | 22.38 | 28.3 | 8.6 | 274.0 | 11K | 4K | 72K | 46K | 21K | 581K |
| wbop_30_30.txt | 20% | 23.94 | 791.6 | 551.4 | 4715.1 | 71M | 795K | 1G | 78M | 5M | 1G |
| wbp_10_10.txt | 100% | 7.28 | 1.6 | 0.1 | 14.3 | 1K | 668 | 46K | 18K | 1K | 162K |
| wbp_10_20.txt | 58% | 8.73 | 48.0 | 5.2 | 447.6 | 226K | 17K | 2M | 367K | 32K | 2M |
| wbp_10_30.txt | 62% | 9.31 | 172.1 | 37.4 | 1207.2 | 863K | 220K | 9M | 1M | 661K | 9M |
| wbp_15_15.txt | 40% | 11.08 | 38.9 | 34.6 | 221.3 | 13K | 2K | 91K | 200K | 44K | 1M |
| wbp_15_30.txt | 29% | 13.15 | 534.2 | 66.2 | 29668.6 | 43M | 66K | 4G | 43M | 505K | 4G |
| wbp_20_10.txt | 100% | 15.18 | 1.4 | 0.3 | 12.3 | 2K | 2K | 13K | 9K | 3K | 77K |
| wbp_20_20.txt | 47% | 15.50 | 69.9 | 35.9 | 466.4 | 110K | 30K | 738K | 239K | 109K | 1M |
| wbp_30_10.txt | 100% | 23.27 | 2.7 | 0.5 | 24.0 | 5K | 3K | 57K | 12K | 3K | 87K |
| wbp_30_15.txt | 20% | 23.13 | 32.5 | 36.2 | 56.2 | 6K | 2K | 37K | 21K | 15K | 61K |
| wbp_30_30.txt | | | | | | | | | | | |

Table 1: Aggregate results

| Instance | Value | Proved | Runtime | Best solution | Total |
|---|---|---|---|---|---|
| Miller19 | 13 | YES | 9043 | 139K | 13M |
| GP1 | 46 | NO | 640 | 47M | 78M |
| GP2 | 45 | NO | 47339 | 17K | 2G |
| GP3 | 44 | NO | 570 | 33M | 35M |
| GP4 | 41 | NO | 4243.5 | 89M | 1G |
| GP5 | | | | | |
| GP6 | | | | | |
| GP7 | | | | | |
| GP8 | | | | | |
| NWRS1 | 3 | YES | 65 | 340K | 340K |
| NWRS2 | 4 | YES | 15 | 86K | 87K |
| NWRS3 | 7 | NO | 365 | 27K | 166K |
| NWRS4 | 8 | NO | 344 | 25K | 119K |
| NWRS5 | 12 | NO | 305 | 14K | 53K |
| NWRS6 | 12 | YES | 20 | 14K | 82K |
| NWRS7 | 15 | NO | 5782 | 14K | 29M |
| NWRS8 | 17 | NO | 5656 | 14K | 60M |
| SP1 | 9 | NO | 212.4 | 13K | 429K |
| SP2 | 22 | NO | 84936.7 | 24M | 1G |
| SP3 | 51 | NO | 30192.8 | 600M | 600M |
| SP4 | | | | | |

Table 2: Individual results

# Improved lower bounds for solving the minimal open stacks problem

**Alice Miller,**
Department of Computing Science
University of Glasgow, Scotland
alice@dcs.gla.ac.uk

## Abstract

In this paper we demonstrate the benefit of calculating good lower bounds for the value of $M$, the maximum number of concurrently open stacks, for an optimal solution. We give several theoretical results and illustrate their use when applying a model checking approach to find solutions. The improvement afforded is equally valid when a constraints modelling approach is taken.

## 1 Introduction

When an optimal solution to a problem is sought, whichever modelling approach is used, failure to provide good initial bounds for a solution can make finding a solution more difficult and, in some cases impossible. In this paper we prove some theoretical results which provide some good lower bounds in some cases. Indeed, we could not have solved many of the final instances without these bounds. We also describe a construction technique, based on one of our theoretical results, which often provides us with the optimal solution without the need for search. When the constructed solution is not necessarily optimal (the maximal number of stacks is not equal to the best lower bound) it often provides a very good solution from which to start a search. We combine our theoretical results together with a model checking approach to find an optimal solution in almost all of the final instances. It is important to note, however, that when using a constraints modelling approach, the application of the theoretical bounds is equally valid and, indeed, essential (see [1]).

## 2 The model checking approach

Model checking has been used efficiently to solve a problem that is very similar to the minimal open stacks problem (MOSP), namely to solve the rehearsal problem [2].

We have adapted this model to investigate MOSP. Although we have not exploited the major advantages of model checking (concurrency, and communication for example), in the smaller examples it provides an efficient method for finding a solution with maximum number of open stacks less than or equal to $M$, for given $M$, if such a solution exists.

Explicit state model checking with SPIN involves converting a description of a system (written in the specification language Promela) into a finite graph, or state space, and performing a depth-first search over the state space. The nodes of the state space are the *states*, which are stored as tuples consisting of the current values of every variable in the system. Properties are checked by a depth-first search of the state space. If a state is reached which has been previously visited then the search will automatically backtrack. (This includes self loops.)

As well as checking for deadlock, livelock and *assertion violations*, SPIN allows us to verify properties expressed as linear temporal logic (LTL) formulas. If a path is found for which a given property is false, the search terminates and the current path provided as a counter-example.

The property that we use in this example, is a *safety property* (something bad will never happen). For a given $M$ we assert that no solution with maximum number of open stacks less than or equal to $M$ is possible. If there *is* a solution, we can examine the associated counter-example to construct the corresponding sequence of products. This sequence is constructed automatically using a Perl script, we do not provide details here.

One of the features of SPIN that we exploit is automatic backtracking, described above. If a subsequence of products has been generated in which the maximum number of open stacks is already greater than $M$ (and so the current subsequence can not possibly lead to a solution), a deliberate self-loop is introduced, causing the search to backtrack and the current subsequence to be abandoned. In addition, if the current path has so far successively placed products $p_0, p_1, \ldots, p_{n-1}$, then, because the associated states at this point would be identical, if any path had previously been explored for which the first $n$ products were also $p_0, p_1, \ldots, p_{n-1}$ (in any order), the search would again backtrack (the number of open stacks from this point onwards would be the same in either path).

The Promela specification for a simple model (the first $wbo\_10\_10$ example) is provided in Appendix 2. It is provided for the interested reader, and we are happy to provide additional explanation upon request.

Model checking is a very efficient tool for finding bugs in programs [3] but proving that no errors exist can be impossible due to memory and time constraints. This is because finding an error (or *solution* in our case) involves searching only part of the state space, and proving no errors requires

the entire search space to be searched. As such, in the stacks problem it is usually fairly easy to show that a solution for a given $M$ exists, (and to provide a solution) but - for the larger examples - difficult (and in some cases impossible) to show that no solution exists.

The basic approach involves initially setting $M$ to be the number of customers. We then produce a series of models, reducing the value of $M$ by 1 each time, until a model is produced for which no solution with maximum number of open stacks less than or equal to $M$ exists. We then increase $M$ by 1, recreate the associated model, and find a solution. All of these stages are performed automatically, using a template model and a file containing the particular example.

For some of the larger examples however, it is either very time-consuming, or impossible due to memory constraints, to prove the case where no solution exists using model checking alone. However, by precomputing a good lower bound for the optimal solution, it is often possible to avoid this last step altogether. If we have found a solution for which the maximum number of open stacks is equal to a known lower bound $b$, there is no need to investigate further.

In the following section, we give some theoretical results to enable us to establish a set of lower bounds for each example, from which we can choose the best.

## 3 Finding a good lower bound

The simplest lower bound for the number of concurrently open stacks is given by the maximum number of customers requesting a single product. In this section we give some results which help to improve on this lower bound.

First of all we introduce the idea of the degree of a given customer.

**Definition 1** *For any customer $i$, the degree of $i$, $deg(i)$ is the number of customers $j \neq i$ for which $i$ and $j$ select the same product. (We say that $i$ and $j$ are in a product.)*

For example, in instance Miller19, every customer has degree 10 because they all appear in a products with 10 other customers.

In the following, when we say that $M$ is a lower bound, we mean that there is no ordering of products such that the maximum number of concurrently open stacks is less than $M$.

**Theorem 1** *If*

$$M = min\{deg(i) : 0 < i < no\_of\_customers\}$$

*then $M + 1$ is a lower bound.*

**Proof** Suppose that $x$ is the first customer to have all of its orders satisfied. Then when the last order containing $x$ is filled, there are at least $M + 1$ stacks open. This is because $x$ has degree at least $M$, the orders containing $x$ must have involved opening $M$ stacks for all of the customers adjacent to $x$ plus a stack for $x$, none of which can have been closed as $x$ is the first customer to have had all of its orders filled.

**Corollary 1** *If we list the degrees of the customers in non-descending order, $d_0, d_1, ..., d_n$ say, where $n = no\_of\_customers - 1$ and $d_0 \leq d_1 \leq .... \leq d_n$, then a lower bound is given by $max\{d_i + 1 - i\}$.*

**Theorem 2** *Suppose $M$ is the minimum of the degrees of the customers. If there is only one customer, $x$ say, with $deg(x) = M$, then $M + 2$ is a lower bound.*

**Proof** If there is a solution with the maximum number of open stacks less than $M + 1$ then by theorem 1 it follows that $x$ is the first customer to have all of its products filled. Suppose that the first product in the sequence of products containing $x$ is $p_1$, and the last product in the sequence containing $x$ is $p_2$. Up to and including the point at which $p_2$ is made, no customer ($y$ say) not in a product with $x$ can have had a stack open. (Otherwise at $p_2$ there are at least $M + 1 + 1 = M + 2$ stacks open – every customer in a product with $x$, $x$ itself, and $y$). Similarly, if we reverse the order in which the products are made, since the maximum number of open stacks will not increase, $x$ must again be the first customer to have all of its products filled, and there can be no products containing a customer not in a product with $x$ up to the last product containing $x$ ($p_1$). This means, going back to the original order, if $y$ is some customer not in a product with $x$, $y$ can not appear in any product either up to and including $p_2$ or from $p_1$ to the final product inclusive. Thus $y$ can never appear. This is a contradiction.

**Corollary 2** *If there is only one customer, $x$ say, with minimum degree $M$, and all other customers have degree at least $M + 2$, then if there is some product $p$ containing 2 of the customers not in a product with $x$, then $M + 3$ is a lower bound.*

**Proof** As above, but this time we show that if there is a sequence with the maximum number of concurrently open stacks less than $M + 3$, then the product $p$ can not appear anywhere in the sequence.

**Theorem 3** *For any pair of customers $x$ and $y$, let $R_{x,y}$ be the size of the set of customers formed by taking the union of the neighbours of $x$ and the neighbours of $y$ and (if necessary) removing customers $x$ and $y$. Let $R_2$ be the minimum $R_{x,y}$ for all pairs of customers $x$ and $y$ (where $x \neq y$). Then $R_2 + 1$ is a lower bound.*

**Proof** Let customers $x$ and $y$ be the first two customers to have all of their products filled. If $x$ and $y$ have their last product filled at the same time, and the last product containing $x$ and $y$ is $p$ say, then when $p$ is placed, $R_{x,y} + 2$ customers have orders open. Suppose $x$ and $y$ have their last product filled at different times, and suppose $x$ has all of its products filled first. When the last product for $y$ is filled, there are $R_{x,y} + 1$ customers with orders open. If $R_2$ is the minimum such $R_{x,y}$, clearly $R_2 + 1$ is a lower bound.

**Corollary 3** *Let $R_3$ be the minimum $R_{x,y,z}$ for all triples $x, y, z$ where $x \neq y \neq z$, then $R_3 + 1$ is a lower bound. If we define $R_n$ in a similar way for all $2 \leq n \leq no\_of customers$ then $max\{R_n + 1 : 2 \leq n \leq no\_of\_customers\}$ is a lower bound.*

The following result gives an easy way to find a solution with maximum number of open stacks at most $N =$

$no\_customers - 1$, provided all customers do not have degree $N$.

**Theorem 4** *If at least one customer has degree less than $N$, there is a solution with maximum number of concurrently open stacks at most $N$.*

**Proof** Suppose that customer $x$ has degree$< N$ and $x$ requests $r$ products. Then for any sequence in which all of the $r$ products containing $x$ are placed first, the number of open stacks will never exceed $N$. This is because for the first $r$ products the number of stacks is at most $deg(x) + 1$, and for the remaining products the stack for $x$ is closed, and so there are at most $N$ open stacks.

**Theorem 5** *Let $a$ and $b$ be two products, such that the size of the union of the customers requiring product $a$ or $b$ ($C_a \cup C_b$) say) is $M$. If all customers with degree $< M - 1$ belong to both $C_a$ and $C_b$, then $M$ is a lower bound.*

**Proof** If the first customer to have all of its products filled has degree at least $M - 1$ then, by an argument similar to the above, $M$ is an upper bound. Assume w.l.o.g. that $a$ is made before $b$. Then, since all customers with degree $< M - 1$ require both products $a$ and $b$, they will not have their stacks closed until after both $a$ and $b$ have been made. So no open stacks will be closed until both $a$ and $b$ have been made. It follows that there will be at least $|C_a \cup C_b| = M$ stacks open when $b$ is made. In all cases, $M$ is an upper bound

**Theorem 6** *Let $a$ and $b$ be two products, such that $a$ is made before $b$, and the size of the union of the customers requiring product $a$ or $b$ ($C_a \cup C_b$) say) is $M$. Let $S_a$ and $S_b$ be the sets of customers requiring product $a$ but not $b$, and vice versa. If either:*

1. *for every customer $x \in S_a$, $x$ is in a product with every customer in $S_b$ or*
2. *All customers in $S_a$ apart from some customer $x$ satisfy (1) and $x$ is in a product with all but one of the customers in $S_b$, $S_b \setminus \{y\}$ say, $x$ is in a product with some $z$ in $C \setminus C_a \cup C_b$, and $z$ is in some product with $y$,*

*then $M$ is an upper bound. If there is some customer $x$ for which the second case holds, but no suitable $y$ exists, $M - 1$ is a lower bound.*

**Proof** Omitted for space reasons.

# 4 A Construction

The following construction is loosely based on observation and Theorem 6. It often provides an optimal solution immediately (instances GP3 and GP4, for example), and in all cases provides a good starting point. In some instances (GP7 for example) it provided the best solution, where model checking alone could not cope with such a large problem.

**Construction 1** Let $M$ be the minimum size of the pairwise unions (see Theorems 5 and 6). Pick two customers $a$ and $b$ such that the size of the union $C_a \cup C_b$ is equal to $M$, and $deg(a) \leq M$. List the products ordered by $a$ first, followed by any (remaining) products on $b$, followed by any products that are ordered only by customers belonging to $C_a \cup C_b$. It can easily be shown that the number of open stacks at any point so far is at most $M + 1$. Henceforth, when we have placed a product $P$, we say that a customer is *finished* if it does not appear in any unplaced products (*unfinished* otherwise), and *spare* if it has not yet appeared in a placed product. For any customer $i$, let $s(i)$ equal the number of spare points adjacent to $i$, then we define

$$sum_i = \begin{cases} s_i & \text{if } i \text{ is not a spare point} \\ s_i + 1 & \text{otherwise} \end{cases}$$

The construction proceeds as follows: If the number of unfinished customers is at most the maximum stack size so far, stop. Otherwise, pick a customer $i$ such that $sum_i$ is minimal. Add all remaining products ordered by $i$ followed by all products which are ordered only by customers which are not (now) spare. Repeat as often as necessary.

## 4.1 Results

Our experiments were performed on a PC with a 2.4GHz Intel Xenon processor, 3Gb of available main memory, running Linux (2.4.18), with SPIN version 4.2.3.

In Table 1 below, we give results of the preprocess to find the greatest lower bound and the first theorem to find this bound, for the single detailed examples considered in Table 4 of Appendix 1. $GLB$ is the greatest lower bound. Bound 0 is the maximum number of 1s in a column, and bounds 1 to 9 correspond to Theorem 1, Corollary 1, Theorem 3, Corollary 3 (three-way unions), Theorem 2, Corollary 2, Corollary 3 (four-way unions), Theorem 5 and Theorem 6. We decided to not implement bound 7, as it proved to be too time consuming. In each case an example pair is provided to construct a good initial solution using Construction 1, if such a pair exists. The customer with the smaller degree is given first in each case.

Our results for the grouped data sets are given in Tables 2 and 3 of Appendix 1. Note that we only record the percentage solved, the mean value and the average time taken to solve completely. The time is in seconds, and includes the time taken to compute the best lower bound in each case. In order to achieve better, more accurate results, the theoretical results achieved in this paper have been applied to a Constraint programming approach in [1], where results have been recorded more accurately. We initially limited our search time to 1 hour per data set. However, for some of the data sets, due to time constraints, where we were unlikely to achieve an optimal solution within our time limit, we have not tried to find a solution at all. In cases where we believe we could have found optimal solutions to all instances, but did not (because we would have missed the submission deadline!) we have marked all fields with a 'T'.

In table 4 we give our results for the single, larger instances. In many cases we have used the construction approach to find the best solution. Note that in each case, if the constructed solution is optimal, the time, number of fails etc. both up to and including the solution is zero (no search is required). In cases where the constructed solution is one more

| File | Lower bound | Bound used | Example pair |
|---|---|---|---|
| Miller19 | 13 | bound 4 | $(0, 2)$ |
| GP1 | 45 | bound 3 | $(6, 17)$ |
| GP2 | 40 | bound 9 | $(34, 2)$ |
| GP3 | 40 | bound 1 | $(18, 20)$ |
| GP4 | 30 | bound 0 | $(31, 22)$ |
| GP5 | 95 | bound 3 | $(20, 50)$ |
| GP6 | 75 | bound 1 | $(49, 0)$ |
| GP7 | 75 | bound 3 | $(62, 28)$ |
| GP8 | 60 | bound 4 | $(15, 31)$ |
| NWRS1 | 3 | bound 0 | $(3, 0)$ |
| NWRS2 | 4 | bound 0 | $(4, 0)$ |
| NWRS3 | 7 | bound 9 | |
| NWRS4 | 7 | bound 1 | $(2, 1)$ |
| NWRS5 | 11 | bound 9 | $(9, 19)$ |
| NWRS6 | 12 | bound 9 | $(8, 9)$ |
| NWRS7 | 10 | bound 9 | $(13, 6)$ |
| NWRS8 | 12 | bound 3 | $(24, 2)$ |
| SP1 | 8 | bound 0 | |
| SP2 | 9 | bound 0 | $(29, 1)$ |
| SP3 | 12 | bound 4 | $(35, 63)$ |
| SP4 | 13 | bound 0 | $(22, 65)$ |

Table 1: Best lower bounds for the larger instances

than the greatest lower bound, the time etc. up to the best solution is $0$, but the time etc. to prove the optimum solution is not. The number of fails recorded is the number of matched states in each case (when the search is forced to backtrack).

## References


[1] A. Miller, C. Unsworth and P. Prosser. A constraint model and a reduction operator for the minimising open stacks problem. In Ian Gent and Barbara Smith, editors, *Proceedings of the 4th Workshop on modelling and solving problems with constraints. Held in conjunction with IJCAI'05*. Edinburgh, July 2005.

[2] P. Gregory, A. Miller, and P. Prosser. Solving the rehearsal problem with planning and with model checking. In Brahim Hnich and Toby Walsh, editors, *Proceedings of the 3rd Workshop on modelling and solving problems with constraints. Held in conjunction with the 16th European Conference on Artificial Intelligence (ECAI 2004).*, pages 157–171, Valencia, Spain, August 2004.

[3] Gerard J. Holzmann. The logic of bugs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'02*, pages 81–87, Charleston, South Carolina, USA, November 2002. ACM Press.

## Appendix 1

| File | % solved | mean value | mean time |
|---|---|---|---|
| problem_10_10.dat | 100 | 8.04 | 2.05 |
| problem_10_20.dat | 100 | 8.93 | 2.72 |
| problem_15_15.dat | 100 | 12.88 | 2.83 |
| problem_15_30.dat | - | - | - |
| problem_20_10.dat | T | T | T |
| problem_20_20.dat | T | T | T |
| problem_30_10.dat | - | - | - |
| problem_30_15.dat | T | T | T |
| problem_30_30.dat | - | - | - |
| problem_40_30.dat | - | - | - |
| ShawInstances.txt | 100 | 13.72 | 10.56 |
| wbo_10_10 | 100 | 6.03 | 3.43 |
| wbo_10_20 | 100 | 7.4 | 3.05 |
| wbo_10_30 | 100 | 8.23 | 72.38 |
| wbo_15_15 | 100 | 9.35 | 5.35 |
| wbo_15_30 | - | - | - |
| wbo_20_10 | 100 | 12.9 | 6.5 |
| wbo_20_20 | 100 | 13.7 | 9.02 |
| wbo_30_10 | 100 | 20.06 | 9.41 |
| wbo_30_15 | 100 | 20.98 | 9.7 |
| wbo_30_30 | - | - | - |

Table 2: Times to find the optimal solution for each data set

| File | % solved | mean value | mean time |
|---|---|---|---|
| wbop_10_10 | 100 | 6.78 | 1.4 |
| wbop_10_20 | 100 | 8.1 | 2.5 |
| wbop_10_30 | - | - | - |
| wbop_15_15 | 100 | 10.4 | 4.35 |
| wbop_15_30 | - | - | - |
| wbop_20_10 | 100 | 14.3 | 5.2 |
| wbop_20_20 | 100 | 14.88 | 7.1 |
| wbop_30_10 | 100 | 22.5 | 7.25 |
| wbop_30_15 | 100 | 22.38 | 8.53 |
| wbop_30_30 | - | - | - |
| wbp_10_10 | 100 | 7.3 | 2.53 |
| wbp_10_20 | 100 | 8.73 | 3.1 |
| wbp_10_30 | - | - | - |
| wbp_15_15 | 100 | 11.05 | 4.0 |
| wbp_15_30 | - | - | - |
| wbp_20_10 | 100 | 15.13 | 4.65 |
| wbp_20_20 | 100 | 15.40 | 10.56 |
| wbp_30_10 | 100 | 23.18 | 6.83 |
| wbp_30_15 | 100 | 23.0 | 8.32 |
| wbp_30_30 | - | - | - |

Table 3: Times to find the best solution for each data set

| File | solved | best value | time to best solution | fails to best solution | time to prove | fails to prove |
|---|---|---|---|---|---|---|
| Miller19 | Yes | 13 | 0 | 0 | 0 | 0 |
| GP1 | Yes | 45 | 0 | 0 | 0 | 0 |
| GP2 | Yes | 40 | 0 | 0 | 1.79 | 29888 |
| GP3 | Yes | 40 | 17.34 | 668 | 0.18 | 1675 |
| GP4 | Yes | 30 | 0 | 0 | 0 | 0 |
| GP5 | Yes | 95 | 0 | 0 | 0 | 0 |
| GP6 | No | 76 | 0 | 0 | - | - |
| GP7 | No | 76 | 0 | 0 | - | - |
| GP8 | No | 61 | 0 | 0 | - | - |
| NWRS1 | Yes | 3 | 0 | 0 | 0 | 0 |
| NWRS2 | Yes | 4 | 0 | 0 | 0.01 | 47 |
| NWRS3 | Yes | 7 | 0.58 | 27616 | 0.88 | 182412 |
| NWRS4 | Yes | 7 | 0 | 0 | 0 | 0 |
| NWRS5 | Yes | 12 | 1.093 | 970 | 23.82 | 79984 |
| NWRS6 | Yes | 12 | 0 | 0 | - | - |
| NWRS7 | Yes | 10 | 0 | 0 | 0 | 0 |
| NWRS8 | No | 16 | 0 | 0 | - | - |
| SP1 | Yes | 9 | 0.375 | 282 | 277.79 | 1135110 |
| SP2 | No | 22 | 0 | 0 | - | - |
| SP3 | No | 35 | 0 | 0 | - | - |
| SP4 | No | 54 | 0 | 0 | - | - |

Table 4: Times and number of fails to prove the best solution for each problem

## Appendix 2

The Promela specification for a simple model (the first $wbo\_10\_10$ example) is provided below. Note that we use a set of three dots ..., to indicate where code has been omitted, for space reasons.

The main process is the scheduler process, and the product order is decided non-deterministically. Note that $MAX$ is set to $8$ here. This model is used to investigate the existence of a solution with maximum stack size at most $8$.

```
#define MAX 8
#define no_prods 10
/* prods labelled 0 to 9 */
#define no_custs 10
/* custs labelled 0 to 9 */

byte no_needed=9; bit STOP=0;
bit prod_made[no_prods]=0;
/*set to 0 if prod still not made*/

byte stacks=0; byte no_made=0;
/*no of prods currently made*/

byte orders_left[no_custs]=0;
/*no of prods left for each cust*/
/*set in init*/
bit order_started[no_custs]=0;
/*has custs order started*/
typedef array {bit prod[no_prods]};
hidden array orders[no_custs]=0;

inline choose_next_prod(choice)
{do
 ::atomic{prod_made[0]==0->
                  choice=0;break}
 ::atomic{prod_made[1]==0->
                  choice=1;break}
 ...

 ::atomic{prod_made[8]==0->
                  choice=8;break}
 ::atomic{((prod_made[9]==0)
    &&(prod_made[0]==1))->
                  choice=9;break}
    od}


inline make_prod(j)
{byte count1=0;
 stacks=0;
 do
 ::atomic{(count1==no_custs)->
    prod_made[j]=1;
    count1=0;break}
 ::atomic{else->
  if
  ::((order_started[count1]==1)
      &&(orders_left[count1]>0))->
     stacks++;
     if
     :: (orders[count1].prod[j]==1)->
                   orders_left[count1]--
     :: (orders[count1].prod[j]==0)->skip
     fi;
  ::((order_started[count1]==0)
      &&(orders[count1].prod[j]==1))->
     stacks++;
     order_started[count1]++;
     orders_left[count1]--
```

```
  ::else->skip
  fi;
  count1++}
 od}

proctype scheduler()
{byte count=no_prods;
 byte mymax=0; stacks=0;
 start:
 do
 ::atomic{(stacks>MAX)->skip}
 ::atomic{((stacks<=MAX)
    &&(no_made==no_needed))->break}
 ::atomic{else->choose_next_prod(count);
    make_prod(count); no_made++;
    printf("\n make prod %d\n",count);
    if
    ::(stacks>mymax)->mymax=stacks
    ::else->skip
    fi}
 od;
 printf("\n Max stacks %d now\n",mymax);
 STOP=1
 }

init{atomic{

orders[0].prod[3]=1;
orders[0].prod[7]=1;
orders_left[0]=2;

orders[1].prod[1]=1;
orders[1].prod[4]=1;
orders[1].prod[5]=1;
orders_left[1]=3;
...

orders[9].prod[2]=1;
orders_left[9]=1;

prod_made[0]=1; /* don't need this prod */
run scheduler()
}
}

#define p (STOP==1)

#include "rehearsal.ltl"
```

# A Constraint Model and a Reduction Operator for the Minimising Open Stacks Problem

**Alice Miller and Patrick Prosser and Chris Unsworth**
Department of Computing Science
University of Glasgow, Scotland
{alice/pat/chrisu}@dcs.gla.ac.uk

## Abstract

We present two constraint models for the Minimising Open Stacks Problem (MOSP). Our first model is based on that reported in [1], and our second is a refinement and is more space efficient. We also present two reduction operators for the MOSP. One reduction operator is applied as a pre-process to remove elements of the problem that are provably redundant, the second dynamically reduces the problem during search. We also introduce conditional lower bounds, which are lower bounds associated with a partial assignment. Experiments are then performed using the two reduction operators, and our best constraint model using the lower bounds reported in [2] and the conditional lower bounds.

## 1 Introduction

The minimising open stacks problem (MOSP) is in principal very similar to the rehearsal problem, as described in [3]. The MOSP is essentially a permutation problem. We are given $m$ products and $n$ customers. A customer may demand a number of different products. Therefore we can think of a customer as a 0/1 vector (or row) of length $m$ and a product as a 0/1 column, where the column has $n$ elements corresponding to customers. When the first product for a customer is produced, a stack (or a pallet) is opened for that customer, and that stack is closed when we have made the last product for that customer. The products can be made in any order, i.e. we can permute the columns in $m!$ ways. By permuting the columns we can then control when customer orders are open and closed. The goal is then to find a permutation that minimises the number of stacks/pallets open at any time.

Below we present two constraint encodings for the MOSP. The first encoding is based on that in [1] and was implemented in JChoco. Our second encoding is more compact, using less variables and less constraints, and allows us to model larger problems. This model was then encoded in ILOG's JSolver.

We also present a reduction operator. This is a pre-processing step in problem solving, i.e. the problem is processed to produce a smaller representative problem that can then be modelled and solved. The solution to this problem can then be inflated, in linear time, to give a solution to the original problem.

## 2 A Constraint Programming Encodings

Below we introduce the variables and the constraints (in *italics*). The main aspects of the model are to *1-fill* rows of a 0/1 array, such that for a given row, say i, we locate the position of the first 1 in that row and also the last 1 in that row, and then fill the intermediate elements with 1's. This corresponds to a customer order being open from the first product demanded up to and including the last product demanded.

- M a two dimensional array. M[i][j] = 1 if and only if customer i requires product j. The array M is essentially an array of constants, i.e. M is the data read in initially and does not change.
- S a one dimensional array. If S[j] = k then product j will be produced in time slot (column) k. That is, S gives us the permutation of the columns. Each variable in S has a domain 1 to $m$, and all the variables in S must take different values.
- P a one dimensional array. If P[k] = j then in time slot (column) k, product j will be produced. In order to force S and P to maintain a permutation we use the channelling constraints $S[j] = k \leftrightarrow P[k] = j$.
- T is the timetable, and is a two dimensional array of 0/1 variables. $S[j] = k \rightarrow T[i][k] = M[i][j]$. That is, if product j is made in time slot k (i.e. S[j] = k) and customer i demands product j (i.e. M[i][j]) then product j is made for customer i in time k (i.e. T[i][k] = M[i][j]).
- open is a two dimensional array of 0/1 variables. If open[i][k] = 1 then something is made for customer i in time k or earlier. Consequently, a stack is open for that customer at time k and if a stack is open for customer i at time k the stack is also open at time k+1. Therefore we have a right-rippling constraint such that $open[i][k] = 1 \rightarrow open[i][k+1] = 1$. This right-ripple is initiated when T[i][k] = 1. We then have the following constraints, $T[i][k] = 1 \rightarrow open[i][k] = 1$.
- nc is also a two dimensional array of 0/1 variables. This array is symmetrical to the array above, stating when a stack is not closed, and has a left-rippling constraint. If nc[i][k] = 1 then something is made for customer i

in time k or earlier, consequently the stack for this customer cannot be closed at time k, and neither can it be closed at time k-1. Therefore we have the left-rippling constraint $nc[i][k] = 1 \rightarrow nc[i][k-1] = 1$. Again, the left-ripple is kicked off when T[i][k] = 1. The constraint is then $T[i][k] = 1 \rightarrow nc[i][k] = 1$.

Stacked is a two dimensional array, such that Stacked[i][k] is 1 if and only if the ith customer's order is stacked at time k and the stack has not been closed at time k. Therefore we have the constraint $Stacked[i][k] = 1 \leftrightarrow open[i][k] = 1 \wedge nc[i][k] = 1$.

soat is a one dimensional array of $m$ variables with domains 0 to $n$, such that soat[k] is the number of stacks open at time k. Therefore soat[k] is the sum of the variables in the kth column of array Stacked. Therefore we have the constraint $soat[k] = \sum_{i=0}^{i=n-1} Stacked[i][k]$.

cost is the objective variable to be minimised, and is the maximum of the values in the vector soat (stacks open at time). That is, we want to minimise the maximum number of stacks open at any time, consequently we have the constraint $maximum(cost, \{soat[k] \ : \ 0 \leq k < m\})$. The maximum constraint works as follows. If the lower bound of some variable soat[k] increases then $lwb(cost) = max(lwb(cost), lwb(soat[k]))$. If $lwb(cost)$ increases then there is no effect. If the upper bound of some variable soat[k] decreases then we find the variable soat[l] that has the largest upper bound and set $upb(cost) = min(upb(cost), upb(soat[l]))$. If $upb(cost)$ decreases then for all values of k ($0 \leq k < m$) $upb(soat[k]) = min(upb(soat[k]), upb(cost))$.

The $maximum$ constraint can also be realised using primitive constraints. Consider the case where we have 3 variables A, B, C and we constrain variable X to be the maximum of A, B, and C. This can be done as follows: $X \geq A \wedge X \geq B \wedge X \geq C \wedge (X = A \vee X = B \vee X = C)$.

The above model was coded in JChoco using the S variables as the decisions. Variables were instantiated in the static order S[0], S[1], S[2], ... , S[m-1]. A symmetry breaking constraint was applied, similar to that in [1], such that S[0] < S[m-1].

A second model was also coded up in JSolver. The P variables were used as decisions, and the arrays open, nc and T were not used (and consequently the ripple constraints in those arrays were not used). To replace the 2-dimensional arrays T, open and nc, we introduced the following one dimensional arrays.

start a one dimensional integer array. If start[i] = k then the first product for customer i is made in time slot k. This is maintained by the constraint $minimum(start[i], \{S[j] \ : \ 0 \leq j < m \wedge M[i][j] = 1\})$. The $minimum$ constraint can be realised either as a n-ary constraint, similar to $maximise$ above, or by using primitives (also as above).

end a one dimensional integer array. If end[i] = k then the last product for customer i is made in time slot k. This is maintained by the constraint $maximum(end[i], \{S[j] \ : \ 0 \leq j < m \wedge M[i][j] = 1\})$.

The variables in the Stacked array are then maintained by the constraint, $Stacked[i][k] = 1 \leftrightarrow start[i] \leq k \leq end[i]$. This (JSolver) model is significantly more compact than the one first described (coded in JChoco) using less variables and less constraints. This allows us to model larger problems.

## 3 A Reduction Operator

We now describe a reduction operator, and present a proof that this reduction is sound. Essentially this operator is a preprocess step where we input the data for the problem and remove from it products that have no effect on the optimal solution. This delivers a new representative problem that has less columns/products than the original. This reduced problem can then be solved to optimality and an optimal solution to the original problem constructed in linear time.

### 3.1 Removing Subsumed Columns

Our reduction operator removes a product $p_i$ from the problem if there exists some other product $p_j$ such that all the customers that demand product $p_i$ also demand product $p_j$, i.e. we say that $p_i$ is a subset of $p_j$. A practical example of this might be two products, the first a dustbin and the second a dustbin lid. No customer demands a dustbin lid if they have not already demanded a dustbin. We can ignore the production of the dustbin lid in constructing our schedule. We can then re-insert the production of the dustbin lid into the schedule immediately after the production of the dustbin without altering the cost of that schedule.

**Theorem 1** *Let $P_n$ be a problem (instance) consisting of $n$ products $p_1, p_2, \ldots, p_n$. If, for some $i$ and $j$, $p_i \subseteq p_j$, $P^i_{n-1}$ is the problem obtained from $P_n$ by removing $p_i$ and $s^i_{n-1}$ an optimal sequence for $P^i_{n-1}$, then $s^i_{n-1}(i)$, the sequence formed by inserting $p_i$ after $p_j$ in $s^i_{n-1}$, is an optimal sequence for $P_n$.*

To prove the theorem, we use two lemmas:

**Lemma 1** *If $s_{n-1}$ is a sequence corresponding to any problem with $n-1$ products, and $s_n$ a sequence formed by adding a new product $p_i$, then if $R$ and $R'$ are the maximum number of open stacks for $s_{n-1}$ and $s_n$ respectively, $R' \geq R$.*

**Proof** Suppose that the maximum number of stacks ($R$) for $s_{n-1}$ is achieved when product $p_k$ is placed. All of the customers that have stacks open at that point have ordered products before (or at) $p_k$. Regardless of where $p_i$ is placed, this is still true, and so the number of stacks open when $p_k$ is placed is still at least $R$.

**Lemma 2** *If $s_{n-1}$ is a sequence corresponding to any problem with $n-1$ products, and $s_n$ a sequence formed by adding a new product $p_i$, which is a subset of some existing product $p_j$, directly after $p_j$, then if $R$ and $R'$ are the maximum open stacks for $s_{n-1}$ and $s_n$ respectively, $R' = R$.*

**Proof** We show that when $p_i$ is inserted after $p_j$, the number of stacks open when any product $p_k$ is placed is unchanged.

We also show that the number of open stacks when $p_i$ is placed will be less than or equal to that of $p_j$.

The number of open stacks open when $p_k$ is placed equals the number of orders opened on or before $p_k$ minus the number or orders closed before $p_k$. If $p_k$ is positioned before $p_i$ then clearly these totals will be unaffected, thus the number of open stacks will remain the same.

If $p_k$ is positioned after $p_i$ then the number of open orders will remain the same as all orders that require $p_i$ will already have been opened by $p_j$. The number of closed orders will also remain the same as any orders that where previously closed by $p_j$ that require $p_i$ will now be closed by $p_i$. Therefore the number of open stacks will remain the same.

Because $p_i$ will not open any new orders, the number of open stacks when $p_i$ is placed will be the same as $p_j$ minus the number of orders closed at $p_j$.

**Proof of Theorem 1** Suppose that $s^i_{n-1}$ is the optimal solution of $P^i_{n-1}$ and that $R$ is the maximum number of open stacks associated with $s^i_{n-1}$. By Lemma 2, the maximum number of open stack for $s^i_{n-1}(i)$, with $p_i$ placed directly after $p_j$ is $R$.

If there is no solution to $P_n$ with maximum number of open stacks less than $R$ then we are done. Suppose then, that there is some solution, $s_n$ with maximum number of open stacks equal to $R'$ say, where $R' \leq R$. Removing $p_i$ from $s_{n-1}$ will result in a sequence for $P^i_{n-1}$ which must have maximum number of open stacks $R'' \geq R$ since we know that $R$ is the optimal value for $P^i_{n-1}$. But by replacing $p_i$ we must obtain a sequence with maximum number of open stacks $\geq R''$, by Lemma 1. Thus $R' \geq R''$. So we must have $R' = R''$, which is a contradiction as $R'' \geq R$.

### 3.2 Dynamic Reduction

A variant of the reduction operator is used inside the search process. This operator forces a product to be produced next, based on the state of the partial solution. Assume the search process is free to select the next product to produce, and we have computed the current number of open stacks. If there is an unselected product such that if selected next the number of open stacks does not increase, then that decision is forced. Furthermore, this selection will not increase the number of open stacks in any subsequent time slots.

### 3.3 Lower Bounds

A set of lower bounds is calculated for each problem, during a pre-processing stage. The first, $bound_0$, is simply the maximum number of 1s contained in any given column of the input data, i.e. the maximum demand for a product. The other bounds, together with the proof of their validity, are described in full in [2]. We summarise them below. Note that, for any customer $i$, $deg(i)$ denotes the *degree* of $i$, defined as the number of customers that order at least one of the same products as $i$ (i.e. the number of *neighbours* of $i$). Also, if $n$ is the number of customers, $d_0, d_1, \ldots, d_{n-1}$ is a list of the degrees of the customers in non-decreasing order. For any pair of customers $i$ and $j$, $i \neq j$, we define $R_{i,j}$ to be the set of neighbours of $i$ and $j$, not including $i$ or $j$. Similarly we define $R_{i,j,k}$ for any set of (distinct) customers $i$, $j$ and $k$.

$$
\begin{aligned}
bound_1 &= d_0 + 1 \\
bound_2 &= \max\{d_i + 1 - i : 0 \leq i < n\} \\
bound_3 &= \begin{cases} d_0 + 2 & \text{if } d_1 - d_0 > 0 \\ 0 & \text{otherwise} \end{cases} \\
bound_4 &= \max\{|R_{x,y}| + 1 : x \neq y\} \\
bound_5 &= \max\{|R_{x,y,z}| + 1 : x \neq y \neq z\}
\end{aligned}
$$

From this set of lower bounds we choose the maximum, $G$. If we find a solution with cost $G$, we know that this must be an optimal solution.

### 3.4 Conditional Lower Bounds

An additional set of lower bounds are calculated which take into account a partial assignment. We define $D_a$ to be the set of customers that require product $a$. We define $R_i$ to be the set of neighbours of $i$. $bound_a$ is the lower bound for a partial solution where a product $a$ has been assigned to the first time slot. $bound^{a,b}$ is the lower bound when products $a$ and $b$ have been assigned to the first two time slots.

$$
\begin{aligned}
bound^a_1 &= \max\{|R_x \cup D_a| + 1 : x\} \\
bound^a_2 &= \max\{|R_{x,y} \cup D_a| + 1 : x \neq y\} \\
bound^a_3 &= \max\{|R_{x,y,z} \cup D_a| + 1 : x \neq y \neq z\} \\
bound^a &= \max\{bound^a_1, bound^a_2, bound^a_3\} \\
bound^{a,b}_1 &= \max\{|R_x \cup D_a \cup D_b| + 1 : x\} \\
bound^{a,b}_2 &= \max\{|R_{x,y} \cup D_a \cup D_b| + 1 : x \neq y\} \\
bound^{a,b}_3 &= \max\{|R_{x,y,z} \cup D_a \cup D_b| + 1 : x \neq y \neq z\} \\
bound^{a,b} &= \max\{bound^{a,b}_1, bound^{a,b}_2, bound^{a,b}_3\}
\end{aligned}
$$

These bounds are used by adding the following constraints.

$$
\begin{aligned}
\{s[1] = a &\rightarrow cost \geq bound^a : 1 \leq a \leq m\} \\
\{cost < bound^a &\rightarrow s[1] \neq a : 1 \leq a \leq m\} \\
\{s[1] = a \wedge s[2] = b &\rightarrow \\
cost \geq bound^{a,b} &: 1 \leq a, b \leq m \wedge a \neq b\} \\
\{cost < bound^{a,b} &\rightarrow \\
s[1] \neq a \vee s[2] = b &\rightarrow : 1 \leq a, b \leq m \wedge a \neq b\}
\end{aligned}
$$

Our current implementation has a very high cost to calculate these bounds. The worst of which being $bound^{a,b}$, which takes $O(n^4)$ time. We calculate this for all $a$ and $b$ therefore the total time for this is $O(m^2n^4)$. Because of the limited time available for development our implementation is very naive. As a result we have incorporated these bounds into the model, but the cost to calculate the bounds has not been included in the results. We do this to show the potential benefits of the technique without the disadvantage of our naive time restricted code. We have not measured the time to calculate

these bounds but we estimate the cost for a problem of size $n = 100$ $m = 100$ to be a few minutes. For the problem $GP5$ adding these bounds reduced the total time from one and a half hours to 26 minutes, and reduced the number of fails from 12324 to 3706.

## 4 The Experimental Model

Our experiments were performed using our second model, implemented in ILOG's JSolver. Problems are read in and reduced, using the operator derived from Theorem 1 above. The search process uses the dynamic reduction method described in section 3.2. The optimal solution was then found for this representative problem and proved to be optimal (either using the greatest lower bound $G$ discussed above, or by showing that no solution with smaller cost exists). The actual optimal solution was then reconstructed by re-inserting the subsumed products, and this was done in linear time.

## 5 Work Not Done

We have not fully explored the effectiveness of the reduction operator. It would be interesting to measure the amount of reduction achieved on each of the challenge problems. We have only done an ad-hoc study, which demonstrated that the use of the operator invariably reduced the time taken to obtain a solution. Indeed in some cases the improvement was quite dramatic.

We have not investigated the relative performances of the various lower bounds, i.e. to measure which one is typically best. Indeed, for each of the bounds there were example instances where the bound gave the greatest lower bound. It would also be possible to increase the number of lower bounds checked. For example, extending the notation used in bounds 4 and 5, $max\{|R_{x,y,z,w}| + 1\}$ also gives an upper bound, but is expensive to compute.

We have implemented two upper-bounds but we have not incorporated these into our model. The first upper-bound reads in the problem and takes the identity permutation as a solution and then costs that. The second upper bound greedily produces a solution in a "furthest-insertion" style. It was considered that if the furthest-insertion upper-bound was reasonably good then we could use this not only as an upper-bound but also use that permutation as a static variable ordering heuristic.

We have not systematically investigated variable or value ordering heuristics.

## Acknowledgements

We are grateful to ILOG SA for providing us with the JSolver toolkit via an Academic Grant licence. We would like to thank Barbara and Ian for producing such an exciting challenge.

## References

[1] P. Gregory, A. Miller, and P. Prosser. Solving the rehearsal problem with planning and with model checking. In Brahim Hnich and Toby Walsh, editors, *Proceedings of the Workshop on modelling and solving problems with constraints. Held in conjunction with the 16th European Conference on Artificial Intelligence (ECAI 2004).*, pages 157–171, Valencia, Spain, August 2004.

[2] Alice Miller. Improved lower bounds for solving the open stacks problem. In Ian Gent and Barbara Smith, editors, *Proceedings of the 5th Workshop on modelling and solving problems with constraints. Held in conjunction with IJCAI-05.*, 2005.

[3] B.M. Smith. Constraint Programming in Practice: Scheduling a Rehearsal. Technical report, APES, 2003.

## Appendix of results

Our experiments were performed on a Pentium4 2.8Ghz processor with 512 Mbytes of RAM running Microsoft Windows XP Professional and Java2 SDK 1.4.2.6 with an increased heap size of 512 Mbytes. Our model was coded up in ILog's JSolver. For the experiments in Tables 1 and 2 we allowed 60 seconds per instance and for Table 3 7200 seconds (2 hours) per instance. These times include model creation but do not include the time to compute the lower bounds. In Tables 1 and 2 we give the average and max results for only those instances that we found and proved optimality.

| File | % solved | mean value | time (sec) | | | number of fails | | |
|---|---|---|---|---|---|---|---|---|
| | | | mean | median | max | mean | median | max |
| problem_10_10.dat | 100 | 8.07 | 0.14 | 0.14 | 0.24 | 2 | 0 | 81 |
| problem_10_20.dat | 100 | 8.92 | 0.16 | 0.17 | 0.41 | 5 | 0 | 679 |
| problem_15_15.dat | 100 | 12.87 | 0.21 | 0.19 | 1.14 | 42 | 1 | 3440 |
| problem_15_30.dat | 92.73 | 14.02 | 0.65 | 0.22 | 35.25 | 446 | 0 | 40820 |
| problem_20_10.dat | 100 | 15.88 | 0.21 | 0.2 | 0.66 | 40 | 9 | 1422 |
| problem_20_20.dat | 85 | 17.98 | 0.85 | 0.23 | 36.47 | 1167 | 1 | 79327 |
| problem_30_10.dat | 100 | 23.95 | 0.27 | 0.24 | 2.89 | 128 | 27 | 10208 |
| problem_30_15.dat | 85.91 | 25.98 | 1.24 | 0.28 | 37.95 | 2583 | 26 | 125372 |
| problem_30_30.dat | 66.36 | 28.55 | 3.01 | 0.3 | 49.74 | 898 | 0 | 17141 |
| problem_40_20.dat | 62.73 | 36.49 | 7.03 | 0.42 | 59.77 | 6948 | 56 | 67796 |
| ShawInstances.txt | 68 | 13.68 | 0.87 | 0.33 | 9.64 | 1006 | 77 | 16015 |
| wbo_10_10 | 100 | 5.95 | 0.18 | 0.19 | 0.23 | 11 | 6 | 60 |
| wbo_10_20 | 100 | 7.35 | 0.24 | 0.22 | 0.56 | 45 | 5 | 1005 |
| wbo_10_30 | 97.5 | 8.2 | 0.29 | 0.25 | 0.77 | 20 | 3 | 244 |
| wbo_15_15 | 100 | 9.35 | 0.26 | 0.24 | 0.44 | 72 | 25 | 618 |
| wbo_15_30 | 60 | 11.62 | 2.07 | 0.36 | 48.88 | 773 | 24 | 25680 |
| wbo_20_10 | 100 | 12.9 | 0.23 | 0.22 | 0.33 | 49 | 28 | 342 |
| wbo_20_20 | 81.11 | 13.7 | 1.58 | 0.39 | 34.75 | 2392 | 119 | 71762 |
| wbo_30_10 | 100 | 20.05 | 0.29 | 0.27 | 1.06 | 119 | 66 | 2345 |
| wbo_30_15 | 98.33 | 20.96 | 0.65 | 0.42 | 5.52 | 811 | 257 | 10234 |
| wbo_30_30 | 32.14 | 22.99 | 4.09 | 0.63 | 39.92 | 1921 | 148 | 32462 |
| wbop_10_10 | 100 | 6.75 | 0.17 | 0.17 | 0.2 | 6 | 6 | 21 |
| wbop_10_20 | 100 | 8.07 | 0.26 | 0.22 | 1.72 | 48 | 0 | 1515 |
| wbop_10_30 | 95 | 8.55 | 0.69 | 0.25 | 12.55 | 212 | 1 | 4813 |
| wbop_15_15 | 100 | 10.37 | 0.24 | 0.25 | 0.39 | 54 | 31 | 494 |
| wbop_15_30 | 75 | 12.27 | 1.85 | 0.33 | 42.33 | 608 | 11 | 14609 |
| wbop_20_10 | 100 | 14.28 | 0.21 | 0.22 | 0.3 | 40 | 27 | 196 |
| wbop_20_20 | 85.56 | 14.87 | 2.09 | 0.38 | 47.45 | 2811 | 113 | 97374 |
| wbop_30_10 | 100 | 22.48 | 0.26 | 0.27 | 0.45 | 70 | 60 | 506 |
| wbop_30_15 | 95 | 22.38 | 0.57 | 0.41 | 2.33 | 625 | 207 | 4578 |
| wbop_30_30 | 47.86 | 24.37 | 4.32 | 0.49 | 53 | 2256 | 30 | 42519 |
| wbp_10_10 | 100 | 7.33 | 0.15 | 0.16 | 0.2 | 5 | 2 | 73 |
| wbp_10_20 | 100 | 8.71 | 0.17 | 0.17 | 0.28 | 6 | 0 | 106 |
| wbp_10_30 | 100 | 9.31 | 0.19 | 0.19 | 0.45 | 13 | 0 | 545 |
| wbp_15_15 | 100 | 11.05 | 0.22 | 0.2 | 0.42 | 44 | 14 | 365 |
| wbp_15_30 | 87.5 | 13.1 | 0.91 | 0.25 | 27.59 | 605 | 1 | 20633 |
| wbp_20_10 | 100 | 15.13 | 0.21 | 0.2 | 0.28 | 35 | 26 | 236 |
| wbp_20_20 | 86.67 | 15.42 | 2.15 | 0.3 | 42.33 | 4660 | 60 | 209328 |
| wbp_30_10 | 100 | 23.18 | 0.3 | 0.25 | 1.14 | 187 | 47 | 2794 |
| wbp_30_15 | 75 | 22.98 | 1.65 | 0.42 | 23.59 | 3864 | 254 | 68342 |
| wbp_30_30 | 47.86 | 24.96 | 3.64 | 0.36 | 59.2 | 2955 | 12 | 103704 |

Table 1: Times and number of fails to find the optimal solution for each data set

| File | % solved | mean value | time (sec) | | | number of fails | | |
|---|---|---|---|---|---|---|---|---|
| | | | mean | median | max | mean | median | max |
| problem_10_10.dat | 100 | 8.07 | 0.15 | 0.14 | 0.53 | 15 | 4 | 1665 |
| problem_10_20.dat | 100 | 8.92 | 0.21 | 0.17 | 12.52 | 159 | 4 | 35527 |
| problem_15_15.dat | 100 | 12.87 | 0.57 | 0.2 | 52.19 | 1234 | 9 | 160625 |
| problem_15_30.dat | 92.73 | 14.02 | 1.26 | 0.22 | 43.47 | 1803 | 4 | 29657 |
| problem_20_10.dat | 100 | 15.88 | 0.31 | 0.22 | 1.98 | 416 | 29 | 7988 |
| problem_20_20.dat | 85 | 17.98 | 3.66 | 0.22 | 54.13 | 9030 | 5 | 86966 |
| problem_30_10.dat | 100 | 23.95 | 0.57 | 0.33 | 6.45 | 1014 | 188 | 22108 |
| problem_30_15.dat | 85.91 | 25.98 | 6.54 | 0.28 | 54.72 | 15474 | 46 | 117260 |
| problem_30_30.dat | 66.36 | 28.55 | 1.68 | 0.3 | 49.74 | 265 | 5 | 6880 |
| problem_40_20.dat | 62.73 | 36.49 | 5.58 | 0.28 | 59.77 | 4268 | 5 | 26289 |
| ShawInstances.txt | 68 | 13.68 | 19 | 14.02 | 58.78 | 46529 | 35835 | 120177 |
| wbo_10_10 | 100 | 5.95 | 0.19 | 0.19 | 0.31 | 26 | 17 | 262 |
| wbo_10_20 | 100 | 7.35 | 0.78 | 0.22 | 17.41 | 624 | 21 | 17438 |
| wbo_10_30 | 97.5 | 8.2 | 1.45 | 0.25 | 30.94 | 1260 | 10 | 24299 |
| wbo_15_15 | 100 | 9.35 | 1.13 | 0.38 | 7.53 | 2793 | 252 | 26457 |
| wbo_15_30 | 60 | 11.62 | 1.77 | 0.38 | 22.59 | 541 | 32 | 10375 |
| wbo_20_10 | 100 | 12.9 | 0.3 | 0.28 | 0.77 | 284 | 176 | 2385 |
| wbo_20_20 | 81.11 | 13.7 | 9.92 | 0.55 | 54.55 | 20594 | 283 | 94726 |
| wbo_30_10 | 100 | 20.05 | 0.55 | 0.47 | 1.89 | 812 | 542 | 3755 |
| wbo_30_15 | 98.33 | 20.96 | 8.06 | 2.64 | 51.77 | 21915 | 4539 | 142875 |
| wbo_30_30 | 32.14 | 22.99 | 1.72 | 0.31 | 11.98 | 430 | 9 | 4409 |
| wbop_10_10 | 100 | 6.75 | 0.18 | 0.19 | 0.28 | 21 | 14 | 234 |
| wbop_10_20 | 100 | 8.07 | 1.4 | 0.22 | 24.11 | 2219 | 6 | 43075 |
| wbop_10_30 | 95 | 8.55 | 0.72 | 0.25 | 12.55 | 228 | 6 | 4821 |
| wbop_15_15 | 100 | 10.37 | 0.54 | 0.27 | 6.98 | 1034 | 63 | 16378 |
| wbop_15_30 | 75 | 12.27 | 2.78 | 0.27 | 42.33 | 1439 | 8 | 15190 |
| wbop_20_10 | 100 | 14.28 | 0.27 | 0.25 | 0.69 | 219 | 110 | 2020 |
| wbop_20_20 | 85.56 | 14.87 | 5.12 | 0.39 | 57.14 | 9828 | 154 | 86323 |
| wbop_30_10 | 100 | 22.48 | 0.37 | 0.36 | 0.84 | 358 | 242 | 1764 |
| wbop_30_15 | 95 | 22.38 | 7 | 1.98 | 43.05 | 18200 | 4115 | 131457 |
| wbop_30_30 | 47.86 | 24.37 | 1.56 | 0.3 | 36.69 | 280 | 7 | 6207 |
| wbp_10_10 | 100 | 7.33 | 0.16 | 0.16 | 0.33 | 23 | 7 | 474 |
| wbp_10_20 | 100 | 8.71 | 0.24 | 0.17 | 2.72 | 196 | 6 | 5901 |
| wbp_10_30 | 100 | 9.31 | 0.3 | 0.19 | 3.75 | 310 | 4 | 7527 |
| wbp_15_15 | 100 | 11.05 | 1.29 | 0.22 | 44.02 | 4457 | 31 | 183345 |
| wbp_15_30 | 87.5 | 13.1 | 2.28 | 0.25 | 42.92 | 3678 | 7 | 36740 |
| wbp_20_10 | 100 | 15.13 | 0.32 | 0.3 | 0.89 | 458 | 210 | 3095 |
| wbp_20_20 | 86.67 | 15.42 | 7.23 | 0.34 | 58.73 | 21671 | 121 | 131545 |
| wbp_30_10 | 100 | 23.18 | 0.73 | 0.5 | 2.64 | 1535 | 689 | 8030 |
| wbp_30_15 | 75 | 22.98 | 8.34 | 0.89 | 58.75 | 24118 | 774 | 121667 |
| wbp_30_30 | 47.86 | 24.96 | 1.56 | 0.3 | 33.42 | 841 | 6 | 7876 |

Table 2: Times and number of fails to prove the optimal solution for each data set

| File | solved | best value | time to best solution | fails to best solution | time to prove | fails to prove |
|---|---|---|---|---|---|---|
| Miller19 | Yes | 13 | 0.312 | 0 | 0.328 | 17 |
| GP1 | Yes | 45 | 0.484 | 0 | 0.5 | 6 |
| GP2 | Yes | 40 | 7.375 | 448 | 7.375 | 458 |
| GP3 | Yes | 40 | 11.78 | 399 | 11.78 | 411 |
| GP4 | Yes | 30 | 47.89 | 7523 | 47.89 | 7542 |
| GP5 | Yes | 95 | 1329.71 | 3700 | 1329.73 | 3706 |
| GP6 | Yes | 75 | 6812.98 | 81443 | 6813 | 81469 |
| GP7 | Yes | 75 | 2020.32 | 428 | 20.32 | 453 |
| GP8 | No | 84 | 2.985 | 22 | 7200 | 1105756 |
| NWRS1 | Yes | 3 | 0.172 | 12 | 0.172 | 15 |
| NWRS2 | Yes | 4 | 0.203 | 11 | 0.203 | 17 |
| NWRS3 | Yes | 7 | 0.25 | 28 | 0.375 | 579 |
| NWRS4 | Yes | 7 | 0.219 | 21 | 0.219 | 31 |
| NWRS5 | Yes | 12 | 0.437 | 155 | 26.5 | 77122 |
| NWRS6 | Yes | 12 | 1.015 | 354 | 203.01 | 615419 |
| NWRS7 | No | 10 | 23.15 | 38377 | 7200 | 16551474 |
| NWRS8 | No | 16 | 681.46 | 154359 | 7200 | 4554999 |
| SP1 | Yes | 9 | 0.391 | 282 | 287.58 | 1134916 |
| SP2 | No | 22 | 11.51 | 3415 | 7200 | 3009686 |
| SP3 | No | 40 | 97.7 | 8115 | 7200 | 236553 |
| SP4 | No | 65 | 31.72 | 1148 | 7200 | 219177 |

Table 3: Times and number of fails to prove the optimal solution for each problem

# Trying Hard to Solve the Simultaneously Open Stacks Problem with CP*

**Gilles Pesant**
Cork Constraint Computation Centre, University College Cork, Cork, Ireland
pesant@crt.umontreal.ca

## Abstract

This short paper presents a constraint programming approach to the Simultaneously Open Stacks Problem. It reformulates the problem as a constrained graph colouring problem. The algorithm is described in detail and experimental results are reported.

## 1 Introduction

As part of the Fifth Workshop on Modelling and Solving Problems with Constraints held during IJCAI 2005, a Modelling Challenge was proposed. The object of the First Constraint Modelling Challenge is to model the *Simultaneously Open Stacks* (SOS) problem as a constraint problem in the constraint programming or constraint logic programming language of your choice. Quoting the organisers:

> A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customer's order is started (i.e. the first product in the order has been made) a stack is created for that customer. When all the products that a customer requires have been made, the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the number of stacks that are in use simultaneously i.e. the number of customer orders that are in simultaneous production, should be minimised.

I believe the main constraint of this challenge is actually to find a solution approach based on constraint programming. In its original form, the problem consists of finding a permutation of the products (corresponding to the production order) that minimises the number of simultaneously open stacks. Every permutation is a valid candidate solution i.e. any element in the space of all permutations is feasible. Therefore we are faced with an unconstrained optimisation problem, which is doubly bad news for constraint programmers:

- Intuition and practice agree that constraint programming tends to perform relatively better when the problem at hand has constraints that we can exploit. Some constraints may be harder than others to exploit well but having no constraint at all gives us very little leverage.
- We much prefer solving satisfaction problems over optimisation problems. The natural approach to optimisation in constraint programming even recasts the problem as a succession of satisfaction problems.

Pure local search would do well on this problem and is probably the best line of attack, as evidenced by some of the scientific literature on the subject. Unfortunately we are constrained to use constraint programming.

## 2 Problem Transformation

I started out implementing a local search approach, hoping to find some role for constraint programming along the way. Then I had an idea about a transformation of the problem which would definitely involve constraint programming. In this section, I describe a transformation of the SOS problem into a *constrained* graph colouring problem.

In minimising the number of stacks used, we are in fact trying to find customers whose orders can share a stack (actually the physical space taken by a stack), given an appropriate permutation of the products. Each time $c$ customers share, we "save" $c-1$ stacks. Of course these savings can only be combined if there is a permutation of the products which is consistent with every such saving. So which customers can possibly share a stack? A necessary condition is that their orders do not have a product in common, otherwise they will each require a stack when this product is being made. Consider then a graph $G = (V, E)$ whose vertices correspond to customers and with an edge between two vertices if and only if the corresponding orders have a product in common. Such a graph has already been defined in the literature, e.g. [1]. Any subset of customers sharing a stack must correspond to a stable set in that graph. Actually, an admissible vertex colouring of the graph corresponds to a potential solution to the SOS problem, where a colour stands for a stack. It is a *potential* solution because we must make sure that there exists a permutation which separates the orders in every shared stack. In that sense it is a constrained graph colouring problem. Consider an instance with $m$ orders and $n$ products. Let $c : V \mapsto \{1, 2, \ldots, k\}$ represent a mapping of the vertices of $G$ to $k$ colours (identified as integers). Such a mapping ef-

*Research conducted while the author was on sabbatical leave from École Polytechnique de Montréal.

fectively partitions $V$ into colour classes $C_1, \ldots, C_k$. Let $P_v$ be the set of products required for the order corresponding to vertex $v \in V$, $\pi_p \in \{1, 2, \ldots, n\}$ represent the position of product $p$ in the production sequence, and $\rho_{ij} \in C_i$ the element of colour class $C_i$ whose rank is $j$ on the stack (i.e. the $j^{\text{th}}$ one to use that physical space). We must then find a $k$-colouring $c$ of $G$ such that

$$c(u) \neq c(v) \qquad (u, v) \in E \tag{1}$$

$$\pi_p \neq \pi_q \qquad 1 \leq p < q \leq n \tag{2}$$

$$\rho_{ir} \neq \rho_{is} \qquad 1 \leq r < s \leq |C_i|, 1 \leq i \leq k \tag{3}$$

$$\pi_p < \pi_q \qquad p \in P_{\rho_{ij}}, q \in P_{\rho_{ij+1}}, \; 1 \leq j \leq |C_i| - 1, 1 \leq i \leq k \tag{4}$$

Equation (1) is for the standard vertex colouring requirement, (2) ensures that the $\pi_p$'s represent a permutation of the products, and (3) similarly ensures a consistent ranking of the orders in each colour class. It is Equation (4) that bring the constrainedness to the colouring: for every colour class and then every pair of consecutively ranked orders on the stack corresponding to that class, there is a strict ordering constraint between their respective products since these orders may not overlap. This is a strong requirement that can significantly reduce the number of feasible permutations of the products, each time a colour class is fixed.

# 3 An Exact Algorithm

My exact CP algorithm solves successive constraint satisfaction problems, looking each time for a $k$-colouring of $G$ with a different value of $k$. A colouring is only valid if it is compatible with at least one product permutation constrained as described above. The colouring is done one colour at a time and the constraints on the product permutations are added gradually.

In the rest of this section I go into more detail. Constraint programming actually comes into play in Section 3.3.

## 3.1 Preprocessing

Often we can reduce the size of an instance by disregarding some of the products that are dominated. Product $i$ *dominates* product $j$ when the set of orders requiring $j$ is a subset of the set of orders requiring $i$. Given any solution to the instance without considering product $j$ (and in particular any optimal solution), we can construct a solution to the original instance (i.e. including $j$) using the same number of open stacks simply by inserting $j$ immediately after $i$ in the production sequence: clearly when $j$ is being made, every order involved already has an open stack from product $i$ just before ([1]).

Once dominated products have been removed, I build graph $G$ as defined in the previous section. If that graph is a clique then no two orders may share a stack so we can immediately conclude that the minimum number of stacks is the number of orders.

In the other, more interesting cases, lower and upper bounds on the number of colours necessary to colour the graph (i.e. the number of stacks necessary) are computed once before the start of the search. The upper bound is very simple and corresponds to the number of orders minus one: if the graph is not a clique then there must be a pair of vertices that are not adjacent and the corresponding orders can easily share a stack, using a production sequence which groups their respective products, thus separating them. There are many lower bounds possible and I essentially follow what is described in [1] but initially proposed in [2]. They suggest taking the minimum of three values:

- the maximum number of orders per product;
- $1+$ the smallest degree of a vertex in $G$;
- the size of a clique which is a minor of $G$.

For the latter, graph minors are computed by heuristic arc contraction as described in [1]. To these, I add a fourth alternative: the size of a subgraph of $G$ which forms a clique, obtained through a simple constructive heuristic. That clique is actually constructed to break some symmetries during the graph colouring phase so I use it here "for free" and experiments confirm that it is sometimes the best of the four.

## 3.2 Graph Colouring

The strategy to select the number of colours $k$ that we try is fairly simple. We start at the lower bound and look for a valid constrained colouring. If we prove that there is no solution for $k$ colours, the lower bound is set to $k + 1$ and we repeat the process with a number of colours that is half way between the current number and the upper bound. If a solution is found, the upper bound is set to $k$ and we repeat the process with $k-1$ colours. Otherwise we have reached a given cutoff time and we repeat the process with a number of colours that is half way between the current number and the upper bound, without updating the lower bound. We stop when the lower and upper bounds coincide, in which case we have proven optimality, or when a global cutoff time is reached.

The actual graph colouring proceeds one colour at a time and selects all the vertices to share that colour. It is advantageous to proceed in this way because as soon as a colour class is closed, we can add constraints on the product permutation (see Section 3.3). The colours are processed in lexicographic order and so are the vertices.

In order to break some obvious symmetries, I preassign vertices in some of the colour classes and impose a lexicographic ordering on the rest of them. As indicated before, a simple constructive heuristic identifies a hopefully large clique in $G$. Each vertex from that clique $\{v_1, \ldots, v_\ell\}$ must belong to a distinct colour class so I can safely preassign $v_i$ to $C_i$. For the remaining colour classes $C_{\ell+1}, \ldots, C_k$, I add constraints $C_i \prec C_{i+1}$, $\ell + 1 \leq i \leq k - 1$, meaning that the smallest index of a vertex in $C_i$ should be less than the smallest index of a vertex in $C_{i+1}$. No doubt much more could be done to break symmetries.

## 3.3 Product Permutation

As soon as a colour class $C_i$ is closed (i.e. all the vertices of that colour have been selected), we have identified the corresponding set of orders sharing a stack. This in turn constrains the space of feasible product permutations: there must exist an ordering of the elements of $C_i$ consistent with an ordering of the products which separates the orders. As each colour

class is closed, the number of feasible product permutations decreases. If that number reaches zero (i.e. the domain of a $\pi_p$ becomes empty), we can backtrack.

The basic CP model corresponding to (2)-(4) is:

$$\texttt{alldifferent}((\pi_p)_{1\leq p\leq n}) \quad (5)$$

$$\texttt{alldifferent}((\rho_{ij})_{1\leq j\leq |C_i|-1}) \qquad 1\leq i\leq k \quad (6)$$

$$\pi_p < \pi_q \qquad p\in P_{\rho_{ij}}, q\in P_{\rho_{ij+1}}, \; 1\leq j\leq |C_i|-1, \; 1\leq i\leq k \quad (7)$$

$$\pi_p \in \{1,2,\ldots,n\} \qquad 1\leq p\leq n \quad (8)$$

$$\rho_{ij}\in C_i \qquad 1\leq j\leq |C_i|-1, \; 1\leq i\leq k \quad (9)$$

I search this constrained solution space in the following way. Constraints (5) and (8) are present from the start. Whenever a colour class is closed, say $C_i$, we may add constraints (6), (7), and (9) for that $i$. I then look for a feasible assignment of the $(\rho_{ij})_{1\leq j\leq |C_i|-1}$ variables ($i$ fixed), which allows constraints (7) to do some filtering. This assignment may need of course to be backtracked over. Only once the colouring is complete do I seek a feasible assignment of the $(\pi_p)_{1\leq p\leq n}$ variables.

It is possible to tighten that model by adding implied unary constraints. If the orders preceeding an order $v$ on a stack they share involve a total of $t$ products and since these products are distinct, a product involved in $v$ must be sequenced after these $t$ other products. The same reasoning applies for the orders succeeding $v$, giving us:

$$\sum_{\ell=1}^{j-1}|P_{\rho_{i\ell}}| < \pi_p \leq n - \sum_{\ell=j+1}^{|C_i|}|P_{\rho_{i\ell}}| \qquad p\in P_{\rho_{ij}}, \; 1\leq j\leq |C_i|, \; 1\leq i\leq k \quad (10)$$

We could go further than these "monochrome" constraints and add more global unary inequalities that keep track, for every product $p$, of all the other products necessarily before or after $p$ in the sequence, based on the same reasoning but applied collectively to every colour class closed so far. I have tried this but limited experimentation seems to indicate that even though it decreases the number of backtracks required, it slows down the algorithm.

## 4 Results

The algorithm was run on all the instances provided for the challenge. A single run was performed on each instance since the algorithm is deterministic and identical parameters were used throughout (except for the cutoff time, as indicated below). The computer used was a Sunfire 4800 equipped with 900MHz processors and 8Gb of RAM. The constraint programming library used was ILOG Solver 4.4. Because of the great number of instances, I had to keep the cutoff time low. Table 1 reports on the files containing multiple instances, with a cutoff time of one minute. In order to see how sensitive these results are to the choice of the cutoff time, I also ran the same experiments with a five minute cutoff time (Table 2). For the individual instances, which are generally larger, a cutoff time of one hour was selected.

Here are a few observations. The smaller instances are all solved to optimality but that percentage goes down as the size increases, as should be expected. For a fixed number of orders, the performance generally increases with the number of products: this is not surprising since more products mean fewer opportunities of sharing a stack between orders and lower combinatorics for this algorithm. The very low median values also indicate that a majority of instances are very easily solved by the algorithm. One exception is the set "ShawInstances.txt" which appears to be more balanced. Its percentage of instances solved to optimality also climbs quickly as more time is given (from a very low 16% within one minute to 52% within five minutes). Upon further investigation, the algorithm actually solves every instance of that set to optimality within twenty minutes. Many of the individual instances are solved to optimality quickly but the algorithm also performs poorly on some of the larger ones.

## 5 Discussion

In this challenge, I aimed to design a solution approach in which constraint programming plays a significant role at the modeling level. I purposely directed most of my efforts to the part that involved CP. I also generally kept it simple, no bells and whistles, to make it easier to analyse and compare the core ideas.

One consequence of this is that the graph colouring component is very basic and there lies a potential source of improvement. I believe much of the vast expertise in solving graph colouring problems could be added here and the algorithm's performance would greatly benefit from it. In my opinion, this approach is interesting because it calls on a well-studied problem, graph colouring, with an extra twist that CP can handle well.

## Acknowledgements

This research was conducted while I was on sabbatical leave at the Cork Constraint Computation Centre (4C) and was supported in part by Science Foundation Ireland. I wish to thank the other members of 4C who participated in this challenge for stimulating discussions, comparison of results, and pointers to relevant literature.

## References

[1] J. C. Becceneri, H. H. Yanasse, and N. Y. Soma. A Method for Solving the Minimization of the Maximum Number of Open Stacks Problem within a Cutting Process. *Computers & Operations Research*, 31:2315–2332, 2004.

[2] H. H. Yanasse, J. C. Becceneri, and N. Y. Soma. Bounds for a Problem of Sequencing Patterns. *Pesquisa Operacional*, 19(2):49–77, 1999.

| File | % solved | mean value | time (sec) | | | number of fails | | |
|---|---|---|---|---|---|---|---|---|
| | | | mean | median | max | mean | median | max |
| problem_10_10.dat | 100 | 8.03 | 0.01 | 0.01 | 0.30 | 63 | 0 | 8268 |
| problem_10_20.dat | 100 | 8.92 | 0.01 | 0.01 | 0.17 | 22 | 0 | 2534 |
| problem_15_15.dat | 99.27 | 12.87 | 1.05 | 0.01 | 57.02 | 15156 | 0 | 1136844 |
| problem_15_30.dat | 100 | 14.02 | 0.26 | 0.01 | 12.23 | 951 | 0 | 46392 |
| problem_20_10.dat | 80.36 | 16.00 | 2.69 | 0.01 | 56.41 | 40322 | 2 | 988414 |
| problem_20_20.dat | 90.91 | 18.00 | 2.17 | 0.01 | 52.34 | 12073 | 0 | 312110 |
| problem_30_10.dat | 60.73 | 24.60 | 2.37 | 0.01 | 58.83 | 29055 | 1 | 936763 |
| problem_30_15.dat | 71.36 | 26.39 | 2.26 | 0.01 | 59.62 | 14581 | 0 | 378698 |
| problem_30_30.dat | 86.36 | 28.50 | 0.79 | 0.01 | 28.65 | 1134 | 0 | 46246 |
| problem_40_20.dat | 70.91 | 36.87 | 1.10 | 0.01 | 30.93 | 3847 | 0 | 113004 |
| ShawInstances.txt | 16.00 | 13.84 | 31.97 | 32.72 | 59.82 | 183205 | 206948 | 355744 |
| wbo_10_10 | 100 | 5.93 | 0.05 | 0.01 | 0.54 | 1824 | 22 | 33509 |
| wbo_10_20 | 100 | 7.35 | 0.11 | 0.01 | 2.58 | 695 | 2 | 13354 |
| wbo_10_30 | 100 | 8.20 | 0.20 | 0.01 | 2.78 | 514.08 | 0 | 7158 |
| wbo_15_15 | 76.67 | 9.53 | 5.77 | 0.03 | 47.64 | 109751 | 119 | 1383663 |
| wbo_15_30 | 76.67 | 11.78 | 4.28 | 0.01 | 55.48 | 11158 | 0 | 234343 |
| wbo_20_10 | 55.71 | 13.33 | 5.55 | 0.23 | 32.74 | 71561 | 2454 | 397844 |
| wbo_20_20 | 55.56 | 14.81 | 1.23 | 0.01 | 15.54 | 4195 | 11 | 56077 |
| wbo_30_10 | 30.00 | 21.32 | 4.73 | 1.21 | 35.51 | 56092 | 12499 | 447865 |
| wbo_30_15 | 39.17 | 22.74 | 3.04 | 0.16 | 55.15 | 19841 | 783 | 386402 |
| wbo_30_30 | 51.43 | 25.25 | 1.57 | 0.02 | 57.80 | 2203 | 0 | 87438 |
| wbop_10_10 | 100 | 6.75 | 0.06 | 0.01 | 0.58 | 2481 | 32 | 32166 |
| wbop_10_20 | 100 | 8.08 | 0.23 | 0.01 | 4.67 | 1121 | 1 | 23807 |
| wbop_10_30 | 100 | 8.55 | 0.15 | 0.01 | 3.57 | 353 | 0 | 9753 |
| wbop_15_15 | 73.33 | 10.60 | 2.33 | 0.05 | 41.63 | 37878 | 170 | 859363 |
| wbop_15_30 | 85.00 | 12.22 | 5.66 | 0.01 | 39.26 | 12311 | 0 | 158432 |
| wbop_20_10 | 52.50 | 14.60 | 2.19 | 0.08 | 44.62 | 25965 | 604 | 533277 |
| wbop_20_20 | 66.67 | 16.06 | 4.52 | 0.01 | 49.06 | 18820 | 0 | 214185 |
| wbop_30_10 | 50.00 | 24.00 | 1.07 | 0.69 | 4.45 | 9357 | 5007 | 42331 |
| wbop_30_15 | 50.00 | 24.50 | 2.10 | 0.04 | 9.83 | 10543 | 178 | 49722 |
| wbop_30_30 | 57.86 | 26.34 | 1.53 | 0.01 | 57.79 | 2019 | 0 | 81157 |
| wbp_10_10 | 100 | 7.28 | 0.01 | 0.01 | 0.08 | 134 | 3 | 2677 |
| wbp_10_20 | 100 | 8.71 | 0.02 | 0.01 | 0.73 | 123 | 0 | 7512 |
| wbp_10_30 | 100 | 9.31 | 0.01 | 0.01 | 0.16 | 13 | 0 | 555 |
| wbp_15_15 | 91.67 | 11.08 | 4.59 | 0.01 | 48.24 | 120432 | 48 | 2568624 |
| wbp_15_30 | 94.17 | 13.11 | 2.16 | 0.01 | 45.91 | 10370 | 0 | 218073 |
| wbp_20_10 | 72.50 | 15.30 | 5.89 | 0.04 | 30.85 | 74714 | 295 | 360531 |
| wbp_20_20 | 67.78 | 15.80 | 1.98 | 0.01 | 48.75 | 10862 | 0 | 369693 |
| wbp_30_10 | 50.00 | 24.33 | 1.11 | 0.12 | 11.72 | 10543 | 981 | 113464 |
| wbp_30_15 | 50.00 | 24.23 | 1.03 | 0.03 | 8.44 | 5226 | 87 | 46843 |
| wbp_30_30 | 63.57 | 26.09 | 2.20 | 0.01 | 54.98 | 3529 | 0 | 103541 |

Table 1: Performance of the algorithm for each data set given a one minute cutoff time.

| File | % solved | mean value | time (sec) | | | number of fails | | |
|---|---|---|---|---|---|---|---|---|
| | | | mean | median | max | mean | median | max |
| problem_10_10.dat | 100 | 8.03 | 0.01 | 0.01 | 0.30 | 63 | 0 | 8268 |
| problem_10_20.dat | 100 | 8.92 | 0.01 | 0.01 | 0.17 | 22 | 0 | 2534 |
| problem_15_15.dat | 99.82 | 12.87 | 1.63 | 0.01 | 148.97 | 21990 | 0 | 1682021 |
| problem_15_30.dat | 100 | 14.02 | 0.26 | 0.01 | 12.23 | 951 | 0 | 46392 |
| problem_20_10.dat | 84.36 | 15.94 | 9.78 | 0.01 | 294.00 | 162810 | 4 | 6060689 |
| problem_20_20.dat | 92.73 | 17.99 | 5.15 | 0.01 | 231.60 | 26999 | 0 | 1373590 |
| problem_30_10.dat | 63.82 | 24.52 | 11.16 | 0.01 | 297.42 | 137742 | 3 | 3793694 |
| problem_30_15.dat | 71.81 | 26.30 | 4.22 | 0.01 | 118.26 | 27776 | 0 | 888522 |
| problem_30_30.dat | 89.09 | 28.48 | 5.12 | 0.01 | 208.91 | 8580 | 0 | 370158 |
| problem_40_20.dat | 75.45 | 36.85 | 4.44 | 0.01 | 148.94 | 17039 | 0 | 622593 |
| ShawInstances.txt | 52.00 | 13.68 | 98.74 | 91.57 | 221.67 | 593952 | 516640 | 1509980 |
| wbo_10_10 | 100 | 5.93 | 0.05 | 0.01 | 0.54 | 1824 | 22 | 33509 |
| wbo_10_20 | 100 | 7.35 | 0.11 | 0.01 | 2.58 | 695 | 2 | 13354 |
| wbo_10_30 | 100 | 8.20 | 0.20 | 0.01 | 2.78 | 514.08 | 0 | 7158 |
| wbo_15_15 | 85.00 | 9.40 | 18.92 | 0.13 | 170.70 | 408472 | 668 | 6918121 |
| wbo_15_30 | 86.67 | 11.72 | 21.80 | 0.02 | 207.86 | 41807 | 3 | 320574 |
| wbo_20_10 | 57.14 | 13.20 | 10.20 | 0.24 | 190.18 | 153138 | 2454 | 3335626 |
| wbo_20_20 | 64.44 | 14.57 | 27.88 | 0.02 | 287.45 | 135922 | 36 | 1298301 |
| wbo_30_10 | 36.00 | 21.24 | 26.15 | 4.68 | 243.63 | 306058 | 48578 | 2793437 |
| wbo_30_15 | 44.17 | 22.40 | 25.15 | 0.19 | 258.37 | 185333 | 1001 | 1956629 |
| wbo_30_30 | 55.71 | 24.99 | 12.39 | 0.02 | 170.82 | 19021 | 1 | 253642 |
| wbop_10_10 | 100 | 6.75 | 0.06 | 0.01 | 0.58 | 2481 | 32 | 32166 |
| wbop_10_20 | 100 | 8.08 | 0.23 | 0.01 | 4.67 | 1121 | 1 | 23807 |
| wbop_10_30 | 100 | 8.55 | 0.15 | 0.01 | 3.57 | 353 | 0 | 9753 |
| wbop_15_15 | 88.33 | 10.42 | 36.54 | 0.15 | 274.62 | 682564 | 729 | 9306640 |
| wbop_15_30 | 86.67 | 12.18 | 7.00 | 0.01 | 73.06 | 14487 | 0 | 158472 |
| wbop_20_10 | 75.00 | 14.45 | 39.28 | 0.15 | 274.77 | 583197 | 1258 | 4719165 |
| wbop_20_20 | 66.67 | 15.52 | 5.89 | 0.01 | 85.95 | 25251 | 0 | 420993 |
| wbop_30_10 | 50.00 | 23.93 | 1.07 | 0.65 | 4.43 | 9357 | 5007 | 42331 |
| wbop_30_15 | 50.00 | 23.95 | 2.10 | 0.04 | 9.81 | 10543 | 178 | 49722 |
| wbop_30_30 | 63.57 | 26.15 | 14.11 | 0.01 | 280.10 | 22278 | 0 | 498671 |
| wbp_10_10 | 100 | 7.28 | 0.01 | 0.01 | 0.08 | 134 | 3 | 2677 |
| wbp_10_20 | 100 | 8.71 | 0.02 | 0.01 | 0.73 | 123 | 0 | 7512 |
| wbp_10_30 | 100 | 9.31 | 0.01 | 0.01 | 0.16 | 13 | 0 | 555 |
| wbp_15_15 | 98.33 | 11.05 | 18.15 | 0.02 | 282.00 | 540213 | 167 | 10230988 |
| wbp_15_30 | 95.83 | 13.11 | 4.19 | 0.01 | 178.10 | 25740 | 0 | 1502581 |
| wbp_20_10 | 75.00 | 15.28 | 7.65 | 0.03 | 60.09 | 97469 | 295 | 755475 |
| wbp_20_20 | 68.89 | 15.66 | 4.53 | 0.01 | 161.24 | 25340 | 0 | 907633 |
| wbp_30_10 | 50.00 | 24.18 | 1.12 | 0.12 | 11.72 | 10543 | 981 | 113464 |
| wbp_30_15 | 53.33 | 24.20 | 7.01 | 0.03 | 175.87 | 55936 | 103 | 1101325 |
| wbp_30_30 | 65.00 | 25.81 | 7.15 | 0.01 | 268.93 | 12353 | 0 | 528258 |

Table 2: Performance of the algorithm for each data set given a five minute cutoff time.

| File | solved | best value | time to best solution | fails to best solution | time to prove | fails to prove |
|---|---|---|---|---|---|---|
| Miller19 | | 13 | 60.56 | 113521 | 3600.00 | 3295988 |
| GP1 | √ | 45 | 0.29 | 96 | 0.29 | 96 |
| GP2 | √ | 40 | 0.52 | 136 | 0.52 | 136 |
| GP3 | √ | 40 | 0.15 | 42 | 0.15 | 42 |
| GP4 | √ | 30 | 0.38 | 678 | 0.38 | 678 |
| GP5 | √ | 95 | 2.98 | 95 | 2.98 | 95 |
| GP6 | √ | 75 | 0.80 | 36 | 0.80 | 36 |
| GP7 | √ | 75 | 7.46 | 3292 | 7.46 | 3292 |
| GP8 | | 90 | 340.30 | 66188 | 3600.00 | 983359 |
| NWRS1 | √ | 3 | 0.01 | 4 | 0.01 | 4 |
| NWRS2 | √ | 4 | 0.01 | 13 | 0.01 | 13 |
| NWRS3 | √ | 7 | 0.01 | 45 | 0.01 | 45 |
| NWRS4 | √ | 7 | 0.04 | 194 | 0.04 | 194 |
| NWRS5 | √ | 12 | 0.01 | 13 | 0.01 | 13 |
| NWRS6 | √ | 12 | 0.01 | 8 | 0.01 | 8 |
| NWRS7 | √ | 10 | 17.95 | 49321 | 17.95 | 49321 |
| NWRS8 | √ | 16 | 36.94 | 40083 | 36.94 | 40083 |
| SP1 | | 12 | 60.42 | 1439078 | 3600.00 | 70878095 |
| SP2 | | 42 | 180.15 | 921702 | 3600.00 | 15123139 |
| SP3 | | 62 | 180.65 | 188937 | 3600.00 | 3904945 |
| SP4 | | 90 | 241.48 | 164766 | 3600.00 | 2506194 |

Table 3: Performance of the algorithm on individual instances given a one hour cutoff time.

# Open Stack Minimisation by Local Search and Reverse Dominance Reasoning

**Steven Prestwich**
Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
s.prestwich@cs.ucc.ie

## Abstract

Dominance reasoning can be used to add constraints to a model, pruning the search space and improving backtrack search. This paper proposes the reverse approach for local search: reformulation to add artificial solutions that are dominated by true solutions. On the open stack minimization problem this technique can super-exponentially increase the solution density, significantly improving local search performance. However, experiments indicate that solution density is not the only important model property.

## 1 Introduction

The idea that higher solution density[1] makes a problem easier to solve seems natural, and is usually assumed to have at least some effect on search performance [1; 3; 5; 11; 12]. But little has been done to exploit this conjecture. One way to increase the solution density of a problem is to increase problem symmetry by reformulation, and this *supersymmetry* technique has been shown to improve local search performance [7]. Any solution to a supersymmetric model is either a true solution to the problem, or can be transformed to one by applying a symmetry transformation.

Unfortunately supersymmetric reformulations have been hard to find — perhaps because supersymmetry is the inverse of *symmetry breaking by reformulation*, which is known to be powerful but non-trivial to apply [2]. Moreover, problem features other than solution density also seem to affect search cost, and there seems to be no agreement on what these features are. More than one CSP-to-SAT encoding (the *totally weakened* and *log* encodings [8]) increases solution density but can make the problem harder to solve by local search. Nevertheless, it seems worth pursuing the idea of increasing solution density by reformulation. Choosing a good model can be as important as choosing a good search algorithm, and new modelling techniques are of interest to the CP community.

This paper extends supersymmetry to the more general notion of *dominance*. The idea is to reformulate a problem in such a way that new, dominated "pseudo-solutions" are added, increasing the solution density of the model and boosting local search performance (we shall ignore backtrack search). This might be done by simply removing or weakening constraints in the model. However, this must be done very carefully: we must ensure that any pseudo-solution can be transformed to a true solution that dominates it. As with supersymmetry this is non-trivial, but this paper demonstrates that it is possible and can yield good results.

[1] Defined as the number of solutions divided by the number of total variable assignments.

## 2 Modelling open stack minimisation

We use the following problem as an illustration. A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customer's order is started a stack is created for that customer. When all the products that a customer requires have been made the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the number of stacks that are in use simultaneously should be minimized.

### 2.1 A matrix representation

We can model the problem as a matrix $M$ of binary variables, in which the columns correspond to the products required by the customers, and the rows to the customers' orders. Matrix entry $M_{ij} = 1$ if and only if customer $i$ has ordered some quantity of product $j$ (the quantity ordered is irrelevant). Any 0 with a 1 in a column to its left, and another 1 in a column to its right, is counted as a 1. A score is assigned to each column: its number of 1s (including 0s counted as 1s). The score of the matrix is the maximum of its column scores. The problem is to permute the columns of the matrix to minimise its score.

### 2.2 An integer model

The matrix representation can be modelled as an integer program. Suppose the matrix has $R$ rows and $C$ columns. Assume that each customer orders at least one product (if not then that customer can be removed from the problem), so that every row has an open order of length at least 1. In the following, $i$ is an integer with range $1 \ldots R$ and $j, k$ are integers with range $1 \ldots C$. Define variables $p_{jk}$ such that $p_{jk} = 1$ denotes that product $j$ is placed in column $k$. Each product

must be placed in exactly one column and each column must receive exactly one product:

$$\sum_k p_{jk} = 1 \qquad \sum_j p_{jk} = 1 \tag{1}$$

(These two sets of constraints imply each other.) To model the idea of 1s influencing 0s to their left and right, define variables $l_{ij}$ and $r_{ij}$ such that $l_{ik} = 1$ if and only if $M_{ik}$ has a 1 to its left, and $r_{ik} = 1$ if and only if $M_{ik}$ has a 1 to its right. We need constraints

$$p_{jk} \leq l_{ik} \qquad p_{jk} \leq r_{ik} \tag{2}$$

where $M_{ij} = 1$, and

$$l_{ik} \leq l_{i\,k+1} \qquad r_{i\,k+1} \leq r_{ik} \tag{3}$$

We also define variables $o_{ik}$ such that $o_{ik} = 1$ denotes that there is an open order on row $i$ at column $k$:

$$l_{ik} + r_{ik} \leq 1 + o_{ik} \tag{4}$$

The $l, r, o$ variables model the fact that any 0 between two 1s is counted as though it were a 1. The objective is to find consistent values for the $p, l, r, o$ variables while minimising

$$\max_k \sum_i o_{ik}$$

This optimisation problem can be solved as a series of constraint satisfaction problems (CSPs) with additional linear constraints:

$$\sum_i o_{ik} \leq \Omega \tag{5}$$

where $\Omega$ is an integer variable. We must minimise $\Omega$: starting with $\Omega = R$ a feasible solution is found; $R$ is then decremented and the search resumed with the same variable assignments (to exploit any solution clustering); and so on until timeout occurs.

### 2.3 Creating new solutions

We would like to remove or weaken some constraints in order to create new pseudo-solutions, thus increasing the solution density of the problem. But we cannot simply remove arbitrary constraints, because solutions to the resulting problem might not be solutions to the original problem, or they might be solutions with larger scores. We must do it in such a way that any pseudo-solution can be transformed into a true solution of equal or lower score: in other words pseudo-solutions must be dominated by true solutions.

Suppose we weaken constraints (1) to

$$\sum_k p_{jk} \geq 1 \tag{6}$$

Now each product may be placed in more than one position in the sequence, and some sequence positions might receive no products. At first sight this appears useless because solutions might not be permutations. However, it can be seen as a model for a generalised form of the problem: find a *sequence of sets of products* such that each product appears in at least one set, and the orders are open in set $k$ if there is a product required in an order in another set $i$ that appears in a set before or after set $k$, or in $k$ itself. Such a sequence is a pseudo-solution for the original problem.

### 2.4 From pseudo-solutions to solutions

A dominating solution can be derived from a pseudo-solution as follows. For any product that appears in more than one set, remove all but one of its appearances. For example the pseudo-solution

$$(\{4\}, \{4\}, \{4, 5\}, \{\}, \{1, 2, 3\})$$

becomes

$$(\{4\}, \{\}, \{5\}, \{\}, \{1, 2, 3\})$$

if we delete all but the first appearance of each product. We now have the same number of products as sets, and can obtain a permutation by moving products without violating the ordering among sets. For example 5 can be moved one set to the left to obtain

$$(\{4\}, \{5\}, \{\}, \{\}, \{1, 2, 3\})$$

then 1 moved two sets to the left to obtain

$$(\{4\}, \{5\}, \{1\}, \{\}, \{2, 3\})$$

and finally 2 moved one set to the left to obtain the permutation

$$(\{4\}, \{5\}, \{1\}, \{2\}, \{3\})$$

### 2.5 Increase in optimal solution density

The effect on the density of optimal (pseudo-)solutions can be spectacular. Consider a problem that we shall call $A_N$, represented by a $2 \times 2N$ matrix:

$$\left[\begin{array}{c|c} 00\ldots0 & 1\ldots11 \\ 11\ldots1 & 0\ldots00 \end{array}\right]$$

The column permutation shown has 1 open stack in any column, and this is optimal. We may permute the left and right columns separately, and reverse all columns, to obtain another optimal permutation. Thus there are $2(N!)^2$ optimal permutations, and any other permutation such as

$$\left[\begin{array}{c} 0\ldots01\mathbf{0}1\ldots1 \\ 1\ldots1\mathbf{0}10\ldots0 \end{array}\right]$$

has at least one column with two open stacks (because of the zeroes shown in **bold**). However, there are many more optimal pseudo-solutions. Considering only those cases in which all (0,1)-columns are in the $N$ left (or right) columns and all (1,0)-columns are in the $N$ right (or left) columns, there are $2(2^N - 1)^{2N}$ optimal pseudo-solutions. So the number of optimal solutions has been increased by a factor of at least

$$\frac{2(2^N-1)^{2N}}{N!} > \frac{2^{N^2}}{N^N} = \frac{2^{N^2}}{2^{\log_2 N^N}} = 2^{N^2 - N\log_2 N} \approx 2^{N^2}$$

A super-exponential increase in solution density might be expected to have an effect on local search performance.

At the other extreme, for some problems there will be exactly the same number of solutions in both models: in other words the reformulation introduces no pseudo-solutions. Consider a problem we shall call $B_N$, represented by the $2N \times N$ matrix

$$\left[\frac{U_N}{L_N}\right]$$

where $U_N$ and $L_N$ are upper and lower diagonal $N \times N$ matrices respectively:

$$U_N = \begin{bmatrix} 111\ldots111 \\ 011\ldots111 \\ 001\ldots111 \\ \vdots \\ 000\ldots001 \end{bmatrix} \qquad L_N = \begin{bmatrix} 100\ldots000 \\ 110\ldots000 \\ 111\ldots000 \\ \vdots \\ 111\ldots111 \end{bmatrix}$$

Notice that every column has exactly $(N+1)$ 1s and no 0 has a 1 to the left and the right, but in any other permutation this is no longer true. For example with $B_4$ if we exchange the middle two columns:

$$\begin{bmatrix} 1111 \\ 0111 \\ 0011 \\ 0001 \\ 1000 \\ 1100 \\ 1110 \\ 1111 \end{bmatrix} \longrightarrow \begin{bmatrix} 1111 \\ 0111 \\ 01\mathbf{0}1 \\ 0001 \\ 1000 \\ 1\mathbf{0}10 \\ 1110 \\ 1111 \end{bmatrix}$$

then both have 6 open stacks instead of 5. So the matrix has two optimal solutions, each with $(N+1)$ open stacks: one is with $U_N$ and $L_N$ as shown, the other is obtained by reversing their columns. Notice also that each column has at least one 1 where the other has a 0 in the same row. Therefore no two columns can be placed in the same set without increasing the maximum number of open stacks, and there are only two optimal pseudo-solutions (corresponding to the optimal solutions).

### 2.6 Discussion of the models

We shall refer to the original integer model as model 1, and the new one as model 3. We shall also consider an intermediate model 2 in which all sets in the pseudo-solution must be non-empty, which is obtained by replacing (1) with

$$\sum_k p_{jk} \geq 1 \qquad \sum_j p_{jk} \geq 1 \tag{7}$$

Model 2 might also have super-exponentially more solutions than model 1. Considering only optimal pseudo-solutions to $A_N$ in which product 1 appears in every set, there are $2(2^{N-1}-1)^{2(N-1)}$ of them, which is still at least $O(2^{N^2})$ more than model 1 solutions.

The weakened constraints of models 2 and 3 are analogous to the removal of "at-most-one" clauses in a well-known SAT encoding of CSPs [9]. This allows a CSP variable to be assigned more than one domain value, thus creating pseudo-solutions for the CSP. This is known to improve local search performance (though it is not certain that increased solution density is the explanation). Those pseudo-solutions are very closely related to CSP solutions, which can be obtained simply by taking any assigned value for each CSP variable. The pseudo-solutions of models 2 and 3 require a far less obvious transformation to obtain true solutions, are less closely related in terms of Hamming distance, and are dominated by true solutions in terms of the objective function in the open stacks problem.

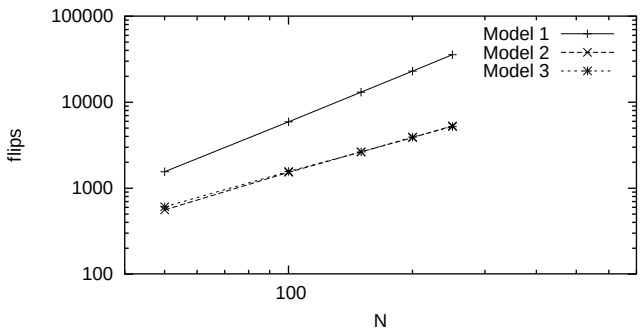



Figure 1: Results on the $A_N$ benchmarks

All three models have $O(R^2)$ $p, l, r$ variables and $O(RC)$ $o$ variables, giving a total of $O(R(R+C))$. They have $O(C)$ constraints (1) or (6) or (7) of size $O(R)$, $O(\Delta RC^2)$ constraints (2) of constant size where $\Delta$ is the matrix density,[2] $O(RC)$ constraints (3) of constant size, $O(RC)$ constraints (4) of constant size, and $O(C)$ constraints (5) of size $O(R)$, giving a total space complexity of $O(RC(1+C\Delta))$. For sparse matrices the space complexity is low, making the models suitable for large problems in which each customer orders a small number of products.

## 3 Experiments

Integer programs can be solved by local search. We use an unpublished algorithm based on a recent SAT algorithm called VW2 [6]. The new algorithm (and VW2) is related to Walksat variant B [4; 10], and uses a modified objective function that dynamically weights variables to improve search diversification. In the experiments in this section the Walksat noise parameter $p$ is set to 0.05, the VW2 $s$ parameter is set to 0.1, and the VW2 $c$ parameter to 0.000001.

For the $A_N$ benchmark (with super-exponentially many optimal pseudo-solutions) the results are shown in Figure 1. The graph is a log-log plot so the straight lines show that the search effort is polynomial in $N$. Models 2 and 3 are indistinguishable, but model 1 has a steeper gradient and therefore a higher polynomial degree. We also experimented with $N = 500$ and tuned the search algorithm parameters, and the results were very similar: models 1 and 2 took tens of seconds to solve while model 3 took tens of minutes.

For the $B_N$ benchmark (with no optimal pseudo-solutions) the results are shown in Figure 2 and are quite different. Model 1 is now the best, model 2 is slightly worse, and model 3 much worse. However, model 2 appears to scale better than model 1 as $N$ increases: unfortunately the crossover (if there is one) occurs as the problems become expensive to solve. Still, these small differences might vanish under more careful tuning of the search algorithm. The main point about this graph is that model 2 scales comparably with model 1, and much better than model 3.

[2] Defined as the number of 1-entries divided by the total number of entries.

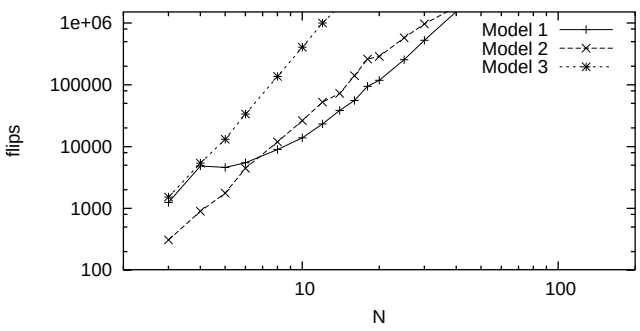



Figure 2: Results on the $B_N$ benchmarks

## 4 Conclusion

Model 2 emerged as the most robust in our experiments. We used unrealistic problems that were solved in polynomial time, but in experiments on some Challenge instances model 2 also gave the best results. It is hoped that the new model and local search algorithm will perform fairly well on at least some of the Challenge benchmarks. However, our main aim was to demonstrate the idea of reformulating a problem to increase its solution density, which generalises the technique of supersymmetry.

## References

[1] D. Clark, J. Frank, I. P. Gent, E. MacIntyre, N. Tomov, T. Walsh. Local Search and the Number of Solutions. *Second International Conference on Principles and Practices of Constraint Programming, Lecture Notes in Computer Science* vol. 1118, Springer, 1996, pp. 119–133.

[2] I. P. Gent, J.-F. Puget. Symmetry Breaking in Constraint Programming. Tutorial, Tenth International Conference on Principles and Practice of Constraint Programming, Toronto, Canada, 2004.

[3] Y. Hanatani, T. Horiyama, K. Iwama. Density Condensation of Boolean Formulas. *Sixth International Conference on the Theory and Applications of Satisfiability Testing*, Santa Margherita Ligure, Italy, 2003, pp. 126–133.

[4] D. A. McAllester, B. Selman, H. A. Kautz. Evidence for Invariants in Local Search. *Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, AAAI Press / MIT Press, 1997, pp. 321–326.

[5] A. Parkes. Clustering at the Phase Transition. *Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference* AAAI Press / MIT Press, 1997, pp. 340–345.

[6] S. D. Prestwich. Random Walk With Continuously Smoothed Variable Weights. *Eighth International Conference on Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science* vol. 3569, Springer, 2005, pp. 203–215.

[7] S. D. Prestwich. Negative Effects of Modeling Techniques on Search Performance. *Annals of Operations Research* vol. 118, Kluwer Academic Publishers, 2003, pp. 137-150.

[8] S. D. Prestwich. Local Search on SAT-Encoded Colouring Problems. *Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science* vol. 2919, Springer, 2004, pp. 105–119.

[9] B. Selman, H. Levesque, D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *Tenth National Conference on Artificial Intelligence*, MIT Press, 1992, pp. 440–446.

[10] B. Selman, H. A. Kautz, B. Cohen. Noise Strategies for Improving Local Search. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, AAAI Press, 1994, pp. 337–343.

[11] J. Singer, I. P. Gent, A. Smaill. Backbone Fragility and the Local Search Cost Peak. *Journal of Artificial Intelligence Research* vol. 12, 2000, pp. 235–270.

[12] M. Yokoo. Why Adding More Constraints Makes a Problem Easier for Hill-climbing Algorithms: Analyzing Landscapes of CSPs. *Third International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 1330, Springer-Verlag 1997, pp. 356–370.

## A Experimental results

The results shown in Table 1 were obtained on a 733 MHz Pentium II. We consider only the individual instances, not the large families of problems. We performed one run per instance, with a threshold of $10^8$ local moves (*flips*) and the following algorithm parameter settings: $p = 0.05$, $s = 0.01$ and $c = 0.05/n$ where $n$ is the number of variables in the model. (In further experiments with more time and finer tuning we found a 20-solution for SP2, a 35-solution for SP3, and a 54-solution for SP4.)

| File | No. of runs | Objective value | | flips | secs |
|---|---|---|---|---|---|
| | | Best | Worst | | |
| Miller19 | 1 | 13 | 13 | 168279 | 0.5 |
| GP1 | 1 | 45 | 45 | 3216551 | 31 |
| GP2 | 1 | 40 | 40 | 148422 | 2.4 |
| GP3 | 1 | 40 | 40 | 1039508 | 7.2 |
| GP4 | 1 | 30 | 30 | 1031501 | 15 |
| GP5 | 1 | 95 | 95 | 3336715 | 74 |
| GP6 | 1 | 76 | 76 | 5152962 | 117 |
| GP7 | 1 | 79 | 79 | 1996366 | 41 |
| GP8 | 1 | 60 | 60 | 4702041 | 92 |
| NWRS1 | 1 | 3 | 3 | 27767 | 0.06 |
| NWRS2 | 1 | 4 | 4 | 4540 | 0.02 |
| NWRS3 | 1 | 7 | 7 | 148432 | 0.3 |
| NWRS4 | 1 | 7 | 7 | 21939 | 0.09 |
| NWRS5 | 1 | 12 | 12 | 39657 | 0.6 |
| NWRS6 | 1 | 12 | 12 | 44284 | 0.2 |
| NWRS7 | 1 | 10 | 10 | 201125 | 1.0 |
| NWRS8 | 1 | 16 | 16 | 256892 | 1.2 |
| SP1 | 1 | 9 | 9 | 155477 | 0.36 |
| SP2 | 1 | 21 | 21 | 3569961 | 15 |
| SP3 | 1 | 37 | 37 | 65277580 | 458 |
| SP4 | 1 | 57 | 57 | 57188506 | 547 |

Table 1: Results on individual instances

# A Constraint Programming Approach to the Min-Stack Problem

**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 $0-1$) $\alpha_{ij}$ 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) $\alpha_i$ 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 $c_j \in \{1 \ldots m\}$ 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 $\alpha$. 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 $\langle p_1, \ldots, p_n \rangle$. 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 $\alpha$, 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 $\alpha$.

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 |

# Modelling Challenge: Benchmark Results

**Helmut Simonis**
IC-Parc, Imperial College London
hs@icparc.ic.ac.uk

## Abstract

In this note we present a model for the constraint modelling challenge 2005 based on mixed finite domain and continuous variables implemented using the IC library of ECLiPSe. We are interested in proving optimality of our solutions, and therefore choose a model which minimizes the width of the search tree to be explored.The model is naive in that it does not use any deep theoretical results on path width to check consistency.

## 1 Introduction

We present a model for the open stack problem [Fink and Voss, 1999] posed as the first constraint modelling challenge [Gent and Smith, 2005]. Instead of just introducing the model used for our evaluation, we try to explain the steps which led to the selection of our method, and why we found other approaches less satisfactory.

We paraphrase the problem definition from [Gent and Smith, 2005] as follows: We consider a problem with $n$ product types and $m$ orders. All products of the same type are made consecutively, but we can arrange the order of the product types freely. A product $i$ is required by order $j$ if the 0/1 integer entry $c_{ij}$ is equal to one. A stack for an order is required (is *open*) from the time the first product of the order is made to the time the last product is made (inclusive). The objective is to minimize the maximal number of open stacks.

## 2 Basic Model

The basic model of the problem is given by the following definitions and constraints. Let $P$ be the set of production time slots, i.e. $P_i$ denotes the time point when product $i$ is made. Initially, we assume integer time points from 1 to $n$. The sets of variables $S$ and $E$ denote the start and finish time points for the orders, i.e. $S_j$ ($E_j$) is the time point when production of order $j$ starts (ends). These variables also range from 1 to $n$. For each time point $i$ and order $j$ we have a binary variable $O_{ij}$ which denotes whether order $j$ is open at time $i$. The variable $U_i$ denotes the number of open orders at time point $i$, and the overall objective is to minimize the maximal utilization $Limit$. We also use the constants $c_{ij}$ which indicate whether order $j$ requires product $i$. The model is shown in table 3.

This model can be implemented easily with most finite domain solvers, and can be combined with a search routine which assigns the $P$ variables. By propagation, all other variables are fixed and an optimal solution can be found by a *minimize* or *min_max* branching scheme[Prestwich, 1999].

Note that constraints 11 to 14 implement a form of *cumulative* [Aggoun and Beldiceanu, 1993] constraint, and that the reified constraints over the $O$ variables perform obligatory part reasoning[Simonis *et al.*, 2000]. In many constraint languages it would be better to use a single cumulative constraint of the form[1]

$$\mathrm{cumulative}(S, D, \bar{1}, E, Limit)$$

### 2.1 Redundant constraints

We can strengthen the model by adding a simple redundant constraint, which uses the fact that all products must be assigned to different slots and that we know the number of products required for each order.

$$\forall_{1 \le j \le m} : E_j \ge S_j + (\sum_{1 \le i \le n} c_{ij}) - 1$$

This is a special case of a more generic redundant constraint, which works for all subsets of the products required for an order, as shown in table 4. Note that we don't have to create one constraint for each subset, it suffices to order the products by earliest start (resp. latest end) and to count the number of tasks which are placed later (earlier), as shown in table 11.

## 3 Labeling Choices

The basic model leaves us with different possibilities on how to assign values to the variables. We explore three possible alternatives:

### 3.1 Assigning slots to products

The most common search strategy will try to fix values for the variables (see table 8). At each step, we deterministically

[1] To compute utilization profiles, it is preferable to redefine the end variables by adding one, to make them consistent with duration variables $D$.

choose a variable to be assigned next based on some heuristic, and then non-deterministically try out all possible values, until all variables are assigned. Due to the *alldifferent* constraint we know that a trivial bound on the size of the search space is $n!$, we have $n$ choices for the first variable, then $n-1$ for the second variable, and so on, until the last variable has only one value left.

### 3.2 Assigning products to slots

An alternative is to assign slots to variables, e.g. to select a slot in the schedule, and then to try out all possible products for this slot. We can for example (table 9) assign the slots left to right, by selecting the product for the left-most slot first (this means assigning it to value 1), then choosing a product for the second slot, and so on. The trivial search tree size bound again is $n!$, the first choice explores $n$ alternatives.

### 3.3 Partial ordering of products

A third alternative is not to instantiate the variables directly, but to partially order them by enforcing inequality constraints between them. This is similar to the technique used in [Dincbas *et al.*, 1990] to solve disjunctive scheduling problems. It is also related to MIP models for this problem, which typically use $n^2$ 0/1 integer variables stating that one product is made before another. If we decide the order of all pairs of products consistently, we impose an order on the tasks which instantiates the variables. Unfortunately, this search strategy does not mix well with the constraint model, we only achieve very limited constraint propagation until most tasks are ordered. We could perhaps improve the interaction by using constraint handling rules (CHR)[Frühwirth, 1998], but choose not to follow that route.

## 4 Problems with model

The reified constraint model does not remove all inconsistent values, as we will show on two examples. For both examples we consider 5 products, 5 orders and a limit of 4 open stacks. Table 5 shows the first instance. If $P_2$ is assigned to 2 and $P_4$ assigned to 4, then $P_3$ can no longer be assigned to 3. As the stacks for orders $O_4$ and $O_5$ are already open, there is no room for the additional three stacks required by product $P_3$. Table 6 shows a different situation. If $P_2$ and $P_3$ are already assigned to 2 and 3 resp., then placing $P_4$ to 4 is no longer possible because this would require 5 stacks at time point 3.

The problem is caused by the interaction of multiple orders when we fix a product to a time slot. The reified constraint (but also the cumulative formulation) ignore this interaction and therefore fail to detect the inconsistency. We can solve these problems by implementing a stronger propagation for a combined cumulative constraint, as shown in section 5.

A more fundamental problem is the shape of the search tree. If we use a variable assignment method, we have to pick the first product and assign a slot to it. The choice of the slot will be rather uninformed, and for a proof of optimality we will have to explore all alternatives. For the second variable, we will still need to explore $n-1$ possible values. Unless our estimation of the cost is very good, we will need to explore a very large search tree even if we only consider the first $k$ variables. A proof of optimality by enumeration seems very unlikely even for modest problem sizes.

The same argument holds for the slot assignment, the situation is even worse in this case. We have to try all products for the selected slot, but many of them will be non critical. This means that the cost bound will only increase slowly, so that we will have to explore more of the search tree.

Introducing a partial ordering between products seems to be a good strategy. Unfortunately, we can not use the original model to do this, as the added inequality constraints between products do not interact sufficiently with the other constraints. We need another model.

## 5 A combined cumulative global constraint

Due to time limitations, we could not fully develop the constraint, but we did build a simple variant which performs the following pruning (table 12). At each invocation, we build a profile of open orders at each time point. As starting point we use the $S$ and $E$ variables. As soon as the upper bound of the start is less or equal to the lower bound of the end variable for an order, we know that we have an obligatory part for this order. We add up all obligatory parts and obtain a profile of overall stack usage. If the number of open stacks exceeds the cost limit, we can fail. Otherwise, we check for each unassigned product and each value left in its domain, whether it would fit in the space left in the profile. If this is not possible we prune the value from the domain. This solves the problem from table 5.

If the product can be placed in a given location, we then check how this would modify the profile. We can update the profile by comparing the slot value considered with the start and end variables for the orders belonging to the product. We then check if the generated profile exceeds the cost bound, in which case we again can prune the value from the domain. This solves the problem of table 6.

This constraint together with the redundant constraints (16) and (17) drastically reduces the number of choices explored. Unfortunately, this is not enough to prove optimality even for medium sized instances.

## 6 IC Model

The key idea for our new model is the concept of a partial order between products. If we arrange the products in a sequence, we can calculate the start and end times of the orders, and compute the number of open stacks at each time point. Instead of using values 1 to $n$ for the variables, we use an arbitrary real interval as their domain. The first two variables can be placed arbitrarily in the interval together with two sentinels at the beginning and at the end of the interval. This also eliminates the back-to-front symmetry in the problem. In our assignment routine (table 10 shows a simplified form), we decide between which already placed products we should place the selected product. As value we fix the middle of the interval. If the initial interval is sufficiently large, we can always find a number between two already selected values, even when using floating point arithmetic. We remove the alldifferent constraint from the model, and satisfy its condition by construction of the search routine. When we have

obtained a solution, we can renumber the products in order of their assigned values to the range 1 to $n$.

Note that it would be very difficult to obtain this splitting behaviour with integer domains. We would have to use a very large initial domain to make sure that there are enough integer values left between any two already assigned variables to fit in all remaining variables. If we are not extremely careful, we might loose completeness of the search method.

The new model is given in table 7. This model has been implemented using the IC library of the ECLiPSe [Wallace *et al.*, 1997][Cheadle *et al.*, 2003] system which allows a mix of discrete and continuous variables.

Note that the search tree of this model has a very nice form. The first two variables can be fixed arbitrarily, for the third variable we have three possible choices, for the next one four and so on. The tree is quite narrow at the top, but we have to explore more choices the deeper we go in the tree. If we select the initial products carefully, we can hope to limit our search to the top part, exploring relatively few choices. The cumulative constraint has been ported to the continuous domain, but we found that the shaving technique described below is performing slightly more pruning.

# 7 Search Strategy

For our search method (called cresha (credit+shaving)), we have to define a variable selection routine and a value ordering. The variable selection decides which variable to assign next, the value ordering defines the order in which different values are tested during the search.

## 7.1 Variable selection

We use an initial static ordering and complement this by a dynamic selection which is computed at every step of the search. We found experimentally that the following function is quite successful as a static order:

$$w_i = \sum_{1 \le j \le m} c_{ij} * \frac{1}{\sum_{1 \le k \le n} c_{kj}}$$

The contributing value of an order is inversely proportional to the number of products that are required by the order.

As the dynamic part we evaluate at each step the minimal cost increase that would be caused by selecting a variable and placing it in the schedule. We choose the variable which causes the maximal increase of the cost. For ties we use the "domain size", the number of slots where the product can be placed.

## 7.2 Value ordering

As value ordering heuristic we choose the positions in order of minimal increase of the cost function. We break ties by checking how many $S$ and $E$ variables will be updated by a given assignment value. In order to calculate these strategies we have to test each remaining value for each unassigned variable at every step. We use this quite expensive operation at the same time to remove inconsistent values at each step and to achieve a form of global forward checking. This is a shaving technique[Torres and Lopez, 2000] applied over the complete domain.

# 8 Problem Reduction

We will now discuss some additional techniques to reduce the number of nodes that need to be explored. Ideally, these methods would be used to create new, reduced problem instances with fewer variables and constraints. For this evaluation we did apply the reductions at runtime.

## 8.1 Lower Bounds

A trivial lower bound on the cost is given by the maximal number of orders for some product.

$$\text{Limit} \ge \max_{1 \le i \le n} \sum_{1 \le j \le m} c_{ij}$$

We can also obtain lower bounds by looking at subsets of the products and considering all permutations in which they can be arranged. We use subsets of 3 to 5 products. Instead of testing all such subsets, which we deem too expensive, we only use the first 7 products according to our weight function. These bounds can sometimes significantly increase the lower bound, and so help to avoid some optimality proofs by enumeration.

## 8.2 Preprocessing

There are a number of other preprocessing steps that can reduce problem size significantly, so that optimality proofs become much simpler.

**Subsumed Product**

Product $k$ is subsumed by product $i$ if

$$\{j | 1 \le j \le m \wedge c_{kj} = 1\} \subseteq \{j | 1 \le j \le m \wedge c_{ij} = 1\}$$

We can remove the subsumed product $k$ from the problem and schedule it directly before or after product $i$. It is easy to see that this does not change the cost of the schedule.

**Singleton Product**

If a product is required by a single order and that order needs no other products, then we have a singleton product. Formally, we can define the set of singleton products as

$$\{i | \sum_{1 \le j \le m} (c_{ij} * \sum_{1 \le k \le n} c_{kj}) = 1\}$$

We can schedule such products at the very beginning of the sequence without impact on the overall cost. This reduction seems to occur quite frequently in randomly generated problems.

**Problem Decomposition**

More generally, the problem can be decomposed into multiple subproblems if the graph induced by the matrix $c_{ij}$ separates into multiple connected components [Cormen *et al.*, 2001]. The problems can be solved independently and can be combined by piecing together the sub product sequences. Experiments on the problem data have shows that this occurs on a number of instances, but since these problems were easily solved anyway, we did not implement the decomposition.

## 9 Incomplete Search

As initially indicated in [Van Hentenryck and Carillon, 1988], it can be advantageous to consider two different search routines when solving a constraint optimization problem, e.g. to provide an incomplete routine to heuristically find a good solution quickly, and a complete routine which explores the search space efficiently. We also use these two types of routines, the incomplete one is based on partial search, the complete routine is limited by a timeout of 600 seconds.

The initial choices in our search are very important, but quite uninformed. Later in the search the insertion points are constrained by the previously placed products and the search is effectively guided. We use a partial search technique called *credit-based search*[Beldiceanu *et al.*, 1997] to explore the top of the search tree completely, while controlling the overall effort expended. We also tried to use a constraint-based, large neighborhood search routine to locally improve results, but could not complete the tests in the given time frame. The form of the neighborhood in the continuous model seems to be very interesting for this type of local search.

## 10 Result tables

All experiments were run on Linux machines with ECLiPSe 5.8[2]. The overview of the results is given in table 1. It shows the problem set (Set) with author, strategy used, number of products (Pr), number of customers (Cu) and number of instances (In), the percentage of instances solved to optimality (Opt), information about the lower bound (LB), the number of solutions found (NrS) and the best solution (Sol). We then show the number of assignment steps (Ass), the time (Time) and the number of backtracking steps (BT) required to find the best solution (Best) and to prove optimality (Optimal). We give the average (Avg), geometric mean (Geom), median (Median), and minimal (Min) and maximal (Max) values where appropriate. The values for optimality are computed from those instances for which optimality was proved (by reaching the lower bound or by exhaustive search).

The results for individual problems are shown in table 2. Values in italics show that the optimality proof timed out. Solution values in parenthesis give better solutions which were found with other search strategies. The time out for the GP100 and SP sets (marked by *) was increased from 600 to 3600 seconds so that the search routine could find a first solution. Two instances in the NWRS (marked with +) set were modified by hand to remove illegal empty rows.

## 11 Summary

In this note we have presented our model for the constraint modelling challenge. We use a mixed integer and continuous variable model with the IC library of the ECLiPSe language. The schedule is generated by defining an order of the products on the real number axis, inserting new tasks between already placed ones. This model leads to a nice search tree which is rather narrow at the top, allowing proofs of optimality by enumeration for medium sized problems.

[2]The IC library currently does not support holes in continuous domains, we therefore had to simulate this in the user code

## References


[Aggoun and Beldiceanu, 1993] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, pages 57–73, 1993.

[Beldiceanu *et al.*, 1997] N. Beldiceanu, E. Bourreau, P. Chan, and D. Rivreau. Partial search strategy in CHIP. In *2nd Int. Conf. on Meta-Heuristics*, 1997.

[Cheadle *et al.*, 2003] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. ECLiPSe: An introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College London, 2003.

[Cormen *et al.*, 2001] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[Dincbas *et al.*, 1990] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *J. Log. Program.*, 8(1):75–93, 1990.

[Fink and Voss, 1999] A. Fink and S. Voss. Applications of modern heuristic search methods to pattern sequencing problems. *Computers and Operations Research*, 26:17–34, 1999.

[Frühwirth, 1998] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37, October 1998.

[Gent and Smith, 2005] I. Gent and B. Smith. Constraint modelling challenge 2005. http://www.dcs.st-and.ac.uk/ ipg/challenge/, 2005.

[Prestwich, 1999] S. Prestwich. Three CLP implementations of branch-and-bound optimization. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, volume 2. Nova Science Publishers, Inc, 1999.

[Simonis *et al.*, 2000] H. Simonis, A. Aggoun, N. Beldiceanu, and E. Bourreau. Complex constraint abstraction: Global constraint visualisation. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 299–317. Springer, 2000.

[Torres and Lopez, 2000] P. Torres and P. Lopez. Overview and possible extensions of shaving techniques for job-shop problems. In *2nd International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'2000)*, pages 181–186, March 2000.

[Van Hentenryck and Carillon, 1988] P. Van Hentenryck and J.P. Carillon. Generality versus specificity: An experience with AI and OR techniques. In *AAAI*, pages 660–664, 1988.

[Wallace *et al.*, 1997] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12(1), May 1997.

# A Results

| K | Set | Opt | Key | LB | NrS | Sol | Best Ass | Best Time | Best BT | Optimal Ass | Optimal Time | Optimal BT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | hs | 100.00 | Avg | 7.9273 | 1.23 | 8.0309 | 80.43 | 0.23 | 0.98 | 93.58 | 0.24 | 2.85 |
| | cresha | | Geom | 7.6572 | 1.16 | 7.7657 | 0.00 | 0.17 | - | 0.00 | 0.18 | - |
| | Pr 10 | | Median | 8 | 1 | 8 | 70 | 0.28 | 0 | 70 | 0.28 | 0 |
| | Cu 10 | | Min | 3 | 1 | 3 | 0 | 0.01 | 0 | 0 | 0.01 | 0 |
| | In 550 | | Max | 10 | 3 | 10 | 434 | 0.53 | 29 | 736 | 0.73 | 59 |
| 2 | hs | 100.00 | Avg | 15.0673 | 1.57 | 15.8782 | 249.84 | 0.80 | 5.99 | 452.71 | 1.03 | 25.26 |
| | cresha | | Geom | 14.5358 | 1.44 | 15.4085 | 225.36 | 0.78 | - | 347.66 | 0.96 | - |
| | Pr 10 | | Median | 16 | 1 | 17 | 200 | 0.81 | 0 | 339 | 0.91 | 18 |
| | Cu 20 | | Min | 6 | 1 | 7 | 44 | 0.19 | 0 | 44 | 0.20 | 0 |
| | In 550 | | Max | 20 | 5 | 20 | 1507 | 1.76 | 129 | 12798 | 12.94 | 930 |
| 3 | hs | 100.00 | Avg | 22.1255 | 1.86 | 23.9527 | 398.88 | 1.42 | 17.28 | 1137.75 | 2.57 | 76.65 |
| | cresha | | Geom | 21.2858 | 1.67 | 23.2988 | 319.28 | 1.36 | - | 784.75 | 2.24 | - |
| | Pr 10 | | Median | 23 | 2 | 25 | 279 | 1.31 | 7 | 755 | 2.12 | 51 |
| | Cu 30 | | Min | 8 | 1 | 9 | 104 | 0.67 | 0 | 104 | 0.68 | 0 |
| | In 550 | | Max | 30 | 5 | 30 | 6427 | 9.79 | 495 | 12103 | 14.92 | 1006 |
| 4 | hs | 100.00 | Avg | 8.5018 | 1.26 | 8.9218 | 294.51 | 0.52 | 3.02 | 661.48 | 0.86 | 30.58 |
| | cresha | | Geom | 8.2994 | 1.18 | 8.7720 | 193.89 | 0.45 | - | 253.30 | 0.56 | - |
| | Pr 20 | | Median | 9 | 1 | 10 | 244 | 0.50 | 0 | 264 | 0.56 | 0 |
| | Cu 10 | | Min | 4 | 1 | 4 | 4 | 0.03 | 0 | 4 | 0.04 | 0 |
| | In 550 | | Max | 10 | 4 | 10 | 5506 | 4.30 | 230 | 60994 | 38.16 | 5233 |
| 5 | hs | 100.00 | Avg | 12.0836 | 1.45 | 12.8691 | 539.23 | 0.93 | 8.34 | 1167.92 | 1.59 | 44.54 |
| | cresha | | Geom | 11.7352 | 1.32 | 12.6003 | 428.92 | 0.87 | - | 620.55 | 1.16 | - |
| | Pr 15 | | Median | 13 | 1 | 14 | 429 | 0.86 | 0 | 573 | 1.02 | 16 |
| | Cu 15 | | Min | 5 | 1 | 5 | 4 | 0.04 | 0 | 4 | 0.04 | 0 |
| | In 550 | | Max | 15 | 4 | 15 | 7674 | 7.31 | 458 | 67312 | 62.83 | 2540 |
| 6 | hs | 99.55 | Avg | 22.9318 | 1.98 | 25.9682 | 5241.24 | 5.29 | 242.91 | 22116.69 | 21.63 | 1039.18 |
| | cresha | | Geom | 22.1306 | 1.73 | 25.5447 | 1486.45 | 2.02 | - | 6300.03 | 7.93 | - |
| | Pr 15 | | Median | 24 | 2 | 28 | 942 | 1.23 | 8 | 6357 | 7.21 | 253 |
| | Cu 30 | | Min | 9 | 1 | 14 | 429 | 0.79 | 0 | 456 | 0.96 | 0 |
| | In 220 | | Max | 30 | 6 | 30 | 278348 | 241.17 | 17507 | 551805 | 419.51 | 29328 |
| 7 | hs | 97.27 | Avg | 12.7545 | 1.40 | 14.0318 | 5979.43 | 4.37 | 165.01 | 19475.19 | 16.58 | 534.86 |
| | cresha | | Geom | 12.4854 | 1.27 | 13.9369 | 2188.66 | 1.76 | - | 4466.01 | 4.08 | - |
| | Pr 30 | | Median | 14 | 1 | 15 | 2254 | 1.67 | 0 | 3125 | 2.60 | 35 |
| | Cu 15 | | Min | 6 | 1 | 9 | 104 | 0.29 | 0 | 104 | 0.29 | 0 |
| | In 220 | | Max | 15 | 4 | 15 | 610546 | 417.07 | 28594 | 617707 | 424.54 | 28793 |
| 8 | hs | 98.64 | Avg | 16.0727 | 1.65 | 17.9773 | 6451.40 | 4.96 | 206.78 | 34178.23 | 26.92 | 1253.65 |
| | cresha | | Geom | 15.5813 | 1.45 | 17.7574 | 2007.04 | 1.80 | - | 5382.35 | 5.08 | - |
| | Pr 20 | | Median | 17 | 1 | 19 | 1500 | 1.34 | 0 | 3353 | 3.46 | 72 |
| | Cu 20 | | Min | 6 | 1 | 10 | 340 | 0.51 | 0 | 340 | 0.51 | 0 |
| | In 220 | | Max | 20 | 5 | 20 | 255303 | 198.54 | 13773 | 875385 | 554.73 | 49653 |
| 9 | hs | 54.55 | Avg | 30.6636 | 1.86 | 36.5909 | 17882.52 | 46.22 | 528.12 | 31674.18 | 104.19 | 921.68 |
| | cresha | | Geom | 29.5883 | 1.60 | 36.3154 | 4006.93 | 13.63 | - | 11200.40 | 45.82 | - |
| | Pr 20 | | Median | 33 | 1 | 39 | 1500 | 7.13 | 0 | 10187 | 46.04 | 262 |
| | Cu 40 | | Min | 15 | 1 | 24 | 1292 | 4.36 | 0 | 1292 | 6.59 | 0 |
| | In 110 | | Max | 40 | 5 | 40 | 175430 | 491.06 | 5630 | 183800 | 460.27 | 6318 |
| 10 | hs | 50.00 | Avg | 23.9000 | 1.41 | 28.6273 | 16223.35 | 28.73 | 237.74 | 22480.24 | 74.09 | 315.60 |
| | cresha | | Geom | 23.2144 | 1.29 | 28.5375 | 7015.01 | 14.48 | - | 11303.57 | 30.56 | - |
| | Pr 30 | | Median | 25 | 1 | 30 | 4900 | 10.52 | 0 | 7740 | 18.54 | 80 |
| | Cu 30 | | Min | 12 | 1 | 22 | 3224 | 5.29 | 0 | 4004 | 9.07 | 0 |
| | In 110 | | Max | 30 | 4 | 30 | 306566 | 447.20 | 8642 | 156794 | 560.15 | 2631 |
| 11 | wbo | 100.00 | Avg | 5.2250 | 1.60 | 5.9250 | 257.10 | 0.37 | 4.97 | 539.40 | 0.55 | 29.15 |
| | cresha | | Geom | 4.8237 | 1.46 | 5.4219 | 235.03 | 0.36 | - | 350.93 | 0.45 | - |
| | Pr 10 | | Median | 5 | 1 | 6 | 200 | 0.38 | 0 | 293 | 0.43 | 9 |
| | Cu 10 | | Min | 2 | 1 | 2 | 104 | 0.20 | 0 | 104 | 0.20 | 0 |
| | In 40 | | Max | 8 | 3 | 10 | 938 | 0.76 | 53 | 5533 | 3.16 | 439 |
| 12 | wbop | 100.00 | Avg | 5.1250 | 1.52 | 6.7500 | 245.18 | 0.37 | 4.38 | 1563.65 | 1.14 | 108.10 |
| | cresha | | Geom | 4.5496 | 1.43 | 6.1393 | 231.98 | 0.36 | - | 777.71 | 0.77 | - |
| | Pr 10 | | Median | 6 | 2 | 8 | 200 | 0.37 | 1 | 752 | 0.72 | 44 |
| | Cu 10 | | Min | 2 | 1 | 3 | 147 | 0.23 | 0 | 147 | 0.25 | 0 |

Table 1: Summary (cont'd)

| | | | | | | | Best | | | Optimal | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | Set | Opt | Key | LB | NrS | Sol | Ass | Time | BT | Ass | Time | BT |
| | In 40 | | Max | 8 | 3 | 10 | 757 | 0.70 | 42 | 7476 | 4.39 | 762 |
| 13 | wbp | 100.00 | Avg | 6.9500 | 1.27 | 7.2750 | 93.33 | 0.27 | 0.65 | 149.50 | 0.31 | 7.95 |
| | cresha | | Geom | 6.4930 | 1.21 | 6.7652 | 83.14 | 0.24 | - | 104.77 | 0.26 | - |
| | Pr 10 | | Median | 8 | 1 | 8 | 91 | 0.31 | 0 | 102 | 0.34 | 1 |
| | Cu 10 | | Min | 3 | 1 | 3 | 25 | 0.04 | 0 | 25 | 0.05 | 0 |
| | In 40 | | Max | 10 | 2 | 10 | 200 | 0.42 | 6 | 978 | 0.71 | 83 |
| 14 | wbo | 100.00 | Avg | 10.7571 | 2.04 | 12.9000 | 343.30 | 0.83 | 13.53 | 1363.89 | 1.98 | 90.07 |
| | cresha | | Geom | 10.1709 | 1.86 | 12.3087 | 313.62 | 0.80 | - | 1084.27 | 1.74 | - |
| | Pr 10 | | Median | 11 | 2 | 14 | 288 | 0.81 | 13 | 1127 | 1.76 | 71 |
| | Cu 20 | | Min | 5 | 1 | 6 | 200 | 0.47 | 0 | 300 | 0.59 | 11 |
| | In 70 | | Max | 17 | 4 | 19 | 1061 | 1.60 | 51 | 5788 | 6.04 | 456 |
| 15 | wbop | 100.00 | Avg | 10.6250 | 2.15 | 14.2750 | 731.58 | 1.23 | 45.45 | 6285.73 | 6.90 | 468.02 |
| | cresha | | Geom | 9.7734 | 1.93 | 13.3796 | 438.94 | 1.00 | - | 3262.00 | 4.16 | - |
| | Pr 10 | | Median | 12 | 2 | 16 | 393 | 0.83 | 26 | 3527 | 4.47 | 230 |
| | Cu 20 | | Min | 5 | 1 | 6 | 200 | 0.51 | 0 | 261 | 0.57 | 21 |
| | In 40 | | Max | 16 | 4 | 20 | 4348 | 4.99 | 337 | 50179 | 52.63 | 3739 |
| 16 | wbp | 100.00 | Avg | 13.3250 | 1.75 | 15.1250 | 284.88 | 0.78 | 10.38 | 1403.13 | 1.83 | 104.60 |
| | cresha | | Geom | 12.6977 | 1.58 | 14.5042 | 235.47 | 0.75 | - | 746.04 | 1.38 | - |
| | Pr 10 | | Median | 15 | 2 | 17 | 200 | 0.74 | 1 | 666 | 1.31 | 52 |
| | Cu 20 | | Min | 7 | 1 | 8 | 70 | 0.46 | 0 | 70 | 0.46 | 0 |
| | In 40 | | Max | 19 | 4 | 20 | 1300 | 1.79 | 84 | 15557 | 13.35 | 1343 |
| 17 | wbo | 100.00 | Avg | 16.2700 | 2.40 | 20.0500 | 635.57 | 1.70 | 38.99 | 3786.50 | 6.62 | 271.37 |
| | cresha | | Geom | 15.4637 | 2.18 | 19.3414 | 456.30 | 1.53 | - | 2800.51 | 5.39 | - |
| | Pr 10 | | Median | 16 | 2 | 21 | 392 | 1.33 | 24 | 2989 | 5.53 | 216 |
| | Cu 30 | | Min | 8 | 1 | 10 | 200 | 0.77 | 0 | 489 | 1.25 | 30 |
| | In 100 | | Max | 25 | 5 | 28 | 4454 | 6.57 | 369 | 19978 | 31.42 | 1459 |
| 18 | wbop | 100.00 | Avg | 16.3750 | 2.17 | 22.4750 | 1376.03 | 2.89 | 96.80 | 13750.42 | 22.38 | 979.15 |
| | cresha | | Geom | 15.0397 | 1.91 | 21.5388 | 644.70 | 2.07 | - | 10926.39 | 18.52 | - |
| | Pr 10 | | Median | 18 | 2 | 26 | 516 | 1.38 | 42 | 11414 | 18.98 | 767 |
| | Cu 30 | | Min | 7 | 1 | 13 | 200 | 0.85 | 0 | 3200 | 5.87 | 191 |
| | In 40 | | Max | 26 | 5 | 30 | 9133 | 16.23 | 623 | 45921 | 68.86 | 3328 |
| 19 | wbp | 100.00 | Avg | 19.1750 | 1.98 | 23.1750 | 1015.38 | 2.17 | 66.80 | 6376.65 | 9.90 | 472.10 |
| | cresha | | Geom | 18.3156 | 1.81 | 22.4438 | 512.28 | 1.69 | - | 3478.32 | 6.32 | - |
| | Pr 10 | | Median | 20 | 2 | 25 | 356 | 1.31 | 23 | 3421 | 6.00 | 219 |
| | Cu 30 | | Min | 10 | 1 | 14 | 169 | 0.83 | 0 | 242 | 0.96 | 17 |
| | In 40 | | Max | 27 | 4 | 30 | 6801 | 10.15 | 478 | 39688 | 52.02 | 3346 |
| 20 | wbo | 100.00 | Avg | 6.7667 | 2.05 | 9.3500 | 2888.00 | 1.66 | 98.72 | 22505.97 | 13.09 | 1020.38 |
| | cresha | | Geom | 5.9933 | 1.86 | 8.3463 | 1261.54 | 0.88 | - | 6221.31 | 4.23 | - |
| | Pr 15 | | Median | 7 | 2 | 10 | 934 | 0.65 | 11 | 5818 | 4.31 | 213 |
| | Cu 15 | | Min | 2 | 1 | 3 | 532 | 0.40 | 0 | 532 | 0.40 | 0 |
| | In 60 | | Max | 12 | 4 | 14 | 61010 | 28.86 | 2636 | 313222 | 181.56 | 15347 |
| 21 | wbop | 98.33 | Avg | 6.7667 | 2.22 | 10.3833 | 9726.05 | 6.05 | 428.68 | 131222.66 | 79.05 | 5923.10 |
| | cresha | | Geom | 5.8449 | 1.99 | 9.1998 | 2048.01 | 1.46 | - | 44260.26 | 28.25 | - |
| | Pr 15 | | Median | 7 | 2 | 12 | 967 | 0.72 | 20 | 79058 | 47.09 | 3244 |
| | Cu 15 | | Min | 2 | 1 | 3 | 650 | 0.45 | 0 | 650 | 0.46 | 0 |
| | In 60 | | Max | 12 | 6 | 15 | 133780 | 83.22 | 5952 | 728131 | 395.59 | 35566 |
| 22 | wbp | 100.00 | Avg | 9.4167 | 1.72 | 11.0500 | 702.28 | 0.54 | 17.68 | 7144.43 | 4.31 | 344.42 |
| | cresha | | Geom | 8.6743 | 1.52 | 10.1101 | 507.88 | 0.46 | - | 1835.01 | 1.42 | - |
| | Pr 15 | | Median | 10 | 1 | 12 | 429 | 0.45 | 0 | 1748 | 1.26 | 68 |
| | Cu 15 | | Min | 3 | 1 | 3 | 147 | 0.20 | 0 | 147 | 0.20 | 0 |
| | In 60 | | Max | 14 | 4 | 15 | 6621 | 2.71 | 357 | 130281 | 74.46 | 5531 |
| 23 | wbo | 97.50 | Avg | 5.0750 | 1.93 | 7.3500 | 3812.88 | 1.86 | 110.40 | 37503.54 | 19.01 | 1734.59 |
| | cresha | | Geom | 4.6410 | 1.71 | 6.9127 | 2188.36 | 1.11 | - | 10622.69 | 6.30 | - |
| | Pr 20 | | Median | 5 | 2 | 8 | 1707 | 0.86 | 9 | 7412 | 4.77 | 198 |
| | Cu 10 | | Min | 2 | 1 | 3 | 609 | 0.32 | 0 | 614 | 0.35 | 6 |
| | In 40 | | Max | 8 | 4 | 10 | 41830 | 20.30 | 1829 | 538568 | 184.02 | 36566 |
| 24 | wbop | 70.00 | Avg | 5.1000 | 1.58 | 8.1000 | 32196.22 | 14.65 | 1491.97 | 120702.39 | 63.20 | 4928.00 |
| | cresha | | Geom | 4.5807 | 1.40 | 7.7854 | 2360.09 | 1.18 | - | 28923.86 | 16.42 | - |
| | Pr 20 | | Median | 6 | 1 | 9 | 1500 | 0.75 | 0 | 33405 | 19.57 | 760 |
| | Cu 10 | | Min | 2 | 1 | 4 | 784 | 0.48 | 0 | 1145 | 0.74 | 19 |
| | In 40 | | Max | 8 | 4 | 10 | 628704 | 292.82 | 29222 | 724484 | 420.03 | 33028 |
| 25 | wbp | 100.00 | Avg | 7.7429 | 1.26 | 8.7143 | 427.31 | 0.31 | 3.94 | 6470.49 | 3.42 | 290.49 |

Table 1: Summary (cont'd)

| | | | | | | | Best | | | Optimal | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | Set | Opt | Key | LB | NrS | Sol | Ass | Time | BT | Ass | Time | BT |
| | cresha | | Geom | 7.4901 | 1.19 | 8.5549 | 0.00 | 0.27 | - | 0.00 | 0.49 | - |
| | Pr 20 | | Median | 8 | 1 | 9 | 340 | 0.28 | 0 | 558 | 0.43 | 21 |
| | Cu 10 | | Min | 4 | 1 | 5 | 0 | 0.02 | 0 | 0 | 0.02 | 0 |
| | In 70 | | Max | 10 | 3 | 10 | 3822 | 1.82 | 103 | 372784 | 188.41 | 16960 |
| 26 | wbo | 65.56 | Avg | 8.9111 | 2.50 | 13.8111 | 27438.20 | 35.19 | 1018.08 | 95476.02 | 156.21 | 2898.73 |
| | cresha | | Geom | 7.5116 | 2.21 | 12.2172 | 6307.57 | 9.78 | - | 44176.78 | 74.82 | - |
| | Pr 20 | | Median | 9 | 2 | 16 | 3546 | 5.87 | 62 | 65923 | 112.42 | 1791 |
| | Cu 20 | | Min | 2 | 1 | 3 | 935 | 1.51 | 0 | 986 | 1.73 | 4 |
| | In 90 | | Max | 16 | 6 | 20 | 455615 | 491.18 | 20321 | 295537 | 480.40 | 10359 |
| 27 | wbop | 26.67 | Avg | 8.9667 | 2.29 | 15.3667 | 26961.39 | 42.86 | 899.46 | 132009.00 | 151.20 | 3731.50 |
| | cresha | | Geom | 7.5419 | 1.92 | 13.7934 | 5029.20 | 8.10 | - | 51583.01 | 64.03 | - |
| | Pr 20 | | Median | 9 | 2 | 18 | 2317 | 4.77 | 32 | 73488 | 86.99 | 1921 |
| | Cu 20 | | Min | 2 | 1 | 3 | 1500 | 1.74 | 0 | 2944 | 3.56 | 21 |
| | In 90 | | Max | 16 | 8 | 20 | 347266 | 461.07 | 9908 | 560353 | 591.79 | 15010 |
| 28 | wbp | 85.56 | Avg | 12.3000 | 1.97 | 15.4556 | 8961.77 | 8.61 | 316.50 | 46275.01 | 46.88 | 1701.68 |
| | cresha | | Geom | 11.2104 | 1.73 | 14.1546 | 2607.38 | 3.00 | - | 12892.73 | 15.72 | - |
| | Pr 20 | | Median | 13 | 2 | 18 | 1500 | 1.68 | 5 | 14151 | 17.58 | 453 |
| | Cu 20 | | Min | 4 | 1 | 4 | 264 | 0.54 | 0 | 269 | 0.55 | 1 |
| | In 90 | | Max | 19 | 5 | 20 | 103050 | 90.22 | 4659 | 538036 | 421.26 | 24508 |
| 29 | wbo | 47.50 | Avg | 4.9500 | 1.55 | 8.3750 | 21851.75 | 18.13 | 509.02 | 118364.84 | 112.54 | 3580.84 |
| | cresha | | Geom | 4.5332 | 1.42 | 8.1421 | 7479.11 | 5.51 | - | 49665.19 | 55.46 | - |
| | Pr 30 | | Median | 5 | 1 | 9 | 4900 | 3.50 | 0 | 32678 | 37.52 | 496 |
| | Cu 10 | | Min | 2 | 1 | 5 | 2254 | 1.65 | 0 | 12190 | 13.83 | 138 |
| | In 40 | | Max | 8 | 4 | 10 | 270313 | 281.90 | 5969 | 859851 | 554.87 | 35187 |
| 30 | wbop | 35.00 | Avg | 4.9500 | 1.38 | 8.7500 | 23221.67 | 24.69 | 667.50 | 46058.71 | 66.34 | 1149.50 |
| | cresha | | Geom | 4.5429 | 1.28 | 8.6002 | 6204.28 | 5.59 | - | 21772.23 | 30.85 | - |
| | Pr 30 | | Median | 5 | 1 | 9 | 4437 | 4.29 | 0 | 20610 | 28.26 | 244 |
| | Cu 10 | | Min | 2 | 1 | 5 | 1607 | 1.72 | 0 | 5271 | 7.82 | 81 |
| | In 40 | | Max | 8 | 3 | 10 | 320114 | 410.74 | 11346 | 334300 | 430.15 | 11710 |
| 31 | wbp | 100.00 | Avg | 8.2200 | 1.21 | 9.3100 | 2344.85 | 2.06 | 69.77 | 4950.81 | 5.15 | 197.65 |
| | cresha | | Geom | 8.0651 | 1.15 | 9.2406 | 0.00 | 0.66 | - | 0.00 | 1.44 | - |
| | Pr 30 | | Median | 9 | 1 | 10 | 650 | 0.65 | 0 | 1152 | 1.16 | 26 |
| | Cu 10 | | Min | 4 | 1 | 6 | 0 | 0.04 | 0 | 0 | 0.04 | 0 |
| | In 100 | | Max | 10 | 3 | 10 | 164794 | 135.56 | 6471 | 173891 | 146.47 | 6719 |
| 32 | wbo | 21.67 | Avg | 6.5667 | 1.87 | 11.7833 | 37849.58 | 39.61 | 598.22 | 177203.38 | 228.80 | 2909.15 |
| | cresha | | Geom | 5.7231 | 1.67 | 11.2386 | 10358.27 | 9.36 | - | 110578.67 | 145.07 | - |
| | Pr 30 | | Median | 7 | 2 | 13 | 6281 | 4.94 | 2 | 142211 | 184.55 | 2122 |
| | Cu 15 | | Min | 2 | 1 | 4 | 3224 | 2.94 | 0 | 9500 | 11.34 | 158 |
| | In 60 | | Max | 11 | 5 | 15 | 374739 | 403.44 | 6460 | 443254 | 526.85 | 7583 |
| 33 | wbop | 10.00 | Avg | 6.6000 | 1.40 | 12.8167 | 12780.73 | 12.32 | 239.20 | 184943.67 | 260.64 | 2937.17 |
| | cresha | | Geom | 5.7376 | 1.28 | 12.4216 | 5859.46 | 5.21 | - | 153923.29 | 209.77 | - |
| | Pr 30 | | Median | 7 | 1 | 14 | 4900 | 4.41 | 0 | 229296 | 259.25 | 3464 |
| | Cu 15 | | Min | 2 | 1 | 6 | 2875 | 3.01 | 0 | 76419 | 86.36 | 1096 |
| | In 60 | | Max | 12 | 4 | 15 | 240524 | 250.97 | 7120 | 373416 | 533.51 | 6141 |
| 34 | wbp | 81.67 | Avg | 10.5750 | 1.58 | 13.1250 | 9835.32 | 10.13 | 267.42 | 47043.44 | 53.21 | 1414.51 |
| | cresha | | Geom | 10.1160 | 1.43 | 12.8301 | 3028.60 | 2.88 | - | 11943.57 | 13.76 | - |
| | Pr 30 | | Median | 11 | 1 | 14 | 2552 | 2.18 | 0 | 10488 | 13.22 | 199 |
| | Cu 15 | | Min | 4 | 1 | 7 | 387 | 0.77 | 0 | 463 | 0.92 | 0 |
| | In 120 | | Max | 15 | 4 | 15 | 433111 | 472.16 | 15024 | 478405 | 552.62 | 16340 |
| 35 | wbo | 86.67 | Avg | 14.5667 | 3.02 | 20.9833 | 17578.88 | 22.45 | 928.18 | 102742.73 | 139.78 | 4688.42 |
| | cresha | | Geom | 13.1834 | 2.78 | 19.7089 | 3699.86 | 5.91 | - | 50765.00 | 74.55 | - |
| | Pr 15 | | Median | 15 | 3 | 23 | 2230 | 3.47 | 77 | 60638 | 83.89 | 2542 |
| | Cu 30 | | Min | 5 | 1 | 7 | 650 | 1.40 | 0 | 3290 | 6.19 | 98 |
| | In 120 | | Max | 25 | 6 | 29 | 236502 | 277.56 | 14682 | 421350 | 548.72 | 22956 |
| 36 | wbop | 23.33 | Avg | 13.5167 | 2.67 | 22.5333 | 40699.43 | 71.71 | 2082.72 | 136558.21 | 275.94 | 5676.07 |
| | cresha | | Geom | 11.9061 | 2.32 | 21.1790 | 5566.81 | 16.99 | - | 93146.43 | 208.80 | - |
| | Pr 15 | | Median | 15 | 2 | 25 | 4221 | 12.11 | 195 | 101430 | 275.82 | 4659 |
| | Cu 30 | | Min | 4 | 1 | 8 | 650 | 1.42 | 0 | 9106 | 32.59 | 373 |
| | In 60 | | Max | 23 | 6 | 30 | 395895 | 582.38 | 21580 | 390955 | 599.35 | 15182 |
| 37 | wbp | 91.67 | Avg | 17.6333 | 2.63 | 23.0000 | 30253.63 | 37.39 | 1332.52 | 103805.02 | 133.66 | 5003.25 |
| | cresha | | Geom | 16.2903 | 2.33 | 21.7611 | 4148.78 | 6.59 | - | 44501.15 | 61.64 | - |
| | Pr 15 | | Median | 18 | 3 | 26 | 2283 | 3.61 | 83 | 69103 | 95.81 | 2982 |

Table 1: Summary (cont'd)

| | | | | | | | Best | | | Optimal | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | Set | Opt | Key | LB | NrS | Sol | Ass | Time | BT | Ass | Time | BT |
| | Cu 30 | | Min | 6 | 1 | 9 | 532 | 1.18 | 0 | 1900 | 3.09 | 82 |
| | In 60 | | Max | 27 | 6 | 30 | 471584 | 591.02 | 16879 | 554472 | 568.48 | 39190 |
| 38 | wbo | 6.43 | Avg | 12.6500 | 2.23 | 23.6357 | 33127.47 | 58.95 | 631.96 | 29021.11 | 81.22 | 368.33 |
| | cresha | | Geom | 10.2571 | 1.91 | 21.2772 | 12092.61 | 21.44 | - | 16607.19 | 43.11 | - |
| | Pr 30 | | Median | 13 | 2 | 27 | 7651 | 12.85 | 22 | 13804 | 35.02 | 92 |
| | Cu 30 | | Min | 2 | 1 | 3 | 4437 | 7.88 | 0 | 7571 | 15.68 | 11 |
| | In 140 | | Max | 24 | 5 | 30 | 333331 | 587.51 | 16381 | 146344 | 408.97 | 2251 |
| 39 | wbop | 2.14 | Avg | 12.6571 | 1.65 | 25.4000 | 12522.08 | 30.68 | 164.41 | 65826.00 | 251.47 | 1044.33 |
| | cresha | | Geom | 10.2150 | 1.45 | 24.0354 | 7621.25 | 18.60 | - | 63787.06 | 238.13 | - |
| | Pr 30 | | Median | 13 | 1 | 29 | 4900 | 13.42 | 0 | 68981 | 296.76 | 1105 |
| | Cu 30 | | Min | 2 | 1 | 3 | 4437 | 8.48 | 0 | 45132 | 146.01 | 625 |
| | In 140 | | Max | 24 | 6 | 30 | 199529 | 434.47 | 5786 | 83365 | 311.64 | 1403 |
| 40 | wbp | 27.14 | Avg | 17.7857 | 1.80 | 25.1786 | 17071.12 | 38.98 | 294.91 | 62200.24 | 145.16 | 1093.74 |
| | cresha | | Geom | 15.8865 | 1.54 | 23.2492 | 7114.11 | 16.75 | - | 25298.37 | 70.54 | - |
| | Pr 30 | | Median | 19 | 1 | 29 | 4900 | 12.30 | 0 | 35496 | 100.70 | 627 |
| | Cu 30 | | Min | 4 | 1 | 4 | 1295 | 4.81 | 0 | 1295 | 4.83 | 2 |
| | In 140 | | Max | 28 | 7 | 30 | 295597 | 459.45 | 8148 | 322076 | 588.20 | 5967 |
| 41 | shaw | 92.00 | Avg | 9.4800 | 2.44 | 13.6800 | 32579.40 | 25.84 | 861.36 | 61404.17 | 48.46 | 2442.83 |
| | cresha | | Geom | 9.4329 | 2.12 | 13.6598 | 3241.43 | 2.74 | - | 23347.72 | 20.15 | - |
| | Pr 20 | | Median | 9 | 2 | 14 | 2149 | 1.75 | 14 | 20660 | 18.44 | 611 |
| | Cu 20 | | Min | 8 | 1 | 12 | 1104 | 1.07 | 0 | 3255 | 2.63 | 43 |
| | In 25 | | Max | 11 | 5 | 15 | 717909 | 570.25 | 19249 | 603810 | 453.77 | 29028 |

Table 1: Summary

| | | | | | | | Best | | | Opt | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Problem | N | M | Den | LB | NrSol | Sol | Ass | BT | Time | Ass | BT | Time |
| Miller.1 | 40 | 20 | 0.20 | 4 | 1 | 14 (13) | 11400 | 0 | 11.98 | *662304* | *6935* | *600.14* |
| GP50.1 | 50 | 50 | 0.81 | 45 | 1 | 45 | 13160 | 0 | 67.74 | 13160 | 0 | 67.77 |
| GP50.2 | 50 | 50 | 0.63 | 40 | 1 | 40 | 19504 | 0 | 81.03 | 19504 | 0 | 81.07 |
| GP50.3 | 50 | 50 | 0.65 | 40 | 2 | 40 | 25765 | 1 | 96.04 | 25765 | 1 | 96.07 |
| GP50.4 | 50 | 50 | 0.54 | 30 | 1 | 30 | 9065 | 0 | 48.39 | 9065 | 0 | 48.43 |
| GP100.1 | 100 | 100 | 0.86 | 95 | 1 | 95 | 171500 | 0 | 3291.27 | 171500 | 0 | 3291.38* |
| GP100.2 | 100 | 100 | 0.68 | 75 | 1 | 75 | 171500 | 0 | 3137.39 | 171500 | 0 | 3137.51* |
| GP100.3 | 100 | 100 | 0.67 | 75 | 2 | 75 | 182940 | 1 | 2892.09 | 182940 | 1 | 2892.22* |
| GP100.4 | 100 | 100 | 0.54 | 60 | 2 | 60 | 157157 | 1 | 2495.84 | 157157 | 1 | 2495.97* |
| NWRS.1 | 20 | 10 | 0.23 | 3 | 2 | 3 | 142 | 7 | 0.19 | 142 | 7 | 0.19 |
| NWRS.2 | 20 | 10 | 0.27 | 4 | 2 | 4 | 391 | 5 | 0.33 | 391 | 5 | 0.33 |
| NWRS.3 | 25 | 15 | 0.24 | 7 | 3 | 7 | 677 | 36 | 0.73 | 677 | 36 | 0.74 |
| NWRS.4 | 25 | 15 | 0.27 | 7 | 2 | 7 | 848 | 9 | 0.82 | 848 | 9 | 0.82 |
| NRWS.1 | 30 | 20 | 0.26 | 11 | 3 | 12 | 2139 | 6 | 2.57 | 2481 | 32 | 2.95 |
| NRWS.2 | 30 | 20 | 0.27 | 11 | 3 | 12 | 3525 | 2 | 3.45 | 3810 | 22 | 3.86 |
| NRWS.3 | 59+ | 25 | 0.15 | 10 | 5 | 10 | 11202 | 369 | 26.50 | 11202 | 369 | 26.53 |
| NRWS.4 | 59+ | 25 | 0.18 | 12 | 5 | 16 | 27312 | 352 | 39.86 | 151272 | 2081 | 294.36 |
| SP.1 | 25 | 25 | 0.10 | 8 | 3 | 9 | 1472 | 6 | 1.89 | 1553 | 11 | 2.04* |
| SP.2 | 50 | 50 | 0.07 | 9 | 1 | 22 | 15092 | 0 | 48.89 | *887511* | *4641* | *3600.39** |
| SP.3 | 75 | 75 | 0.06 | 9 | 1 | 46 (42) | 64680 | 0 | 304.12 | *748912* | *2571* | *3600.58** |
| SP.4 | 100 | 100 | 0.05 | 13 | 5 | 67 | 437413 | 566 | 2658.04 | *506807* | *793* | *3601.20** |

Table 2: Individual Results

## B Models and Code Segments

$$
\begin{array}{lll}
\min & \text{Limit} \quad s.t. & (1)\\
& P[1..n] :: 1..n & (2)\\
& S[1..m] :: 1..n & (3)\\
& E[1..m] :: 1..n & (4)\\
& U[1..n] :: 1..m & (5)\\
& Limit :: 1..m & (6)\\
& O[1..n, 1..m] :: 0..1 & (7)\\
& \text{alldifferent}(P) & (8)\\
& \forall_{1\leq j\leq m} : S_j = \min\{P_i | c_{ij} = 1\} & (9)\\
& \forall_{1\leq j\leq m} : E_j = \max\{P_i | c_{ij} = 1\} & (10)\\
& \forall_{1\leq i\leq n}\forall_{1\leq j\leq m} : & (11)\\
& \quad O_{ij} = (S_j \leq i) \wedge (i \leq E_j) & (12)\\
& \forall_{1\leq i\leq n} : U_i = \sum_{1\leq j\leq m} O_{ij} & (13)\\
& \text{Limit} = \max_{1\leq i\leq n} U_i & (14)
\end{array}
$$

Table 3: Basic Model

$$
\begin{array}{lll}
\forall_{1\leq j\leq m} & \forall PP \subseteq \{P_i | c_{ij} = 1\} : & (15)\\
E_j & \geq \min(PP) + |PP| - 1 & (16)\\
S_j & \leq \max(PP) - |PP| + 1 & (17)
\end{array}
$$

Table 4: Redundant Constraint

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| $O_1$ | 1 | 0 | 1 | 0 | 0 |
| $O_2$ | 0 | 0 | 1 | 0 | 1 |
| $O_3$ | 0 | 0 | 1 | 0 | 1 |
| $O_4$ | 1 | 1 | 0 | 1 | 0 |
| $O_5$ | 0 | 1 | 0 | 1 | 0 |

Table 5: Example 1

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| $O_1$ | 1 | 1 | 0 | 1 | 0 |
| $O_2$ | 0 | 1 | 0 | 1 | 1 |
| $O_3$ | 0 | 0 | 1 | 1 | 0 |
| $O_4$ | 0 | 0 | 1 | 0 | 0 |
| $O_5$ | 0 | 0 | 1 | 0 | 0 |

Table 6: Example 2

$$
\begin{array}{lll}
\min & \text{Limit} \quad s.t. & (18)\\
& P[1..n] :: 0.0..1000.0 & (19)\\
& S[1..m] :: 0.0..1000.0 & (20)\\
& E[1..m] :: 0.0..1000.0 & (21)\\
& U[1..n] :: 1..m & (22)\\
& Limit :: 1..m & (23)\\
& O[1..n, 1..m] :: 0..1 & (24)\\
& \forall_{1\leq j\leq m} : S_j = \min\{P_i | c_{ij} = 1\} & (25)\\
& \forall_{1\leq j\leq m} : E_j = \max\{P_i | c_{ij} = 1\} & (26)\\
& \forall_{1\leq i\leq n}\forall_{1\leq j\leq m} : & (27)\\
& \quad O_{ij} = (S_j \leq P_i) \wedge (P_i \leq E_j) & (28)\\
& \quad c_{ij} = 1 \Rightarrow O_{ij} = 1 & (29)\\
& \forall_{1\leq i\leq n} : U_i = \sum_{1\leq j\leq m} O_{ij} & (30)\\
& \text{Limit} = \max_{1\leq i\leq n} U_i & (31)
\end{array}
$$

Table 7: IC Model

```
labeling([]).
labeling([H|T]):-
  select_var(X,[H|T],Rest,strategy),
  choose_val(X),
  labeling(Rest).
```

Table 8: Assignment Method

```
labeling(L):-
  left_to_right(L,1).

left_to_right([],_).
left_to_right([H|T],N):-
  delete(N,[H|T],Rest),
  N1 is N+1,
  left_to_right(Rest,N1).
```

Table 9: Left to Right Assignment

```
labeling([]).
labeling([P1]):-
  P1 = 200.0.
labeling([P1,P2|Rest]):-
  P1 = 200.0,
  P2 = 800.0,
  insert_lp([0.0,P1,P2,1000.0],Rest).

insert_lp(Current,[]).
insert_lp(Current,[H|T]):-
  select_var(X,[H|T],Rest),
  insert(Current,X,New),
  insert_lp(New,Rest).

insert([A,B|R],X,[A,X,B|R]):-
  X is (A+B)/2.
insert([A|R],X,[A|S]):-
  insert(R,X,S).
```

Table 10: Insertion Routine

```
spread(E,Orders):-
  (foreach(Ex,E),
   foreach(Set,Orders) do
      prep_set(Set,SetN,MinSet),
      sort(0,=<,MinSet,MinSorted),
      (foreach(V,MinSorted),
       for(J,SetN,1,-1),
       fromto(0,A,A1,BoundE) do
         A1 is max(A,V+J-1)
      ),
      Ex #>= BoundE
  ).
```

Table 11: Spread Redundant Constraint

```
cumul(Assign,S,E,Limit):-
  store_obligatory_parts(S,E,Obl),
  create_events(Obl,Events),
  sort(time of event,=<,Events,Sorted),
  build_profile(Sorted,Profile),
  check_resource(Profile,MaxUse),
  Limit #>= MaxUse,
  find_unassigned_vars(Assign,Unass),
  get_max(Limit,LimitMax),
  (foreach(X,Unass),
   param(Obl,Sorted,LimitMax) do
     get_assign_domain(X,DomList),
     (foreach(V,DomList),
      param(X,Obl,Sorted,LimitMax) do
        addit_events(X,V,Obl,NEvents),
        append(NEvents,Sorted,NAll),
        sort(t of event,=<,NAll,NSorted),
        build_profile(NSorted,NProfile),
        check_resource(NProfile,NUse),
        (NUse =< LimitMax ->
           true
        ;
           remove_value(X,V)
        )
     )
  ).
```

Table 12: Combined Cumulative Constraint

# Modelling Challenge - Open Stack Problem

**Radoslaw Szymanek and Mark Hennessy**
Cork Constraint Computation Centre
Cork, Ireland
{radsz}{m.hennessy}@4c.ucc.ie

## Abstract

The Open Stack minimization problem exhibits many features which lead to an interesting CP model for study. In this short paper, we will describe the main ideas behind one possible modeling approach. The experimental results show that our approach can solve the Open Stack instances proposed for this modeling challenge. It is also able to produce proof of optimality within a given search cut-off limit for a large number of instances.

## 1 Introduction

A manufacturer has a number of orders from a customer to satisfy; Once a customer's order is started (i.e. any product in the order has been made) a stack is created for that customer. When all products required by a customer have been made, its stack is closed and the order is sent to the customer. Because of limited space in the production area, the number of stacks that are in use simultaneously (i.e. the number of customer orders that are in simultaneous production), should be minimized. This problem and possible local search approaches are presented in detail in [Fink and Voss, 1999].

Our model consists of five different viewpoints which are connected using channeling constraints. The model also contains implied constraints, symmetry breaking constraints, dominance constraints and a specially designed global constraint. For the purposes of clarity the variables in this paper are represented with a lower case letter and a single or double index while entities such as customers and products are represented with an upper case letter.

## 2 Example

We will use example depicted in Figure 1 to explain our model. It consists of five products and four customers. The order matrix $o_{ij}$ specifies what products are ordered by which customers. For example, product $P_2$ is ordered by customer $C_4$ only. The current ordering of products makes the number of required stacks equal to the number of customers. However, if $P_5$ is swapped with $P_2$ then it is possible to decrease the number of required stacks by one, which is the optimal number of stacks for this example.

```
    P1  P2  P3  P4  P5              P1  P5  P3  P4  P2
C1   1   0   0   0   1               1   1   0   0   0
C2   1   0   1   1   0   ----\       1   0   1   1   0
C3   0   0   1   1   0   ----/       0   0   1   1   0
C4   0   1   0   1   1               0   1   0   1   1
```

Figure 1: An example and its optimal solution

## 3 Viewpoints and Variables

A graphical representation of all viewpoints is depicted in Figure 2. The first basic viewpoint looks at the positions of products. The problem is described by variables $p_i$, which represent the positions of products. Knowledge about positions of products is sufficient to determine the number of required stacks. For our example, there are five variables $p_1$, $p_2$, $p_3$, $p_4$, and $p_5$. Each $p_i$ variable has a domain consisting of $p$ values, where $p$ is the number of products.

The second viewpoint looks at the problem from a customer perspective, however it still uses $p_i$ variables. The stack for $C_i$ is opened at position $s_i$ and is closed at position $e_i$. In addition, we can introduce variable $d_i$ which represents the difference between a closing position and an opening position. Initially, the domains of all variables will have $p$ values.

The third viewpoint considers the customer positions. Variable $c_i$ defines the position at which $C_i$ is serviced. In our example, there are four variables $c_1$, $c_2$, $c_3$, and $c_4$. Each variable will initially have a domain consisting of $c$ values, where $c$ is the number of customers. The positions of products and the number of required stacks can be decided, given the positions for all customers. However, in our approach we also use the $cp$ matrix from the next viewpoint to improve the bound on the number of stacks required given a partial solution.

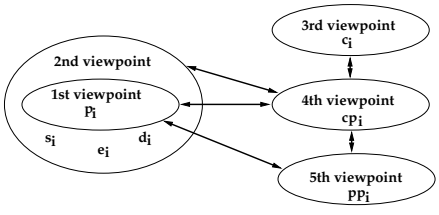



Figure 2: Graphical representation of different viewpoints

| | C1 | C2 | C3 | C4 |
|---|---|---|---|---|
| C1 | 1 | 1 | 0..1 | 1 |
| C2 | 1 | 1 | 1 | 1 |
| C3 | 0..1 | 1 | 1 | 1 |
| C4 | 1 | 1 | 1 | 1 |

Figure 3: Customer precedence matrix for the example

The fourth viewpoint decides the relative positions of customer stacks. A boolean variable $cp_{ij}$ from customer precedence matrix specifies if stack for $C_i$ is **not** closed before stack for $C_j$ is opened. In our example, the optimal solution makes variable $cp_{13}$ equal to zero since the stack for $C_1$ is closed before stack for $C_3$ is open. If customers $C_i$ and $C_j$ overlap then both $cp_{ij}$ and $cp_{ji}$ are equal one. Please note that any customers that share products will overlap. The $cp$ matrix for our example is depicted in Figure 3. This viewpoint works very well if there are many ones in the order matrix.

The fifth viewpoint looks at the relative positions of products. A boolean variable $pp_{ij}$ from the product precedence matrix specifies that $P_i$ is positioned **after** $P_j$. In our example, the left solution will assign value zero to $pp_{25}$ since $P_2$ is not positioned after $P_5$. On the other hand, the right solution will assign value one to $pp_{25}$ since $P_2$ is positioned after $P_5$. Each $pp_{ii}$ is equal to one.

## 4 Constraints

This section presents the constraints that are required to guarantee the correctness of solutions for different viewpoints. It does not include implied constraints nor channeling constraints. The simplest viewpoint is the product position one. It imposes an $alldifferent$ constraint to make all product positions unique.

The second viewpoint requires constraints to express relations between products of $C_i$ and $s_i$, $e_i$, and $d_i$. First, $s_i$ is equal to the minimal position of any product of $C_i$. Second, $e_i$ is equal to the maximal position of any product of $C_i$. Finally, $d_i$ is larger or equal to the number of products of $C_i$ minus one.

The customer position (3rd) viewpoint requires an $alldifferent$ constraint to enforce that each customer is positioned differently. This constraint will put any two customers at different positions even if they have the same set of products.

The fourth viewpoint takes into account the precedence relations between customers. It requires constraints which make sure that there is no pair of customers $C_i$ and $C_j$ such that $C_i$ is closed before $C_j$ and vice verse. Given any pair of variables $cp_{ij}$ and $cp_{ji}$ only one can be equal to zero.

The fifth viewpoint models problem in terms of product precedence relations. The constraints imposed by this viewpoint make sure that for every pair of products $P_i$ and $P_j$ variables $pp_{ij}$ and $pp_{ji}$ are not equal. This prevents a situation when $P_i$ is positioned after $P_j$ and at the same time $P_j$ is positioned after $P_i$.

## 5 Implied Constraints

There are additional constraints which may improve propagation within different viewpoints. For example, it is possible to add implied constraints on $s_i$ and $e_i$ variables after analysis of order matrix $o$. For any pair of customers $C_i$ and $C_j$, which require the same set of products, we can impose implied constraint which forces $s_i$ to be equal to $s_j$ and $e_i$ be equal to $e_j$. If $C_i$ and $C_j$ differ in only one product then the weaker implied constraint is imposed which enforces that $s_i$ is equal to $s_j$ or $e_i$ is equal to $e_j$. On the other hand, if two customers $C_i$ and $C_j$ do not share any products then $s_i$ has to be different from $s_j$ and $e_i$ has to be different from $e_j$.

We also add implied constraints which are imposed over the customer precedence variables. For any pair of customers $C_i$ and $C_j$ which do not share any products, if there exists a customer $C_m$ which is in parallel to $C_j$ and $C_i$ is closed before $C_m$ then $C_j$ can not be closed before $C_i$. It is also possible to add different types of implied constraints for the same viewpoint which check any triplet of customers $C_i$, $C_j$, and $C_m$. If $cp_{ij} = 0$ and $cp_{jm} = 0$ then it implies that $C_i$ is closed before $C_m$, so $cp_{im} = 0$. However, the number of constraints of this type grows quickly with the increase of customers, which often renders those constraints of little value if any. In order to reduce the size of the model we do not include these constraints in a model.

Since the second viewpoint extends the first viewpoint, the following constraints should be rather called implied constraints than channeling constraints. If $s_i$ is equal to $s_j$ then any $P_m$ which does not belong to both customers can not be placed at position equal to $s_i$. Similar constraints can be imposed if we substitute $s$ with $e$. However, we did not include these implied constraints in our model since they increased search time.

## 6 Channeling Constraints

The most important channeling constraints are the ones between the fourth and the fifth viewpoint. They are expressed as $reified$ constraints which take as input boolean variable $cp_{ij}$ and the disjunction of constraints of type $pp_{mv} = 1$, where $P_m$ is any product of $C_i$ and $P_v$ is any product of $C_j$. In other words, it means that $C_i$ can not be closed before $C_j$ if there is at least one product of $C_i$ which is positioned later than any product of $C_j$. These channeling constraints make it possible to reason about the precedence relationship between products implied by the customer precedence relationship and vice verse.

The channeling constraint between the first and the fifth viewpoint are expressed using a $sum$ constraint. Variable $p_i$ is equal to the sum of variables in $i$th row of $pp$ matrix. On the other hand, the channeling constraints between the second viewpoint and the fourth viewpoint are expressed as $reified$ constraints which take as arguments constraint $s_j \leq e_i$ and boolean variable $cp_{ij}$.

There are also single direction channeling constraints which are imposed to improve propagation between models for different viewpoints. For example, $cp_{ij} = 0$ implies that $c_i < c_j$. In addition, $c_i < c_j$ implies that $e_i \leq e_j$. On the other hand, $e_i < e_j$ implies that $c_i < c_j$. Since one product can close more than one open stack, the channeling constraints are expressed as implication constraints.

## 7 Dominance Constraints

There are number of possible dominance constraints which are imposed. The dominance constraint prevents exploration of the search space which contains a wasteful solution if there is a guarantee that there is a better or equally good solution in different part of the search space. We refer the reader to [Prestwich and Beck, 2004] for more elaborate explanation of dominance constraints. In the case of the Open Stack problem, if for some special conditions the existence of the solution with the larger size of open stacks indicates the existence of the solution with a smaller number of open stacks then we can immediately cut the search space which contains the wasteful solution.

The first dominance rule imposes additional constraints on product positions. If the set of customers of $P_i$ is a proper subset of the set of customers of $P_j$ then $P_i$ can always be safely positioned before $P_j$. In this case, constraint $pp_{ij} = 0$ is imposed. In our example, $P_3$ would have a smaller position than $P_4$. We have not included this dominance rule in our model since we are not certain that it does not conflict with other dominance rules. In addition, these constraints prolonged search on instances we have tested.

The second dominance rule imposes additional constraints on customer positions. If the set of products of $C_i$ is a proper subset of the set of products of $C_j$ then $C_i$ is positioned earlier than $C_j$. It is expressed using constraints $c_i < c_j$. In our example, this dominance rule will position $C_1$ before $C_2$.

The third dominance rule is based on the customer neighborhood. The neighborhood of $C_i$ is defined as the set of customers for which a variable from the $i$th row of the $cp$ matrix equals one. That is, $C_i$ has a neighbor $C_m$ if $cp_{im} = 1$. If there is a pair of customers $C_i$ and $C_j$ such that for all $k$, $cp_{ik} \leq cp_{jk}$ and there is a $C_m$ for which $cp_{im} < cp_{jm}$ then $C_i$ is positioned earlier than $C_j$. In other words, $C_i$ is positioned before $C_j$ if the neighborhood of $C_i$ is a proper subset of the neighborhood of $C_j$. For efficiency reasons, a constraint to discover this dominance rule is imposed only for pairs of $C_i$ and $C_j$ where there exists a $C_m$, such that $cp_{im}$ is not fixed and $cp_{jm} = 1$ before search.

## 8 Symmetry breaking

The products of the first customer can be ordered in any way. They could be assigned based on lexicographical ordering of the products. The symmetry breaking will choose one possible ordering and enforce it. However, this symmetry breaking can sometimes conflict with a product dominance rule, which is explained in section 10.

The first three dominance constraints are not imposed in case when sets under consideration are the same. However, in such case we could apply symmetry breaking and still impose dominance like constraints but only for pairs when $i < j$ given $C_i$ and $C_j$ or $P_i$ and $P_j$. In other words, dominance constraints could be strengthen by removing requirement for proper subset if $i < j$. Applying similar approach to the third dominance rule is of little practical use since constraints to detect and enforce this dominance rule are expensive in terms of time and memory. We observed the increase of the search time when we included symmetry breaking to strengthen the third dominance rule.

## 9 Special global constraint

We implemented special global constraint to obtain better lower-bound estimate given partial solution. It involves $cp$, $c$ and $limit$ variable which denotes the maximal number of open stacks. We do do not describe the consistency algorithm in full detailed. Due to space limitation a non incremental version of the algorithm was presented in Algorithm 1. This algorithm presents reasoning for the minimal number of open stacks based on $cp$ and $c$ matrix only.

**Algorithmus 1** Consistency function for the proposed global constraint

```
1:  for i = 0 to c do
2:    lastPosition[i] = 0; minNeighbors[i] = 0;
3:    for j = 0 to c do
4:      if cp_ji.min() = 0 then
5:        lastPosition[i]++;
6:      end if
7:      if cp_ij.min() = 1 then
8:        minNeighbors[i]++;
9:      end if
10:   end for
11: end for
12: {open[i] - boolean value, initially false since C_i'th customer is not open}
13: for i = 0 to c do
14:   {i - currently considered position}
15:   for j = 0 to c do
16:     if lastPosition[j] ≤ i then
17:       open[j] = true; { C_j becomes open since already its last possible position is analyzed}
18:     end if
19:   end for
20:   lowerbound = - i + 1; {number of closed customers}
21:   if only C_m can be placed at position i then
22:     for j = 0 to c do
23:       if cp_mj.min() = 1 then
24:         open[j] = true; {C_m opens C_j}
25:       end if
26:     end for
27:   else
28:     compute the minimal additional open customers (minAdd) given possible customers at position i
29:     lowerbound = lowerbound + minAdd;
30:   end if
31:   for j = 0 to c do
32:     if open[j] = true then
33:       lowerbound++;
34:     end if
35:   end for
36:   if lowerbound < minNeighbors[i]-i then
37:     lowerbound = minNeighbors[i] - i;
38:   end if
39:   use lowerbound to update number of open stacks
40: end for
```

The lines from 1 to 11 compute the last possible position at which $C_i$ can be opened and the minimal number of customers open if $C_i$ is open. Given $C_i$ the number of zero's

in the $i$th column of the $cp$ matrix gives the latest possible position at which $C_i$ is opened. On the other hand, the number of ones in the $i$th row specifies the number of customers which are (were) open if $C_i$ is open. The main loop, which starts at line 13, computes the lowerbound of required stacks taking into account every position separately. The for loop in lines 15 to 19 opens a customer if its last possible open position is equal to current position $i$. If it is known that $C_m$ takes position $i$ then $m$th row of $cp$ matrix is used to update open array. If it is not known which customer takes position $i$ then all candidates are examined and the candidate which will open the smallest number of customers is used to compute the lowerbound. Line 20 initiates the lowerbound with value $i-1$ since this reflects the number of customers which are already closed at position $i$. The lines 32 to 35 simply count the number of previously or currently open customers at position $i$. The lowerbound for number of openstacks can prune the domain of openstack variable or detect inconsistency when partial solution exceeds the number of allowed open stacks.

## 10 Model discussion

The complexity of the model depends on the number of customers and number of products. The number of variables and constraints grow quadratically with the number of customers and products. The model uses many different viewpoints therefore channeling constraints are a significant part of the model. The third and fourth viewpoint require $c^2$ and $p^2$ variables respectively. However, those variables are required to reason directly about the precedence constraints occurring in the partial solution. The model uses few global constraints, like alldifferent and the special global constraint which is presented in section 9. They help to improve the reasoning, especially in the case when proving the optimality of the solution. The dominance rules can make some instances easy to prove since they indicate the search space which can be omitted. The model uses also some symmetry breaking to complement the dominance rules.

While it is often beneficial to add dominance and symmetry breaking constraints to a model, in the case of the open stack problem there can be a conflict between product dominance constraints and symmetry breaking for the products of the first customer. Consider only products $P_3$ and $P_4$ and only customers $C_3$ and $C_4$ from the example. The customer dominance rule will enforce that $c_4 < c_3$. The product dominance rule will enforce that $p_3 < p_4$. Finally, the symmetry breaking for the products of the first customer enforce that $p_4 = 1$ since it is the product of $C_4$, which is the first customer. Clearly, we can not have all three techniques applied at the same time. The decision which technique should be discarded is instance dependent. In our case, due to time limitations we decided to discard the symmetry breaking as it applies only to products of the first customer which can potentially have an impact.

## 11 Search

The search approach is based on standard variable and value ordering heuristic. The variable ordering heuristic uses forward min domain as the first criteria and forward degree as the second criteria. The ties are resolved based on lexicographical ordering. The search variables consists of variables from $cp$ matrix, $p$ variables, $s$ variables, and finally $e$ variables in such order. In addition variables within $cp$ matrix are ordered. The rows of the $cp$ matrix that contain more unfixed variables are considered first. Please note that search does not use $c$ variables as the position of the customer is decided by the $cp$ matrix. Similarly, the $pp$ matrix is not included since $p$ variables denote the product positions. The $c$ and $pp$ variables are only used to improve the reasoning and discover inconsistent partial solutions faster. The value ordering chooses always the minimal possible value within a variable domain. This value ordering will first choose the value zero for any precedence variable, which prefers the situation when a customer is closed before another one is open. This often leads to good quality first solution. In order to be certain that our global constraint is complete, we run additional search with all variables.

## 12 Experimental setup

We measure the search effort in time and number of backtracks. We use JaCoP solver [Kuchcinski, 2003], which we augmented with additional global constraint. The experimental results are presented in the appendix using the suggested data format.

## 13 Conclusions and Future Work

The presented model for the open stack problem uses many modeling techniques. It models the problem using multiple viewpoints, channeling constraints, dominance constraints, implied constraints, symmetry breaking, and a specially designed global constraint. Future work will evaluate the influence of the particular model components on search. For example, the removal of the second viewpoint could reduce search time on average. The choice between conflicting dominance rules and symmetry breaking constraints can be done on a instance basis. In addition, different search algorithms based on instance characteristics could be proposed.

## Acknowledgments

We would like to thank Nic Wilson for interesting discussions.

## References

[Fink and Voss, 1999] A. Fink and S. Voss. Applications of modern heuristic search methods to pattern sequencing problems. *Computers and Operations Research*, 26:17–34, 1999.

[Kuchcinski, 2003] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, July 2003.

[Prestwich and Beck, 2004] Steven Prestwich and J. Christopher Beck. Exploiting dominance in three symmetric problems. In *Proc. 4th Int. Workshop on Symmetry & Constraint Satisfaction Problems*, 2004.

| File | proven optimal | best value | time to best solution | backtracks to best solution | time to prove | backtracks to prove |
|---|---|---|---|---|---|---|
| Miller19 | Yes | 13 | 1.5 | 67 | 926 | 77154 |
| GP1 | Yes | 45 | 2.88 | 149 | 3.64 | 185 |
| GP2 | Yes | 40 | 3.58 | 81 | 5.62 | 170 |
| GP3 | Yes | 40 | 72 | 1552 | 72 | 1552 |
| GP4 | Yes | 30 | 677r | 13055 | 683r | 13066 |
| GP5 | Yes | 95 | 27.5 | 325 | 28.7 | 338 |
| GP6 | Yes | 75 | 71.4 | 311 | 71.4 | 325 |
| GP7 | Yes | 75 | 57.52 | 437 | 57.52 | 437 |
| GP8 | No | 86 | 343.65 | 177 | - | - |
| NWRS1 | Yes | 3 | 0.84 | 49 | 0.84 | 49 |
| NWRS2 | Yes | 4 | 0.78 | 26 | 0.78 | 26 |
| NWRS3 | Yes | 7 | 0.97 | 34 | 1.03 | 39 |
| NWRS4 | Yes | 7 | 1.14 | 107 | 1.14 | 107 |
| NWRS5 | Yes | 12 | 1.24 | 109 | 1.28 | 118 |
| NWRS6 | Yes | 12 | 1.18 | 49 | 1.21 | 50 |
| NWRS7 | Yes | 10 | 2.88 | 112 | 11.13 | 450 |
| NWRS8 | Yes | 16 | 2.35 | 118 | 8.62 | 396 |
| SP1 | No | 11 | 96.0 | 21577 | - | - |
| SP2 | No | 22 | 13.6 | 287 | - | - |
| SP3 | No | 51 | 67.5 | 317 | - | - |
| SP4 | No | 71 | 486.35 | 521 | - | - |

Table 1: Times and number of backtracks to prove the optimal solution for each problem

In order to measure the time to find an optimal solution, the solver is given a lowerbound which corresponds to the optimal solution. This allows the solver to start backtracking to the first search node immediately after finding the solution with optimal value. In other words, we have a two-pass experimental setup. The first time we look for an optimal solution with proof of optimality within a given limit of the search effort. The second time we use the cost of the optimal solution to set the lowerbound and measure the time required to find an optimal solution. The values for last three columns of Table 2 are computed only for instances which were proven optimal. We have set the cut-off limit to 100.000 backtracks. The runtimes specify the amount of CPU seconds as given by a time command from Linux. Due to a tight submission deadline we were not able to obtain results for files wbo_30_15, wbo_30_30, wbop_30_30, and wbp_30_30.

| File | % solved | mean value | time (sec) | | | # backtracks to find optimal | | | # backtracks per instance | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | mean | median | max | mean | median | max | mean | median | max |
| problem_10_10.dat | 100 | 8.031 | 0.65 | 0.61 | 4.13 | 17 | 15 | 179 | 48 | 16 | 3809 |
| problem_10_20.dat | 100 | 8.9218 | 0.70 | 0.68 | 1.61 | 29 | 26 | 92 | 36 | 26 | 645 |
| problem_15_15.dat | 100 | 12.869 | 1.75 | 0.70 | 64.84 | 49 | 23 | 7541 | 486 | 24 | 28299 |
| problem_15_30.dat | 100 | 14.018 | 0.98 | 0.82 | 13.23 | 44 | 39 | 180 | 79 | 39 | 3994 |
| problem_20_10.dat | 100 | 15.878 | 7.89 | 0.89 | 261.00 | 1751 | 34 | 57611 | 97 | 19 | 4276 |
| problem_20_20.dat | 98.2 | 17.173 | 11.31 | 0.87 | 441.00 | 34 | 29 | 92 | 2690 | 31 | 96152 |
| problem_30_10.dat | 93.6 | 23.995 | 87.43 | 1.59 | 1305.00 | 455 | 27 | 28419 | 8187 | 94 | 99703 |
| problem_30_15.dat | 83.6 | 26.05 | 34.89 | 1.03 | 831.00 | 133 | 27 | 4964 | 3933 | 28 | 95281 |
| problem_30_30.dat | 92.7 | 28.336 | 8.79 | 1.23 | 404.00 | 41 | 33 | 92 | 836 | 44 | 33673 |
| problem_40_20.dat | 80.9 | 36.573 | 34.37 | 1.42 | 766.00 | 73 | 35 | 2373 | 2487 | 36 | 56064 |
| ShawInstances.txt | 100 | 13.680 | 22.97 | 10.11 | 128.00 | 164 | 70 | 1292 | 5069 | 2235 | 29124 |
| wbo_10_10 | 100 | 5.925 | 0.71 | 0.65 | 1.31 | 26 | 19 | 128 | 79 | 29 | 491 |
| wbo_10_20 | 100 | 7.35 | 0.77 | 0.67 | 1.40 | 45 | 30 | 233 | 65 | 32 | 339 |
| wbo_10_30 | 100 | 8.2 | 0.82 | 0.75 | 1.35 | 46 | 41 | 110 | 59 | 41 | 216 |
| wbo_15_15 | 86.7 | 9.383 | 12.79 | 0.98 | 179.00 | 268 | 45 | 8491 | 5392 | 95 | 82468 |
| wbo_15_30 | 85 | 11.583 | 2.47 | 0.87 | 26.11 | 160 | 41 | 5473 | 440 | 42 | 6815 |
| wbo_20_10 | 97.1 | 12.900 | 21.93 | 3.51 | 260.00 | 232 | 31 | 6637 | 5283 | 822 | 59325 |
| wbo_20_20 | 76.7 | 14.044 | 26.62 | 1.05 | 287.00 | 620 | 36 | 20857 | 5682 | 66 | 67837 |
| wbo_30_10 | 79.0 | 20.050 | 143.39 | 18.92 | 923.00 | 526 | 61 | 11811 | 14462 | 2400 | 86156 |
| wbop_10_10 | 100 | 6.75 | 0.65 | 0.6 | 1.26 | 17 | 14 | 38 | 51 | 14 | 490 |
| wbop_10_20 | 100 | 8.075 | 0.75 | 0.65 | 1.64 | 33 | 26 | 77 | 56 | 27 | 457 |
| wbop_10_30 | 100 | 8.55 | 0.81 | 0.74 | 1.15 | 49 | 40 | 135 | 54 | 40 | 154 |
| wbop_15_15 | 98.3 | 10.367 | 6.03 | 0.85 | 48.80 | 1186 | 22 | 19942 | 2161 | 36 | 19942 |
| wbop_15_30 | 86.7 | 12.183 | 8.40 | 0.8 | 371.00 | 145 | 41 | 4684 | 2037 | 42 | 98515 |
| wbop_20_10 | 97.5 | 14.275 | 24.53 | 0.76 | 236 | 157 | 23 | 3532 | 6417 | 32 | 63041 |
| wbop_20_20 | 78.9 | 15.378 | 12.76 | 0.89 | 216.00 | 164 | 28 | 5146 | 2240 | 31 | 41430 |
| wbop_30_10 | 75.0 | 22.475 | 13.37 | 1.43 | 78.99 | 36 | 21 | 217 | 1219 | 53 | 7370 |
| wbop_30_15 | 66.7 | 22.917 | 31.8 | 1.19 | 222.00 | 59 | 22 | 533 | 2888 | 48 | 21740 |
| wbp_10_10 | 100 | 7.275 | 0.64 | 0.57 | 1.17 | 17 | 13 | 69 | 50 | 16 | 469 |
| wbp_10_20 | 100 | 8.714 | 0.69 | 0.65 | 1.21 | 29 | 26 | 82 | 36 | 27 | 207 |
| wbp_10_30 | 100 | 9.31 | 0.76 | 0.74 | 1.01 | 41 | 40 | 82 | 43 | 41 | 103 |
| wbp_15_15 | 91.667 | 11.05 | 10.289 | 0.71 | 170 | 2912 | 21 | 49800 | 4392 | 25 | 77695 |
| wbp_15_30 | 100 | 13.092 | 4.24 | 0.81 | 79.35 | 336 | 40 | 13866 | 936 | 40 | 22361 |
| wbp_20_10 | 100 | 15.125 | 9.20 | 0.9 | 65.68 | 87 | 17 | 1087 | 2371 | 31 | 17278 |
| wbp_20_20 | 80 | 15.6 | 10.49 | 0.87 | 228.00 | 200 | 28 | 6186 | 1972 | 28 | 47192 |
| wbp_30_10 | 80 | 23.175 | 38.06 | 1.49 | 504 | 1251 | 26 | 25711 | 3902 | 47 | 55423 |
| wbp_30_15 | 66.7 | 23.467 | 13.66 | 1.19 | 116.00 | 94 | 33 | 1302 | 1247 | 43 | 11011 |

Table 2: Times and number of backtracks to find the optimal solution for each data set

# Tearing customers apart for solving PSP-SOS

**Charlotte Truchet**[1], **Jérémie Bourdon**[1], **Philippe Codognet**[2]
[1] LINA, Université de Nantes
2 rue de la Houssinère, BP92208, 44322 Nantes Cedex 03, France
[2] LIP6, Université de Paris 6,
8, rue du Capitaine Scott, 75015 Paris, France
Charlotte.Truchet.95@normalesup.org, jeremie.bourdon@lina.univ-nantes.fr

## Abstract

This article is about modelling and solving issues for a Pattern Sequencing Problem, proposed as the First Constraint Modelling Challenge. PSP-SOS is difficult in that relevant information is not self-contained in the variables' values. A first model, still global but less global, so to say, is proposed. A derived second model, more precisely dedicated to local search methods, is then implemented with the Adaptive Search meta-heuristic.

## 1 Introduction

This paper intends to describe an entry to the First Constraint Modelling Challenge. The goal is to model and solve, with constraint programming techniques, a particular Pattern Sequencing Problem. PSP consist in finding a permutation of some production patterns, optimizing some objective functions dealing with store-house's size or handling costs. The particular PSP of the challenge is known as the simultaneously open stack problem, or PSP-SOS, as stated in [Fink and Voss, 1999]. An expression of PSPs as graph pathwidth problems can be found in [Linhares and Yanasse, 2002].

Let us recall the problem briefly and introduce our notations, see also figure 1. A delivery service has to satisfy the demands of $n$ customers $c_1 ... c_n$. Each of them has ordered a particular subset of some products $p_1 ... p_m$. We will write $s_i$ the order of customer $i$, $s_i \subset \{p_1 ... p_m\}$. The goal is to find the order in which the products should be delivered.

Suppose that the products are ranked by a permutation $\sigma$: it gives a schedule where product $p_{\sigma(1)}$ is the first one delivered, and so on. Most of our notations will depend on the order $\sigma$ so we will not write it. Let $st_i$ (resp $ft_i$) be the starting time (resp finishing) of customer $c_i$ in the schedule $\sigma$. The delivery service uses a stack per customer: this stack is closed before $st_i$, open from $st_i$ to $ft_i$, and closed again after $ft_i$.

The goal is to minimize the maximum number of simultaneously opened stacks, in order to realize the schedule in the smallest storehouse as possible. With our notations, this can be written as:

$$\text{minimize}_\sigma f(\sigma)$$
$$\text{where} f(\sigma) = \{\max_{1 \leq j \leq m} \sharp\{i \leq n, st_i \leq \sigma(j) \leq ft_i\}\}$$

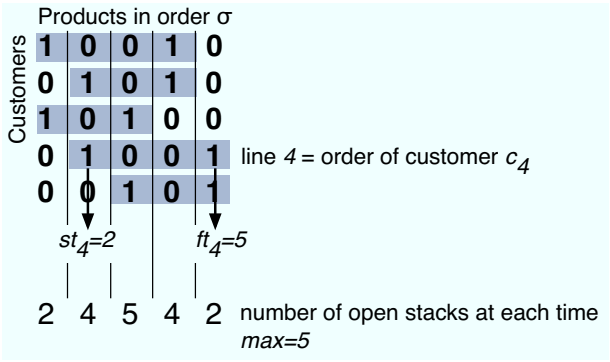



Figure 1: An example of the $Prob$ matrix, with the stacks shown in grey and the number of open stacks for every time unit given at the bottom.

A convenient solution for representing the problem, chosen by the Challenge organizers as their instances' format, is to use a matrix of size $n \times m$, where the $i$-th row, $j$-th column contains a 1 if customer $c_i$ has ordered product $p_j$, and 0 otherwise. In the following, we will write this matrix $Prob$. Line $Prob_i$ represents the order of customer $c_i$.

## 2 Modelling the PSP-SOS as a Constraint Satisfaction Problem

We will discuss in this section some of the issues when modelling PSP-SOS. The goal is to find a model that can be expressed in an existing Constraint Programming Language, then solve it with a classical solver. Several languages and generic solvers exist nowadays, like the Prolog family with for instance GNU-Prolog with constraints [Diaz and Codognet, 2001], CHIP [Aggoun and Beldiceanu, 1991], Localizer and Localizer++ [Michel and Hentenryck, 2001], just to name a few. Their pros and cons in terms of expressivity, efficiency or genericity could be discussed for hours.

Let us very briefly recall that a distinction is made between the complete methods, able of giving a proof of optimality, and the incomplete ones which have been proved very efficient in practice, but are unable to prove optimality.

Whatever language and solver are chosen, PSP-SOS has some features which make it challenging to program. Mainly,

it gives an objective function to minimize that is expressed as a maximum on some of the problem's data. Moreover, the variables are either subsets of $\{p_1...p_m\}$ or sequences of $0$ and $1$ in the matrix model, anyway the cost not only depend on their values, but on their values within the configuration: a $0$ in a column can mean that the stack is closed, or open, depending on the before and after values. So to say, the problem structure is such that the relevant information is not self-contained.

This appears as a major issue for solving PSP-SOS. With complete method, filtering techniques are very efficient when applied locally. Here we have the analoguous of a global constraint and they are well known to be difficult to deal with. With incomplete methods, it leads to other issues that will be discussed below.

## 2.1 Intrinsic issues of the PSP-SOS

As a first point, an obvious remark is that PSP problems are a matter of finding an order on the products. A model will anyhow have to include an alldifferent constraint on the $p_j$. Thus a reasonnable choice is to state the maximum open stacks problem as a permutation problem on the products $p_1...p_m$. Then, a possibility would be to keep the model with a permutation CSP on the $Prob$ matrix columns, and $f$ as objective function. Anyway, it is worth focusing on the problem's structure to understand better how this objective function behaves on the search space.

The second point concerns the customers. Unlikely to many optimization problems, the PSP-SOS objective function ranges over a very small domain of values. The only way to decrease it, is to tear apart two customers in the schedule. Intuitively, thinking about the stacks: the goal is to minimize the number of stacks needed to satisfy the customers' orders, which does well mean that the delivery service wants to reuse as many stacks as possible. The more stacks are re-used, the more likely we are to have a schedule with a minimum number of SOSs.

In the order $\sigma$, we will write $c_{i_1} \ll c_{i_2} \leftrightarrow \forall \alpha \in s_{i_1}, \forall \beta \in s_{i_2}, \sigma(\alpha) < \sigma(\beta)$, that is, customer $c_{i_1}$ has been fully served before customer $c_{i_2}$ begins to be served. In that case they can share a stack and the objective function may decrease of 1, depending on whether those two customers where situated on the columns realizing the maximum or not.

Intuitively, the range of $g$ may be greater than the range of $f$ but they behave in the same way. See example on figure 2. Think of an elementary step from $\sigma$ to $\sigma'$, $g(\sigma) > g(\sigma') \rightarrow f(\sigma) \leq f(\sigma')$ if $g$ decreases, $f$ may increase or not depending on the position where $g$ decreases w.r.t. the maximum. Conversely, $f(\sigma) < f(\sigma') \rightarrow g(\sigma) > g(\sigma')$, except in rare cases. Now our goal will be to maximize $g$.

As a third point, we observe that, whatever the schedule $\sigma$, it is not possible to tear apart two customers who share at least one product. Such two customers can obviously never share a stack and from a resolution point of view, we had best not wasting any search effort on this couple. So it shall be usefull to include this knowledge on the problem's structure in the model.

For this we define the relation $Sep$ on the customers in order to distinguish the customers' couple for which we may be able to decrease the objective function (separable customers), from the ones for which we will certainly not (unseparable ones):

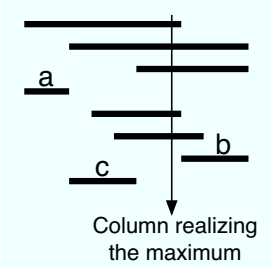



Figure 2: On this example, only the open stacks in the current schedule are represented. Pairs of stacks $(a, b)$ and $(c, b)$ do count in $f$ as they are separated by the column, or time, realizing the maximum of $f$. Pair of stacks $(a, c)$ do count in $g$ but not in $f$.

$$Sep(c_{i_1}, c_{i_2}) \leftrightarrow \forall \alpha \in s_{i_1}, \forall \beta \in s_{i_2}, \alpha \neq \beta$$

With the matrix representation, one has $Sep(c_{i_1}, c_{i_2}) \leftrightarrow \text{Prob}_{i_1} \cdot \text{Prob}_{i_2} = 0$, that is, the scalar product of the two lines is equal to zero. This can be pre-computed once and for all and accessed in constant time, let us write $Sep$ the resulting $n \times n$ boolean matrix.

All those remarks lead to the following model: find a permutation $\sigma$ on the products $p_1...p_m$, such that:

$$\text{maximize}_\sigma \sharp\{(i_1, i_2), Sep(i_1, i_2) \rightarrow (c_{i_1} \ll c_{i_2} \vee c_{i_2} \ll c_{i_1}\})$$

Or, writing it without all our notations and without constraints instead of errors:

- **Variables** Products $p_1...p_m$
- **Domains** Permutations of the $p_i$s
- **External data** Orders of the customers, that is a set of $n$ subsets of the $p_i$s, $s_1...s_n$
  From which we compute the $Sep$ matrix: $Sep(j_1, j_2) = (s_{j_1} \cap s_{j_2} = \emptyset)$
- **Constraints** For all $j_1, j_2 \leq n$, if $Sep(j_1, j_2)$ then state

$$\exists \epsilon \in \{<, >\}, \forall \alpha \in s_{i_1}, \forall \beta \in s_{i_1}, \alpha \epsilon \beta$$

At that time, we have $m$ variables, a domain of size $n!$ (which is equivalent as before), and a number of constraints depending on the customers' orders, of order $n^2$, but less than before. Each constraint is a disjunction of conjonctions of inequalities (when explicitely written). At least, we have got rid of the starting times and finishing times which where partly responsible of the problem's difficulty. Of course, it still has global constraints, and has probably no solution in general, but that will be handled by an incomplete method.

## 2.2 Issues of PSP-SOS w.r.t. an incomplete method

Incomplete methods have been introduced a few decades ago. They aim at either solve a constraint satisfaction problem or CSP (such as famous SAT [Selman *et al.*, 1992]), or find

good solutions to optimization problems. Among them, local search techniques consists, roughly speaking, in choosing randomly an assignment of domain's values to the variables, explore a neighbourhood of this configuration, trying to improve an error (penalty, cost) function, eventually perform a move and iterate. Meta-heuristics such as Tabu Search [Glover and Laguna, 1997] are added to prevent being stuck in local minima of the error function. This error function is chosen so that it expresses in some way the proximity to a solution, usually by taking the number of violated constraints in the case of a CSP, or the function to optimize in an optimization problem.

Using an local search method for the PSP-SOS problem may seem both natural and quite challenging. It is natural because local search methods usually deal easily with permutation problems (or problems with an alldifferent constraint on all the $m$ variables, and all the variables have the same domain, of size $m$). They start with a permutation on the domain's values, and move by swaping two values. This ensures that the alldifferent constraint is kept satisfied without any effort during the search process. We have chosen to use a local search method for both their well-known efficiency on optimization problems, and this ability of dealing with the permutations problems.

The challenging part comes from the error function to define. Obviously, the real objective function $f$, given as the maximum of simultaneously open stacks, is not accurate enough. It is worth detailing why: the current configuration is very likely to be realized not only for one column, but several. Then the only way of improving the objective function would be to modify all these columns, except in very particular local cases [1].

Improving $f$ thus cannot be done in one move (swap of the variables' values), except if a move is defined as the swaps of several variables but there are two reasons for not trying this: firstly, the neighbourhood would be rather big w.r.t. the search space size, probably resulting in bad performances. Secondly, the goal of the challenge is not to design a dedicated algorithm for solving the PSP-SOS but to use existing constraint programming techniques to solve it, and performing such unusual moves would not play fair, in our opinion. To simply check, we have tried such an objective function with a local search method where a move consists in a swap, and it cycled as expected.

So, one issue in defining the error function is to find a way to express the value of the current configuration more accurately than by $f$. From the above discussion, we will take the $g$ function, that is, the errors w.r.t. the above constraints. Indeed, $g$ has a range greater than $f$, resulting in moves that do not decrease the real cost of the current configuration. This should not be a problem for a local search method: all the contrary, the idea is to guide the search more precisely, allowing moves staying on the same value for $f$, but intuitively improving the chances to find a better configuration. We thus will base our error function on $g$.

[1] when there is only one variable realizing the maximum. But this will not happen twice in a row, or the problem is trivial

# 3 Solving PSP-SOS with adaptive search

Adaptive Search is a local search method introduced in [Codognet, 2000]. Although being a mere local search method, it includes ideas close to greedy algorithms in the way it defines the neighbourhood exploration. The main idea is to rely on a projection of the error function on the variables. Within the current configuration, the values of the different variables are probably not the same: some variables may be close to satisfy the constraints, while other ones may be responsible for a big part of the whole error (think about the queens which attacks the most of the other queens in the $n$-queens problem, for instance). The idea is to select one of these bad variables for the next move. The configuration's neighbourhood is the domain of this particular variable. It is explored trying to minimize the error function (the main one, because of course we do still want to improve this error on not only the ones on the variables). It can happen that moving the selected variable does not enable to decrease the error function. In order to avoid cycling, a tabu-like memory is added to mark those variables for a certain number of iterations. A Tabu solving method for PSPs is also given in [Fink and Voss, 1999], but with a very different coding (as a graph in instanciation).

The adaptive search method has been implemented for permutation problems by D. Diaz and P. Codognet as an open-source C library, available online[2]. Details can be found in [Codognet and Diaz, 2001], which shows that the method is very efficient on classical benchmarks. The library includes all of the functions to solve a CSP by adaptive search, provided the user defines the error functions corresponding to his problem, both for the whole configuration and for the variables. Actually, the first one can be deduced from the second one in a way we describe below.

## 3.1 Writing the model in adaptive search

Now the question is to compute the error at the variable level, corresponding to the model described above where we have discarded the inseparables couples of customers. Counting the error corresponding to a customer is straightforward: just take the number of other customers, separable with him, and not separated. Question is, how to project this on a product $j_0$ ? In particular, should we count all the open stacks at instant $j_0$ ? For the rows having a 1, obviously yes: swaping the column may improve the cost. For the other ones, the rows being open at instant $j_0$ but not effective, it would introduce a bias in the model, because swaping the column cannot result in an improvement of $g$, at least on this row (maybe in the whole configuration, but this is not the point). So we take the following error for column $j_0$:

$$e(j_0) = \sum_{1 \leq i \leq n} [Prob_{i,j_0} = 1] \sharp\{1 \leq i \leq n, Sep(i, i')\}$$

with $[P] = 1$ if predicate $P$ is true, 0 otherwise.

Concerning the error function for the whole configuration, the adaptive search method suggests to aggregate the variables' error with an appropriate operator, coherent with the

[2] at `http://pauillac.inria.fr/~diaz/adaptive/`

problem, such as addition or maximum. As we have made enough efforts to get rid of the poor information given by the maximum, this one is rejected. Although it would not give back the $f$ function, a risk still arises that a single swap may not improve the global error function. The addition is thus a better choice, keeping the accuracy of the model. The error function is: $\sum_{1 \leq j \leq m} e(j)$. It is easily checked that this error function corresponds to the model described above with a min-conflict way of counting.

At that point the reader may have the impression that we simply have replaced the maximum by the addition in the objective function $f$, which is not such a big deal. This belonged to the possilibities that we initially studied in order to have a more accurate objective function: counting the bad zeros in the matrix (those within an open stack), couting the whole number of open stacks, etc. None of them has been kept, because the crucial property for the model to have the same optima as $f$ would not have been satisfied. Counter examples are easily found.

### 3.2 Experiments and results

We have implemented the model and error functions described above in the adaptive search library. A major drawback of this implementation is the fact that the errors are not computed incrementally, although the library leaves the possibility to do it. It would probably improve the calculation time. But it does not affect the number of iterations which is maybe a more neutral measure of the implementation's efficiency.

Experiments have been conducted on the set of instances proposed by the Challenge organization and the table of results is given in appendix A. A classical issue with the local search methods is the parameter tuning. We have chosen a maximal tabu tenure to try and force good minimas, and a maximal percentage of resets variables to ensure diversification. The maximum number of iterations has been chosen big (as an order of hunders $\times m$), which leads to a calculation time of a few minutes per instance. More details on technical issues is to be found in appendix A and code in appendix B.

Let us precise the complexity of the computation of the different functions. We count the number of arrays' accesses, the number of operations being of the same order. For each iteration, are computed:

- Error on the $m$ variables: $m$ for the computation of the $st_i$ and $ft_i$, $m \times n^2$ for the errors. That is, $\mathcal{O}(m \times n^2)$
- Error on the solution: same
- Computation of the effective error (w.r.t. $f$): $\mathcal{O}(n \times m)$
- Updates after a swap: $\mathcal{O}(n \times m)$ for storing the best solution found so far (computation of its effective error).

In the end, the complexity of an iteration is in $\mathcal{O}(m \times n^2)$, which is reasonnable.

Finally, the comparison with other solving methods can hardly be made in this paper, as there is by now few litterature on the PSP-SOS and will probably be far more after the Challenge.

## 4 Conclusion

The maximum open stack definitely appears to be a challenging problem for the constraint community, as a model shall present some features known to be difficult for constraint programming, close the global constraint problems.

We have given a model for representing the maximum open stack problem, and solved it with a local search method. The strength of the model relies on two properties: firstly, it takes into account the inner structure of the problem by focusing on relevant information. Secondly, it modifies the objective function in order to measure more accurately the value of a configuration.

### Acknowledgment

Thank you to Marc Christie, LINA, for his help.

## References

[Aggoun and Beldiceanu, 1991] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the chip compiler system. *Proceedings of ICLP91*, pages 775–789, 1991.

[Codognet and Diaz, 2001] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. *LNCS 2246, SAGA 2001, first Symposium on Stochastic Algorithms : Foundations and Applications*, 2001.

[Codognet, 2000] Philippe Codognet. Adaptive search, preliminary results. *BOOK of the 4th ERCIM/CompulogNet Workshop*, 2000.

[Diaz and Codognet, 2001] Daniel Diaz and Philippe Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 6, 2001.

[Fink and Voss, 1999] Andreas Fink and Stefan Voss. Applications of modern heuristic search methods to pattern sequencing problems. *Computers and Operations Research*, 26:17–34, 1999.

[Glover and Laguna, 1997] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[Linhares and Yanasse, 2002] Alexandre Linhares and Haracio Hideki Yanasse. Connections between cutting-pattern sequencing, vlsi design and flexible machines. *Computers and Operations Research*, 29:1759–1772, 2002.

[Michel and Hentenryck, 2001] L. Michel and P. Van Hentenryck. Localizer++: An open library for local search. *Technical Report, CS-01-03, Brown University*, 2001.

[Selman *et al.*, 1992] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. *AAAI'92*, pages 440–446, 1992.

# A Results

## A.1 Aggregate results (Truchet, Bourdon, Codognet)

We use the adaptive search C library proposed by Diaz and Codognet under public licence, available online at `http://pauillac.inria.fr/~diaz/adaptive/`. The detailed code of our implementation is given in the next appendix B. We do not have modified the solver, except for a minor part to add some time measurements for the benches. The total code (our implementation plus this slightly modified adaptive search library) is sent to the Challenge Organization separately. Experiments have been conducted on an Intel Pentium 4 at 2,66 GHz with 512 Mb RAM. The compiler is gcc version 3.3.4.

The percentage of proved optimality is zero for all instances, due to the incomplete method we use. We thus leave the column blank.

As we use a local search method, we define the search effort as the number of iterations. We recall from the paper that the complexity order for one iteration is $\mathcal{O}(m \times n^2)$, where $m$ is the number of products and $n$ the number of customers. The cutoff limit we chose is defined as a maximal number of iterations depending on $m$, which explains why the next to last column is quasi-constant for instances of same size.

The solver has been used with the following parameters. We only detail the most important ones, please refer to the code and the adaptive search documentation for more details. We did not spend too much time in parameter tuning, in particular did not try and benefit from the plateau's heuristic from adaptive search. It is possible that the results could be improved in terms of calculation time, by incrementally computing the costs, and in terms of search effort, by a finer parameter tuning.

- Number of iterations before a restart is triggered: $100 \times m$. This choice is somehow arbitrary. Our first experiments showed that the best solution had been found far before this limit for the small or easy instances, for example on the first Harvey instances, after a few hundreds of iterations for problem of size $10$ and $20$. One could argue that it would have been sufficient to fix this parameter to far less than $100 \times m$, say around $20 \times m$, to improve our results for the challenge. Now our strategy has been to give the solver a reasonnable chance of finding good solutions. Anyway a run for a typical $30$ instance lasts around a few minutes, which is reasonnable.

- Number of restarts: $10$, and only the best solution among the $10$ runs is kept

- Number of variables reset at a restart: $100$ %

- Number of runs: $1$. It would have been equivalent to perform $10$ runs fixing the number of restarts to $0$, but less convenient.

- Number of iterations a variables is frozen when it does not allow to improve the error: $m$ the number of variables. We have chosen to fix it to the maximum, in order to insist on promising neighbourhoods and get the best possible optima, although it might slow the search (see adaptive search documentation and articles for explanations on the parameter)

Since the first parameter (search effort for one restart) has voluntarily been fixed to a high value, we add another column in the table of results to give the computation time for finding the best solution.

| File | % solved optimally within the cutoff limit | Mean best value found | Total runtime per instance | | | Search effort per instance to find optimal solution | | | Total Search effort per instance | | | Runtime per instance to find optimal solution | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | mean | median | max | mean | median | max | mean | median | max | mean | median | max |
| problem_10_10 | 0 | 8.07 | 0.21 | 0.00 | 0.72 | 29.64 | 3 | 910 | 366.26 | 4 | 1035 | 0.00 | 0.00 | 0.10 |
| problem_10_20 | 0 | 15.77 | 1.34 | 1.93 | 2.32 | 72.00 | 12 | 862 | 680.96 | 1000 | 1062 | 0.02 | 0.00 | 0.45 |
| problem_15_15 | 0 | 12.92 | 1.61 | 0.01 | 4.12 | 195.04 | 7 | 1497 | 713.74 | 45 | 1543 | 0.11 | 0.00 | 2.44 |
| problem_15_30 | 0 | 26.05 | 7.30 | 11.17 | 13.36 | 331.09 | 94 | 1488 | 925.62 | 1500 | 1523 | 0.69 | 0.07 | 9.27 |
| problem_20_10 | 0 | 9.10 | 0.96 | 0.00 | 4.50 | 120.00 | 0 | 1887 | 523.25 | 0 | 2022 | 0.13 | 0.00 | 3.25 |
| problem_20_20 | 0 | 18.02 | 5.49 | 0.01 | 14.19 | 332.04 | 7 | 1960 | 916.93 | 7 | 2010 | 1.01 | 0.01 | 11.50 |
| problem_30_10 | 0 | 9.50 | 1.52 | 0.00 | 13.64 | 101.70 | 0 | 2103 | 424.47 | 0 | 3001 | 0.25 | 0.00 | 10.60 |
| problem_30_15 | 0 | 14.12 | 5.40 | 0.00 | 22.91 | 270.21 | 0 | 2824 | 790.65 | 0 | 3008 | 1.17 | 0.00 | 20.63 |
| problem_30_30 | 0 | 28.43 | 27.36 | 0.01 | 81.48 | 663.53 | 0 | 2961 | 1193.36 | 0 | 3024 | 8.76 | 0.00 | 67.47 |
| problem_40_20 | 0 | 19.27 | 17.50 | 0.00 | 78.42 | 431.87 | 0 | 3653 | 983.37 | 0 | 4001 | 6.36 | 0.00 | 68.88 |
| ShawInstances | 0 | 13.98 | 15.28 | 15.36 | 16.02 | 940.76 | 906 | 1875 | 2002.58 | 2000 | 2025 | 3.48 | 2.14 | 15.13 |
| wbo_10_10 | 0 | 5.92 | 0.43 | 0.57 | 0.70 | 64.88 | 8 | 855 | 729.35 | 1000 | 1034 | 0.01 | 0.00 | 0.17 |
| wbo_10_20 | 0 | 12.70 | 2.02 | 2.08 | 2.44 | 98.25 | 36 | 703 | 1003.25 | 1000 | 1022 | 0.03 | 0.01 | 0.43 |
| wbo_10_30 | 0 | 20.00 | 4.28 | 4.40 | 5.05 | 92.10 | 54 | 529 | 1003.05 | 1000 | 1022 | 0.08 | 0.02 | 1.56 |
| wbo_15_15 | 0 | 9.38 | 2.85 | 3.48 | 4.04 | 373.80 | 280 | 1436 | 1261.03 | 1500 | 1513 | 0.25 | 0.06 | 3.52 |
| wbo_15_30 | 0 | 20.47 | 11.86 | 12.27 | 14.23 | 424.37 | 268 | 1487 | 1504.25 | 1500 | 1543 | 1.27 | 0.30 | 13.64 |
| wbo_20_10 | 0 | 8.07 | 1.92 | 3.24 | 4.52 | 257.63 | 41 | 1729 | 1039.14 | 2000 | 2006 | 0.25 | 0.01 | 2.18 |
| wbo_20_20 | 0 | 13.81 | 10.11 | 12.47 | 14.02 | 719.89 | 597 | 1932 | 1726.08 | 2000 | 2031 | 1.70 | 0.53 | 12.22 |
| wbo_30_10 | 0 | 8.94 | 3.79 | 0.00 | 12.53 | 301.29 | 5 | 2954 | 1028.53 | 5 | 3000 | 0.68 | 0.00 | 9.91 |
| wbo_30_15 | 0 | 12.47 | 12.32 | 18.97 | 24.42 | 710.62 | 346 | 2866 | 1776.16 | 3000 | 3002 | 4.31 | 0.33 | 23.22 |
| wbo_30_30 | 0 | 22.80 | 53.68 | 70.25 | 86.78 | 1195.13 | 1112 | 2998 | 2351.08 | 3000 | 3020 | 22.54 | 16.57 | 80.20 |
| wbop_10_10 | 0 | 6.75 | 0.46 | 0.57 | 0.71 | 36.73 | 8 | 419 | 775.95 | 1000 | 1011 | 0.00 | 0.00 | 0.02 |
| wbop_10_20 | 0 | 13.53 | 1.96 | 2.04 | 2.30 | 70.92 | 53 | 474 | 978.00 | 1000 | 1044 | 0.01 | 0.01 | 0.08 |
| wbop_10_30 | 0 | 20.70 | 4.25 | 4.17 | 5.09 | 104.05 | 49 | 916 | 1004.55 | 1001 | 1023 | 0.05 | 0.02 | 0.36 |
| wbop_15_15 | 0 | 10.40 | 2.67 | 3.41 | 3.84 | 219.58 | 80 | 1401 | 1201.93 | 1500 | 1534 | 0.15 | 0.02 | 2.42 |
| wbop_15_30 | 0 | 21.20 | 11.43 | 11.53 | 14.34 | 549.77 | 487 | 1426 | 1483.28 | 1500 | 1535 | 1.10 | 0.43 | 13.94 |
| wbop_20_10 | 0 | 8.20 | 1.83 | 3.24 | 4.04 | 218.80 | 33 | 1344 | 1014.85 | 2000 | 2005 | 0.11 | 0.00 | 1.14 |
| wbop_20_20 | 0 | 15.00 | 8.58 | 11.23 | 14.26 | 560.24 | 421 | 1962 | 1495.30 | 2000 | 2027 | 1.48 | 0.34 | 12.39 |
| wbop_30_10 | 0 | 9.05 | 2.82 | 0.00 | 11.76 | 250.70 | 7 | 2506 | 832.50 | 7 | 3000 | 0.62 | 0.00 | 8.66 |
| wbop_30_15 | 0 | 12.42 | 11.29 | 18.01 | 23.14 | 717.55 | 160 | 2642 | 1650.95 | 3000 | 3007 | 2.71 | 0.09 | 16.91 |
| wbop_30_30 | 0 | 24.06 | 43.41 | 61.62 | 85.79 | 1049.28 | 947 | 2965 | 1968.21 | 3000 | 3020 | 18.49 | 10.04 | 83.67 |
| wbp_10_10 | 0 | 7.33 | 0.32 | 0.52 | 0.64 | 60.38 | 13 | 541 | 559.15 | 1000 | 1012 | 0.00 | 0.00 | 0.03 |
| wbp_10_20 | 0 | 14.36 | 1.86 | 1.95 | 2.25 | 74.87 | 39 | 358 | 960.51 | 1000 | 1037 | 0.02 | 0.01 | 0.43 |
| wbp_10_30 | 0 | 21.81 | 4.17 | 4.21 | 4.82 | 141.87 | 69 | 780 | 1003.31 | 1000 | 1038 | 0.06 | 0.03 | 0.76 |
| wbp_15_15 | 0 | 11.07 | 2.16 | 3.11 | 3.64 | 284.02 | 187 | 1402 | 1052.35 | 1500 | 1534 | 0.11 | 0.04 | 0.74 |
| wbp_15_30 | 0 | 22.67 | 10.11 | 11.06 | 13.91 | 435.80 | 349 | 1486 | 1342.29 | 1500 | 1549 | 1.06 | 0.28 | 12.97 |
| wbp_20_10 | 0 | 8.55 | 1.68 | 0.02 | 3.94 | 203.60 | 10 | 1829 | 951.85 | 58 | 2000 | 0.14 | 0.00 | 1.30 |
| wbp_20_20 | 0 | 15.52 | 7.14 | 10.45 | 13.22 | 572.47 | 311 | 1877 | 1313.01 | 2000 | 2021 | 1.45 | 0.19 | 10.10 |
| wbp_30_10 | 0 | 9.12 | 3.14 | 0.00 | 11.79 | 297.88 | 0 | 1987 | 919.20 | 0 | 3000 | 0.69 | 0.00 | 9.14 |
| wbp_30_15 | 0 | 12.93 | 10.19 | 17.57 | 22.39 | 707.60 | 404 | 2839 | 1568.67 | 3000 | 3001 | 2.92 | 0.28 | 15.83 |
| wbp_30_30 | 0 | 24.74 | 39.13 | 54.54 | 81.55 | 1053.11 | 990 | 2980 | 1875.76 | 3000 | 3024 | 16.46 | 8.25 | 69.36 |

### A.2 Individual results (Truchet, Bourdon, Codognet)

We have chosen to run the program only once with some restarts from 2 to 10, and a number of iterations before restart from $10 \times m$ to $100 \times m$, depending on the instances' sizes and difficulty. Thus the best and worst objective values found are equal.

| File | Number of runs | Objective value | | Total runtime over all runs (seconds) | Total search effort over all runs (number of iterations) | Runtime to find best solution (seconds) | Search effort to find best solution (number of iterations) |
|---|---|---|---|---|---|---|---|
| | | Best | Worst | | | | |
| Miller19 | 1 | 20 | 20 | 39,33 | 20 000 | 0,09 | 50 |
| GP1 | 1 | 45 | 45 | 387 | 13 470 | 387 | 13 470 |
| GP2 | 1 | 40 | 40 | 1408 | 55 000 | 1,14 | 44 |
| GP3 | 1 | 40 | 40 | 304 | 11 903 | 304 | 11 903 |
| GP4 | 1 | 30 | 30 | 1568 | 55 000 | 1,54 | 53 |
| GP5 | 1 | 98 | 98 | 888 | 2 000 | 12,54 | 28 |
| GP6 | 1 | 75 | 75 | 840 | 2 000 | 46,62 | 111 |
| GP7 | 1 | 75 | 75 | 6 176 | 15 000 | 4 320 | 10 491 |
| GP8 | 1 | 60 | 60 | 830 | 2 000 | 50,63 | 122 |
| NWRS1 | 1 | 8 | 8 | 1,87 | 10 000 | 0,02 | 133 |
| NWRS2 | 1 | 9 | 9 | 1,9 | 10 000 | 0,03 | 177 |
| NWRS3 | 1 | 16 | 16 | 9,27 | 15 000 | 0,18 | 327 |
| NWRS4 | 1 | 18 | 18 | 9,29 | 15 000 | 0,04 | 68 |
| NWRS5 | 1 | 23 | 23 | 26,93 | 20 000 | 0,20 | 164 |
| NWRS6 | 1 | 24 | 24 | 27,96 | 20 000 | 7,95 | 6 267 |
| NWRS7 | 1 | 37 | 37 | 180 | 27 500 | 176 | 26 690 |
| NWRS8 | 1 | 42 | 42 | 204 | 27 500 | 102 | 14 962 |
| SP1 | 1 | 9 | 9 | 0,42 | 500 | 0,24 | 283 |
| SP2 | 1 | 19 | 19 | 9,24 | 1 000 | 0,61 | 66 |
| SP3 | 1 | 37 | 37 | 273 | 7 500 | 160 | 4 391 |
| SP4 | 1 | 55 | 55 | 2 020 | 20 000 | 453 | 4487 |

## B Code

Important remark: we have made the initial mistake to exchange the columns and rows of the $Prob$ matrix from the Challenge's definition. When we realized it, it was too late to correct it. Thus in our implementation the customers are on the columns, and the products on the rows. Although it has no consequence on the resolution nor on the permutation finally provided by the solver, it may affect the lisibility of the displayed matrixes for somebody who would run the code with displays. Anyway, it does not affect the results as we have translated all the input matrixes. We apologize to the Challenge's Organizers.

It is important to remark that we have not fully used the possibilities of the library, which can be seen in the Cost_if_Swap and Executed_Swap functions. Defining these two functions better would allow to incrementally compute the errors.

```
/*
 *  First Constraint Modelling Challenge
 *  Truchet, Bourdon and Codognet Proposal
 *
 *  Based on the Solver
 *  "Adaptive search
 *
 *  Copyright (C) 2002-2003 Daniel Diaz & Philippe Codognet"
 *
 */

// Basics includes
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

// Solver's includes
#include "ad_solver.h"
#include "main.h"

/*-------------*
 * some macros *
 *-------------*/

#define NBCLIENTS Pb_sizex
#define NBMARCHANDISES Pb_sizey

#define INFINITY 10000000

// macro PROBLEME(i,j) to access the Problem matrix
// i corresponds to the product and j corresponds to the client
#define PROBLEME(i,j) Problem[j*NBMARCHANDISES+i]
// macro SEPARABLE(i,j) equals 0 if clients i and j are separable
#define SEPARABLE(i,j) Separable[j*NBCLIENTS+i]


/*------------------*
 * Global variables *
 *------------------*/
// This one override the default
int param_needed = 0;           /* overwrite var of main.c */


// Global variables for the PSP resolution

// All the following variables are precalculated during the Initialization phase
.
int Pb_sizex, Pb_sizey;    // The problem size Pb_sizex is the number of clients
, Pb_sizey is the number of products
int * Problem;             // The problem matrix (Prob in the paper)
int * Separable;           // Matrix to store whether clients i and j are separa
bles (Sep in the paper)
int * uns;                 // Number of 1 in a column product (precalculated)

// All the following variables are used to select two products to swap
// Since they are associated to the permutation, they are recalculated after eac
h swap or reset.
int * debuts, * fins;      // Vectors st_i and ft_i in the paper, relatives to t
he current permutation
int * ouverts;             // Cost vector for the products, relative to the curr
ent permutation

// All the following variables are used to store informations concerning the cur
rent best solution.
int meilleurcout;          // Cost "g" of the current best solution
int meilleurcouteffectif;  // Cost "f" (real cost) of the current best solution
int * meilleuresolution;   // Current best solution
```

```
// All the following variables are present to bench our method
int demarrage;             // Time ellapsed since the beginning of the proc
almost 0)
int termine;               // Time ellapsed when the current instance finis
int solution;              // Time to find the optimal solution
int solution_nb_iter;      // Number of iterations to find the optimal solu
int solution_nb_restart;   // Number of restarts to find the optimal soluti
int ad_passed_swap;        // Total number of swaps

// Toolbox function to obtain the time ellapsed by the current process
int getProcessTime(void) {
    struct rusage rsr_usage;
    getrusage(RUSAGE_SELF, &rsr_usage);
    return rsr_usage.ru_utime.tv_sec * 1000+rsr_usage.ru_utime.tv_usec/1000
}

// Free all allocated memory
void Detruit_Tout() {
    free(debuts);
    free(fins);
    free(uns);
    free(Separable);
    free(meilleuresolution);
    free(Problem);
}



/*------------*
 * Prototypes *
 *------------*/

/*
 *  COST_OF_SOLUTION
 *
 *  Returns the total cost of the current solution.
 *  Also computes errors on constraints for subsequent calls to
 *  Cost_On_Variable, Cost_If_Swap and Executed_Swap.
 */

// Operator used to combine the variables' errors into the configuration's
// Here, we choose the sum.
int CoutChoisi(int val, int max) {
    // if (val>max) return val; else return max;
    return val+max;
}

// function that returns the cost "g" for the current permutation (stored i
sol)
int Update_Ouverts(){
    int deb, fin, i,i1, j,j1, val;
    int max=0;

    for (i=0; i<NBCLIENTS; i++) {
            deb=0;
        fin=NBMARCHANDISES-1;
        while ((PROBLEME(ad_sol[deb],i) == 0) && (deb<NBMARCHANDISES)) deb+
        while ((PROBLEME(ad_sol[fin],i) == 0) && (fin>=0)) fin--;
        debuts[i]=deb;
        fins[i]=fin;
  }

  for(j=0;j<NBMARCHANDISES;j++){
        j1=ad_sol[j];
        val=0;
            for(i=0;i<NBCLIENTS;i++){
```

```
                if (PROBLEME(j1,i)==1){
                    for(i1=0;i1<NBCLIENTS;i1++){
                        if ((SEPARABLE(i,i1)==0) && (debuts[i1] <= j) && (fins[i
1]>=j))
                            val++;
                    }
                }
            }
        ouverts[j]=val;
        max=CoutChoisi(val,max);
  }

    return max;
}


// function that returns the cost "f" (effective cost) for the current permutati
on
int Cout_Effectif() {
  int deb, fin, i, j, val;
  int max=0;

  for (i=0; i<NBCLIENTS; i++) {
        deb=0;
        fin=NBMARCHANDISES-1;
        while ((PROBLEME(ad_sol[deb],i) == 0) && (deb<NBMARCHANDISES)) deb++;
        while ((PROBLEME(ad_sol[fin],i) == 0) && (fin>=0)) fin--;
        debuts[i]=deb;
        fins[i]=fin;
  }

  for (i=0; i<NBMARCHANDISES; i++) {
        val=0;
        for (j=0; j<NBCLIENTS; j++) {
                if ((i>=debuts[j]) && (i<=fins[j]))
                        val++;

        }
      // Here, we take the maximum
      if (val>max) max=val;
  }
  return max;
}


// How a solution should be displayed (No display for benches)
void
Display_Solution(){
#ifndef __BENCHMARK__
    int i, j;
    Update_Ouverts();
    printf("Cout Effectif=%d\n",Cout_Effectif());
    for(i=0;i<NBMARCHANDISES;i++){
        printf(" %3d|",ad_sol[i]);
        for(j=0;j<NBCLIENTS;j++){
            printf(" %d ",PROBLEME(ad_sol[i],j));
        }
        printf("|%3d ",ouverts[i]);
        printf("|%3d", uns[ad_sol[i]]);
        printf("|%3d", Cost_On_Variable(i));
        printf("\n");

    }
    printf("-------");
    for(j=0;j<NBCLIENTS;j++){
            printf("----");
        }
```

```
    printf("-----------");
    printf("\n");
    printf("debuts ");
    for(j=0;j<NBCLIENTS;j++){
        printf("%3d ",debuts[j]);
    }
    printf("\n");

    printf("fins  ");
    for(j=0;j<NBCLIENTS;j++){
        printf("%3d ",fins[j]);
    }
    printf("\n\n");
#endif
}


// Store the best solution if a file (and some stats for the benchmark)
void
Display_Solution_bis(){
    int i;
    FILE * fd;
#ifdef __BENCHMARK__
    termine = getProcessTime();
#endif
    fd=fopen(Result_Filename,"a+");
    fprintf(fd,"Instance du fichier %s\n",Problem_Filename);
    if (meilleurcouteffectif == INFINITY) {
        meilleurcouteffectif=Cout_Effectif();
    }
    fprintf(fd,"Meilleure solution trouvee pour un cout %d\n", meilleurcouteffectif);
#ifdef __BENCHMARK__
    fprintf(fd,"Stats %s %d %d %d %d %d %d %d %d %d %d %d %d %d\n",
            Problem_Filename, meilleurcout, meilleurcouteffectif,ad_nb_iter
            ad_nb_swap, ad_nb_restart, solution_nb_iter, solution_nb_restar
            ad_restart_limit, ad_passed_swap, demarrage , termine , solutio
_nb_local_min);
#endif
    fprintf(fd,"Solution = [ ");
    for (i=0; i<NBMARCHANDISES; i++) {
            fprintf(fd,"%d ",meilleuresolution[i]);
    }
    fprintf(fd,"]\n");
    fprintf(fd,"\n----------------------------------------\n\n");
    fclose(fd);
    Detruit_Tout();
}


// The Cost_Of_Solution function (needed by Adaptative Search solver)
int
Cost_Of_Solution(void)
{
  return Update_Ouverts();
}

/*
 *  COST_ON_VARIABLE
 *
 *  Evaluates the error on a variable.
 */

// The Cost_On_Variable function (needed by Adaptative Search solver)
int
Cost_On_Variable(int i)
{
  return ouverts[i];
```

```
}



/*
 *  COST_IF_SWAP
 *
 *  Evaluates the new total cost for a swap.
 */

// Using the Cost_If_Swap function is optional. This function is called to obtai
n a more efficient, incremental, computation of the neighbourhood exploration. W
e do not use it.

/*
int
Cost_If_Swap(int i1, int i2)
{
  int i,j1, j2, max;
  for (i=0; i<NBMARCHANDISES; i++) {
        if (ad_sol[i]==i1) j1=i;
        if (ad_sol[i]==i2) j2=i;
  }
  ad_sol[j1]=i2;
  ad_sol[j2]=i1;
  max=Update_Ouverts();

  ad_sol[j1]=i1;
  ad_sol[j2]=i2;
  Update_Ouverts();
  return max;
}

*/


/*
 *  EXECUTED_SWAP
 *
 *  Records a swap.
 */

// Called whenever a swap is performed. Could be used to incrementally compute t
he errors, which we do not. Here, we simply store the new permutation if it is b
etter than the optimal

void
Executed_Swap(int i1, int i2){
    int i, cout, max;
    ad_passed_swap++;
    max=Update_Ouverts();
    cout=Cout_Effectif();
    if (max<meilleurcout) {
#ifndef __BENCHMARK__
        printf("Amelioration de la solution a %d iterations(%d), cout=%d, score=%d\n",ad_nb_iter+ad_
nb_restart*ad_restart_limit,ad_nb_restart,cout,max);
#else
        solution=getProcessTime();
#endif
        meilleurcout=max;
        meilleurcouteffectif=cout;
        solution_nb_iter=ad_nb_iter;
        solution_nb_restart=ad_nb_restart;
        for (i=0;i<NBMARCHANDISES;i++){
            meilleuresolution[i]=ad_sol[i];
       }
    }
}

/*
 *  INITIALIZATIONS
 *
 *  Initialization function.
 */

void
Initializations(void)
{
  int i=0,j=0,value,unscolonne=0,k;

  // The instance of the problem is stored in the file Problem_Filename wit
 format:
  // - first line contains "NBProducts NBClients"
  // - the remaining lines contains the matrix Prob

  // BEGIN READING THE PROBLEM
  FILE * fd=fopen(Problem_Filename,"r");
  if (fd == NULL) {
        printf("ERREUR OUVERTURE DU FICHIER\n");
        exit(1);
  }
  fscanf(fd, "%d %d\n",&NBMARCHANDISES,&NBCLIENTS);

  // Creating the matrix Prob
  Problem=(int *) malloc((NBCLIENTS*NBMARCHANDISES)*sizeof(int));

  for(j=0; j<NBCLIENTS; j++) {
    for(i=0; i<NBMARCHANDISES; i++) {
                fscanf(fd,"%d",&value);
                PROBLEME(i,j)=value;
        }
  }
  fclose(fd);

  // END READING THE PROBLEM


#ifndef __BENCHMARK__
  // Display the Problem matrix
  for(i=0; i<NBMARCHANDISES; i++) {
        for(j=0; j<NBCLIENTS; j++) {
                printf("%d ",PROBLEME(i,j));;
        }
        printf("\n");
  }
#endif

  ad_size = NBMARCHANDISES;

  // Several memory allocations and initializations
  ad_sol = (int *) malloc(NBMARCHANDISES * sizeof(int));
  debuts = (int *) malloc(NBCLIENTS*sizeof(int));
  fins = (int *) malloc(NBCLIENTS*sizeof(int));
  ouverts = (int *) malloc(NBMARCHANDISES*sizeof(int));
  uns=(int *) malloc(NBMARCHANDISES*sizeof(int));
  Separable = (int *) malloc(NBCLIENTS*NBCLIENTS*sizeof(int));
  meilleuresolution= (int *) malloc(ad_size * sizeof(int));

  // The initial solution is the identity (overrided by the solver)
  for (i=0; i<ad_size; i++)
        ad_sol[i]=i;

  meilleurcout=Cost_Of_Solution();
```

```
  meilleurcouteffectif=Cout_Effectif();

  // Store the number of 1's by column (of product)
  for(j=0; j<NBMARCHANDISES; j++) {
    unscolonne=0;
    for (i=0; i<NBCLIENTS; i++) {
        unscolonne+=PROBLEME(j,i);
    }
    uns[j]=unscolonne;
  }

  // Pre-compute the matrix Sep
  for(i=0;i<NBCLIENTS;i++){
    for(j=0;j<NBCLIENTS;j++){
        SEPARABLE(i,j)=0;
        for(k=0;k<NBMARCHANDISES;k++){
        SEPARABLE(i,j)+=(PROBLEME(k,i)*PROBLEME(k,j));
        }
    }
  }

#ifndef __BENCHMARK__
  // Display the matrix Sep
  printf("Separables : \n");
   for(i=0; i<NBCLIENTS; i++) {
        for(j=0; j<NBCLIENTS; j++) {
                printf("%d ",SEPARABLE(i,j));;
        }
        printf("\n");
  }
#endif

// Default values for the solver's parameters
// Refer to Diaz and Codognet documentation for details
  ad_base_value = 0;
  ad_break_nl = 0;

// The probability to select a local min is 10%
  if (ad_prob_select_loc_min == -1)
    ad_prob_select_loc_min = 10;

// The adaptive search list
  if (ad_freeze_loc_min == -1)
    ad_freeze_loc_min = NBMARCHANDISES;

// Set to 0 for not using Tabu techniques
  if (ad_freeze_swap == -1)
    ad_freeze_swap = 0;

// Number of tabu before restart
  if (ad_reset_limit == -1)
    ad_reset_limit = NBMARCHANDISES;

// Percent of variable reseted during restart
  if (reset_percent == -1)
    reset_percent = 100;

// Number of iterations by restart (depends on the problem size)
  if (ad_restart_limit == -1)
    ad_restart_limit = 100*NBMARCHANDISES;

// Number of restarts
  if (ad_restart_max == -1)
    ad_restart_max = 10;

#ifdef __BENCHMARK__
    demarrage=getProcessTime();
    ad_passed_swap=0;
#endif
}



/*
 *  CHECK_SOLUTION
 *
 *  Checks if the solution is valid.
 */

// Since we do not know the optimal solution, a solution is never "valid".
int
Check_Solution(void)
{
  return 0;
}
```

# Using Customer Elimination Orderings to Minimise the Maximum Number of Open Stacks

## A Submission for the First Constraint Modelling Challenge

**Nic Wilson and Karen Petrie**
Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{n.wilson,k.petrie}@4c.ucc.ie

## 1 Introduction

The minimisation of the maximum number of open stacks problem involves a set of customers, each which requires a particular subset of a set of products. A solution is a total ordering of the products; the aim is to find a solution which minimises a particular cost function, the maximum number of open stacks (see below). Equivalent problems, such as minimising *pathwidth*, have been studied in the literature (see http://www.dcs.st-and.ac.uk/ ipg/challenge/).

The approach we describe in this paper is using a branch-and-bound algorithm based on a remodelling of the problem. Instead of searching for orderings of products, we search for orderings of customers, specifically, when they are eliminated from the problem. Perhaps not surprisingly, this simple idea has been suggested before for this problem, in: H. H. Yanasse, On a pattern sequencing problem to minimize the maximum number of open stacks, *European Journal of Operational Research*, Vol. 100, 454–463, 1997. In this paper we analyse this approach in some depth, and describe our implementation using constraint programming. We also discuss how the approach might be extended.

## 2 Problem and Notation

We first describe the problem with our notation. We have a set $C$ of $n$ customers and a set $P$ of $m$ products, both of which we totally order in some arbitrary way. Each customer $x$ requires a set $\mathit{prod}_x$ of products.

In the table below we consider an instance with customers $\{a, b, c, d, e, f\}$ and products $\{A, B, C, D, E, F\}$. The elements '1' within the above table indicate which customer requires which product. For example, the set $\mathit{prod}_c$ of products required by customer c is equal to $\{A, C\}$. There are also $m$ timepoints, $\{1, \ldots, m\}$. A solution states which product is produced at which timepoint. Formally, a solution $\pi$ is a function from the set of timepoints to the set of products. Product $\pi(1)$ is made first, followed by $\pi(2)$, and then $\pi(3), \ldots, \pi(m)$. Hence a solution can be considered as a sequence of $m$ products. Product $X$ is made at timepoint $\pi^{-1}(X)$. The table illustrates the solution $\pi = (A, B, C, D, E, F)$, i.e., $\pi(1) = A$, $\pi(2) = B$ etc.

For customer $x$, let $\mathit{start}_x^\pi$ be the timepoint in the solution $\pi$ when the stack for $x$ begins, i.e., when the first product that $x$ requires is made. Similarly let $\mathit{end}_x^\pi$ be the point when the stack for $x$ ends, when the last product that $x$ requires is made. We will often abbreviate $\mathit{start}_x^\pi$ and $\mathit{end}_x^\pi$ to $\mathit{start}_x$ and $\mathit{end}_x$, respectively. In the example, $\mathit{start}_c = 1$ and $\mathit{end}_c = 3$ since the first product in the solution sequence required by customer c is placed at the first time point, and the last product required by c is placed at the third timepoint. Formally, $\mathit{start}_x^\pi = \min\{\pi^{-1}(Y) : Y \in \mathit{prod}_x\}$ and $\mathit{end}_x^\pi = \max\{\pi^{-1}(Y) : Y \in \mathit{prod}_x\}$.

Given a solution $\pi$, for timepoint $j \in \{1, \ldots, m\}$, the set $\mathit{open}(j)$ of open stacks at $j$ is defined to be the set of customers whose stack is open at $j$, i.e., $\{x : \mathit{start}_x \leq j \leq \mathit{end}_x\}$. The cost of solution $\pi$ is defined to be the size of the largest set $\mathit{open}(j)$, i.e., $\max_{j \in \{1,\ldots m\}} |\mathit{open}(j)|$. The aim is to find a solution with minimal cost.

The entries $*$ in the table indicate that a customer has an open stack at a particular timepoint. For example, there is an open stack for customer $c$ at timepoint 2; this is because timepoint 2 is in the interval $[\mathit{start}_c, \mathit{end}_c] = [1, 3]$. There are five open stacks at timepoint 3, when product C is made. The cost of this solution, i.e., of the product ordering $(A, B, C, D, E, F)$ is equal to 5, since the maximum number of open stacks at any timepoint is 5.

| Timepoints $j$: | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | A | B | C | D | E | F |
| a | 1 | $*$ | 1 | $*$ | 1 | |
| b | 1 | 1 | $*$ | $*$ | $*$ | 1 |
| c | 1 | $*$ | 1 | | | |
| d | | 1 | $*$ | 1 | | |
| e | | | 1 | 1 | 1 | |
| f | | | | | 1 | 1 |
| $\lvert\mathit{open}(j)\rvert$ | 3 | 4 | 5 | 4 | 4 | 2 |
| Eliminated | | | $c$ | $d$ | $a, e$ | $b, f$ |

## 3 Customer Elimination

In this section we show how that it is sufficient to focus on a special type of solution, based on an ordering of customers.

### Generating a Customer Ordering from a Solution

We can generate a permutation of the customers from a solution, by considering the order in which customers are eliminated, i.e., the ordering of the values of $\mathit{end}_x$ over customers $x$. This is not usually unique, but we can break ties using

the initial ordering of customers. In the example, customer c is eliminated first, at timepoint 3, since $end_c = 3$, then $d$ at timepoint 4, followed by $a$ and $e$ at timepoint 5, and $b$ and $f$ at timepoint 6, since $end_b = end_f = 6$. We use the alphabetical ordering to break ties giving a customer elimination ordering $(c, d, a, e, b, f)$.

This defines a function $f$ from solutions to customer orderings. (Formally, a customer ordering is defined to be a function from $\{1, \ldots, n\}$ to $C$.) We write the effect of $f$ on solution $\pi$ as $f(\pi)$. For customers $x$ and $y$, customer $x$ is ordered before $y$ by $f(\pi)$ if and only if either (i) $x$ is eliminated before $y$ by $\pi$ (i.e., $end^{\pi}_x < end^{\pi}_y$), or (ii) $x$ is eliminated simultaneously with $y$ by $\pi$ (i.e., $end^{\pi}_x = end^{\pi}_y$), and $x < y$ (according to the input total order on customers).

**Generating a Solution from an Ordering of Customers**
We will generate a function $g$ that maps a customer ordering $\rho$ to a solution $g_\rho$. The idea is that the products required by the first customer (according to $\rho$) are introduced first, and then additional products required by the second customer etc. Ties are broken by the input ordering on products. In the example, the customer ordering $c, d, a, e, b, f$ generates a product ordering $A, C, B, D, E, F$.

Let $G(X)$ be the earliest position in the customer ordering $\rho$ that requires product $X$, so that $G(X) = \min\{\rho^{-1}(x) : prod_x \ni X\}$. Then $g_\rho$ is defined as follows: $X$ is ordered before $Y$ by $g_\rho$ if and only if either (i) $G(X) < G(Y)$ or (ii) $G(X) = G(Y)$ and $X < Y$; that is, either $X$ is first required by an earlier customer (in ordering $\rho$) than $Y$, or they are both first required by the same customer, and $X < Y$ in the input product ordering.

The example illustrates that applying $f$ then $g$ does not necessarily give the same solution as we started with: $A, B, C, D, E, F$ is changed to $A, C, B, D, E, F$. In fact the cost is even changed: the cost of the second solution is just 4, as opposed to 5 for the first solution. The following proposition states that applying $f$ and then $g$ can never increase the cost of a solution.

**Proposition 1** *If we start with a solution $\pi$ and generate the associated customer elimination sequence $f(\pi)$, and generate from that its associated solution $g(f(\pi))$, then the cost of this new solution is no worse than that of the original solution: $cost(g(f(\pi)) \leq cost(\pi)$.*

**Sketch of proof:** Consider any solution $\pi$. Write the associated customer elimination ordering $f(\pi)$ as $x_1, x_2, \ldots, x_n$. For $i = 1, \ldots, n$, let $j_i = end_{x_i}$ be the position at which customer $x_i$ is eliminated.

Let $Prod^1$ be the sequence of products appearing in positions $1, \ldots, j_1$ in the solution $\pi$, i.e., $\{\pi(1), \ldots, \pi(j_1)\}$. The open stacks sets $open(j)$ occurring at positions $j$ corresponding to elements of $Prod_1$ are increasing in size (since no customer is eliminated in this interval), so with the largest occurring at $j_1$, as customer $x_1$ is eliminated. Permuting the products in $Prod_1$ cannot make the last such set any larger, and so cannot increase the cost of the solution. We permute $Prod^1$ to put products required by customer $x_1$ first, and putting those products in input products order. Hence these products are in the order dictated by $g(f(\pi))$.

Let $Prod^2$ be the (possibly empty) set of products appearing in positions $j_1 + 1, \ldots, j_2$ in the solution $\pi$. We permute $Prod^2$ to put products required by customer $x_2$ first, and putting those products in input products order. Again, no customer is eliminated in this interval, so this cannot increase the cost of the solution. We continue this process with $Prod^3, \ldots, Prod^n$.

Applying this sequence of operations generates solution $g(f(\pi))$. None of the operations increases the cost of the solution, so $cost(g(f(\pi)) \leq cost(\pi)$, as required. □

This result leads to the following result, which means that we can search for customer elimination orderings, without losing completeness. The cost $cost(\rho)$ of a customer elimination ordering $\rho$ is defined to be $cost(g(\rho))$, the cost of the associated product ordering.

**Proposition 2** *Suppose elimination ordering $\rho$ has minimal cost, i.e., for all elimination orderings $\rho'$, $cost(\rho') \geq cost(\rho)$. Then $g(\rho)$ is an optimal solution, i.e., for all solutions $\pi$, $cost(\pi) \geq cost(g(\rho)) = cost(\rho)$.*

**Proof:** Let $\pi$ be any solution. By the previous proposition, $cost(\pi) \geq cost(g(f(\pi))$, which by definition is equal to $cost(f(\pi))$, the cost of the elimination ordering $f(\pi)$. By the hypothesis, $cost(f(\pi)) \geq cost(\rho) = cost(g(\rho))$, proving that $cost(\pi) \geq cost(g(\rho))$, Since $\pi$ was an arbitrary solution, this proves the optimality of solution $g(\rho)$. □

## Cost in terms of neighbourhoods

The *neighbourhood* $Nbd(x)$ of a customer $x$ is defined to be the set of customers that share a common product with customer $x$, i.e., $\{y \in C : prod_x \cap prod_y \neq \emptyset\}$. In the example, $Nbd(c) = \{a, b, c, e\}$; this is because $c$ requires products $A$ and $C$, and $a$ and $b$ also require $A$, and customers $a$ and $e$ also require product $C$.

We say that a customer elimination ordering $\rho$ is *feasible* if there is some solution which has $\rho$ as its associated customer elimination ordering, i.e., if there exists solution $\pi$ with $\rho = f(\pi)$. It can be checked that if we generate an elimination ordering $\rho = f(\pi)$ from a solution $\pi$, then generate a solution $g(\rho)$ from that, and an elimination ordering $f(g(\rho))$ from *that*, we get the same elimination ordering: $\rho = f(g(\rho))$. This implies that a customer elimination ordering is feasible if and only if $\rho = f(g(\rho))$.

Consider a feasible customer elimination ordering $\rho = (x_1, x_2, x_3, \ldots)$, and its associated solution $g(\rho)$. When $x_1$ is eliminated, i.e., at timepoint $end_{x_1}$, the products required by $x_1$ have been made, which means that there is an open stack for every neighbour of $x_1$. We write $Stacks(x_1) = Nbd(x_1) = Nbd(\rho(1))$. Similarly, when $x_2$ is eliminated, at timepoint $end_{x_2}$, (unless $end_{x_2} = end_{x_1}$) there is an open stack for every neighbour of $x_1$ or of $x_2$, except for customer $x_1$, which has been eliminated. We write $Stacks((x_1, x_2)) = Nbd(x_1) \cup Nbd(x_2) - \{x_1\}$.

More generally for a sequence of customers $seq = (x_1, \ldots, x_i)$ we write $Stacks(seq) = (Nbd(x_1) \cup \cdots \cup Nbd(x_i)) - \{x_1, \ldots, x_{i-1}\}$. We

can compute this iteratively using the equation $\mathit{Stacks}((x_1,\ldots,x_i)) = (\mathit{Stacks}((x_1,\ldots,x_{i-1})) - \{x_{i-1}\}) \cup (Nbd(x_i) - \{x_1,\ldots,x_{i-1}\})$.

For elimination ordering $\rho$, let $\mathit{ncost}(\rho) = \max_{i=1,\ldots,n} |\mathit{Stacks}((x_1,\ldots,x_i))|$. It is only at time-points in $\{\mathit{end}_{x_i} : i = 1,\ldots,n\}$ that the number of open stacks could decrease. If $\rho$ is feasible then customers get eliminated from $g(\rho)$ in the order $\rho$ (since $\rho = f(g(\rho))$), so the largest set of open stacks for solution $g(\rho)$ is equal to $\mathit{Stacks}((x_1,\ldots,x_i))$ for some $i = 1,\ldots,n$. Hence, for feasible $\rho$, $\mathit{cost}(\rho) = \mathit{ncost}(\rho)$.

For any customer elimination ordering $\rho$ we have: $\mathit{ncost}(\rho) \geq \mathit{cost}(\rho)$. Let $\rho' = f(g(\rho))$, which is a feasible customer elimination ordering. Then, using Proposition 1, $\mathit{ncost}(\rho') = \mathit{cost}(\rho') \leq \mathit{cost}(\rho) \leq \mathit{ncost}(\rho)$. So the minimum of $\mathit{ncost}(\rho)$ over all customer elimination orderings $\rho$ is equal to the minimum of $\mathit{cost}(\rho)$ over all customer elimination orderings, which, by Proposition 2, is equal to the minimum of $\mathit{cost}(\pi)$ over all solutions $\pi$, i.e., the cost of the optimal solutions. This shows that allowing infeasible customer elimination orderings in our search algorithms does not affect the result. The basic algorithm below performs a search over customer elimination orderings.

## 4 Basic Customer Elimination Algorithm

The algorithm is based on chronological backtracking search. The value *maxStacks* is the maximum number of open stacks allowed. For example, if we have already found a solution with cost $R$ then we could set $\mathit{maxStacks} = R - 1$ to see if it is possible to improve on this solution.

Alternatively, we could run the algorithm repeatedly, incrementing *maxStacks* each time, starting with $\mathit{maxStacks} = 1$ (or *maxStacks* equalling the size of the smallest neighbourhood). The optimal cost will be the value of *maxStacks* in the first run that succeeds.

A sequence of customers *seq* is built up incrementally. *seq* is initialised as the empty sequence.

While *seq* doesn't contain all customers, do (a) and (b):

(a) Choose customer $x$ not in *seq*, and add $x$ to the end of sequence *seq*;

(b) If $|\mathit{Stacks}(\mathit{seq})| > \mathit{maxStacks}$ then backtrack to the last reassignable choice (i.e., the last choice such that there exists an alternative not yet tried). If no such choice exists, return 'fail' and stop.

If the algorithm doesn't return 'fail', then (the final) *seq* is a customer elimination sequence with cost no more than *maxStacks*, which can be converted to a solution $g(\mathit{seq})$ with cost no more than *maxStacks*. If the algorithm returns 'fail', then every customer elimination sequence has cost greater than *maxStacks*; in which case there is no solution with cost less than *maxStacks*. (These properties follow from the results and the discussion above.)

### Complexity for problems with high optimum cost

It can be seen that this algorithm will find the optimal cost quickly (and prove optimality) for problems with high path-width, i.e., where the optimal cost is close to the number of customers.

A lower bound for the optimal cost is the size of the smallest neighbourhood. This is because the set of open stacks when the first customer is eliminated is that customer's neighbourhood. If for some customer $x$, $Nbd(x) \neq C$ then eliminating $x$ first leads to an elimination sequence with cost less than $n$. This implies that the optimal cost is equal to $n$ if and only if every customer is a neighbour of every other customer, i.e., for all $x \in C$, $Nbd(x) = C$.

Let $w$ be the cost of the optimal solution. Suppose, during the algorithm, an initial sequence *seq* of length $(n - \mathit{maxStacks})$ has been chosen, so that $(n - \mathit{maxStacks})$ customers have been eliminated, and there are *maxStacks* customers remaining to be eliminated. If $|\mathit{Stacks}(\mathit{seq})| \leq \mathit{maxStacks}$ then any choices for the remainder of the sequence will succeed, since there are only *maxStacks* customers remaining in the problem. This implies that if $\mathit{maxStacks} \geq w$ then success or failure will be determined by choosing a sequence of at most $n - \mathit{maxStacks}$ customers, and hence at most $n - w$ customers.

If $\mathit{maxStacks} < w$ then the algorithm will return 'fail', since there is no solution with cost at most *maxStacks*. Moreover, no sequence longer than $n - w + 1$ will be generated. This is because if a sequence of length $n - w + 1$ were to succeed with the test in (b) then the largest number of stacks generated so far would be not more than *maxStacks* and so at most $w - 1$; but then any extension of this sequence will have cost at most $w - 1$ (since only $w - 1$ customers remain), which contradicts $w$ being the cost of the optimal solution. Therefore we have:

*Finding an optimal solution (and proving optimality) for families of instances with $n - w$ bounded by a constant is polynomial in the number of customers and products*

## 5 Implementation with CP

The CP-based implementation of the customer elimination algorithm has $n$ search variables $x_1,\ldots,x_n$ all of domain $\{1,\ldots,n\}$ representing the ordering in which the $n$ customers are eliminated. The only constraint on these search variables is a global 'all_different' which forces the ordering to form a permutation.

The method commences by a preprocessing step which aims to calculate a lower bound for the optimal value. This is done by calculating the neighbourhood set $Nbd(x)$, for each customer $x$. This is done in a similar manner to that described in Section 3. The lower-bound ($lwb$) for the optimal value is then calculated, which corresponds to the cardinality of the smallest neighbourhood set. At this stage another set of variables $y_1,\ldots,y_n$ is created with domain $\{lwb,\ldots,n\}$. Intuitively these variables correspond to the optimal value of the current partial or full customer elimination ordering, with lower bound of $y_k$ given by the size of the set $\mathit{Stacks}(x_1,\ldots,x_k)$.

Once the bound has been calculated, and the $y$-variables have been allocated their corresponding domains, then branch-and-bound search commences. This takes the form of a standard branch-and-bound search across the $x$ variables. Every time a search variable is instantiated, the corresponding $y$ variable is calculated to give the current bound on the opti-

mal solution. There is a global 'max' constraint across this set of variables so that every time a partial ordering is found with a worse optimal value than a previous solution, early pruning can take place. The combination of this early pruning through the use of the global constraints, and the good bound on the objective value, creates an efficient solving mechanism for this problem.

# 6 Further Techniques

## 6.1 A simple dominance condition removing infeasible elimination orderings

Suppose we have scheduled a subset of the customers $C'$, as *seq*. Let $C'' = C - C'$ be the remaining unscheduled customers. Let $S$ be the union of neighbourhoods of each element in $C'$, i.e., $S = (\bigcup_{x \in C'} Nbd(x))$. Let $x$ and $y$ be two remaining customers. If we schedule $x$ next then the set $Nbd(x) - S$ get added to the current open stacks. Say that $y$ dominates $x$ (given $C'$ or *seq*) if either of the following hold:

(i) $Nbd(x) - S \supsetneq Nbd(y) - S$

(ii) $Nbd(x) - S = Nbd(y) - S$ and $y < x$.

Say that $x \in C''$ is undominated if there does not exist $y \in C''$ which dominates $x$.

When we have scheduled customers $C'$ we only need to consider undominated customers to schedule next. The reason for this is that if $x$ is not undominated there always exists a $y$ which is undominated and which dominates $x$, and any customer ordering beginning *seq*, $x$ is no better than the corresponding customer ordering where customer $y$ is brought forward just before $x$ (and so beginning *seq*, $y, x$).

This view also suggests a simple heuristic for choosing which customer $x$ to schedule next: choose one with smallest set $Nbd(x) - S$.

## 6.2 Before-overlap branching

Here we discuss another kind of decision to branch over, which can be used on its own or in conjunction with customer elimination.

Let $x$ and $y$ be two customers. We say customer $x$ is *before* customer $y$ (with respect to some solution) if $end_x < start_y$, i.e., if the stack for $x$ closes before that of $y$ opens. We say that $x$ and $y$ *overlap* if $x$ is not before $y$, and $y$ is not before $x$, i.e., if there exists some point in which both the stacks for $x$ and $y$ are open. This happens if and only if $start_x \leq end_y$ and $start_y \leq endc_x$. If $x$ and $y$ are neighbours (i.e., they require a common product) then customers $x$ and $y$ overlap. For any two customers $x$ and $y$, exactly one of the following three possibilities occurs (in any given solution):

(i) $x$ is before $y$;

(ii) $y$ is before $x$;

(iii) $x$ and $y$ overlap.

Implied constraints can be generated in each case, especially (i) and (ii).

**Propagation from *before* statements** Obviously: *before*$(x, y)$ and *before*$(y, z)$ imply *before*$(x, z)$.
If *before*$(x, y)$ and *overlap*$(y, z)$ then not *before*$(z, x)$, and also $end_x \leq end_z$.

This latter implication restricts the search for customer elimination orderings: we can assume that $x$ appears earlier in the sequence than $z$ in the customer elimination ordering. *before* statements also strongly restrict directly the possible solutions (ordering of products).

**Propagation from overlaps** Consider a set $R$ of $r$ customers, every pair of which overlap, i.e., for all $x, y \in R$, *overlap*$(x, y)$. Then the cost of the solution is at most $r$. This is because at the point $\min\{end_x : i \in R\}$, there is an open stack for each customer in $R$.

If $x$ and $y$ overlap then, when the first of them is eliminated, there is an open stack for both customers. In particular, if $end_x \leq end_y$ then at the point that $x$ is eliminated, the current set of open stacks includes both $x$ and $y$.

We could therefore construct a search tree by at each node choosing two customers, and constructing a branch for each of the three possibilities above. We will need relatively few *before* decisions at (above) a node for either inconsistency, or becoming close to generating a solution with cost within the upper bound *maxStacks*, since *before* statements strongly restrict solutions and customer elimination orderings. Nodes with almost all *overlap* decisions associated allow weaker direct propagation. However, then searching for feasible customer elimination sequences at such a node may well be effective, since the increased number of overlaps will tend to increase the number of open stacks, potentially clashing with the upper bound and allowing backtracking. (Causing further overlaps is similar to increasing the neighbourhoods of customers; customer elimination is very effective when the neighbourhoods get larger.)

# 7 Discussion

We analysed and proved equivalence of a simple reformulation of the problem (customer elimination sequences), with an associated branch-and-bound approach. We implemented this approach with CP technology; as expected, given the earlier discussion, the approach works very well when the optimum cost is high, as demonstrated by the experimental results. However, the approach is also very successful for many problems with much lower optimum cost; propagating the lower bound based on the sizes of the neighbourhoods seems to be very effective for some of these problems.

# Acknowledgements

This material is based upon works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075. We are grateful for valuable discussions with many of our colleagues including Radek Szymanek, Gilles Pesant, Steve Prestwich, David Burke, Gene Freuder, Tom Carchrae, Armagan Tarim, Joe Bater, Mark Hennesey, Alex Ferguson and Brahim Hnich.

## Appendix: Experimental Results

**Experimental Conditions:**

- All experiments were run on a Dell Latitude D400 laptop with a 2GHz 4M Pentium processor, and 1GB of RAM.
- ECL$^i$PS$^e$ 5.8 ♯79 was used for the implementation of the full algorithm, the code relies on the $ic$, $ic\_global$, $ic\_search$ and the $branch\_and\_bound$ algorithms.
- One run was used for each instance.
- The maximum time allowed for each run was 5 minutes.

The method of gauging search effort is a *deep backtrack* count. A deep backtrack is where a variable has been set to a value and search has continued, and later this search tree node has had to be returned to, and the alternative branch taken. Therefore the deep backtrack count does not include *shallow backtracks*, where a variable is assigned a value which is found purely by propagation to be inconsistent.

The mean, median and maximum number of backtracks to find the optimal solution have not been included in the two tables of aggregate results. This is because in ECL$^i$PS$^e$ a handler is triggered when a better optimal solution is found than the incumbent. It is possible to print the number of backtracks at this point (as we did with the single instances), but we could not see how to get the program to store such a value. The large number of instances made it impractical to go through the program output and calculate such values by hand.

| File | Best value | Solved Optimally? | Runtime (sec) | Search effort (bts) to find optimal solution | Total search effort (backtracks) |
|---|---|---|---|---|---|
| Miller19 | 13 | Yes | 1.26 | 0 | 40 |
| GP1 | 45 | Yes | 0.34 | 0 | 2 |
| GP2 | 40 | Yes | 0.93 | 0 | 3 |
| GP3 | 40 | Yes | 0.88 | 1 | 1 |
| GP4 | 30 | Yes | 2.04 | 0 | 2 |
| GP5 | 95 | Yes | 2.06 | 0 | 1 |
| GP6 | 75 | Yes | 13.3 | 0 | 0 |
| GP7 | 75 | Yes | 16.7 | 0 | 1 |
| GP8 | 60 | Yes | 36.4 | 0 | 2 |
| NWRS1 | 3 | Yes | 0.01 | 0 | 0 |
| NWRS2 | 4 | Yes | 0.01 | 0 | 1 |
| NWRS3 | 7 | Yes | 0.50 | 1 | 97 |
| NWRS4 | 7 | Yes | 0.02 | 0 | 0 |
| NWRS5 | 12 | Yes | 0.16 | 0 | 4 |
| NWRS6 | 12 | Yes | 0.24 | 0 | 8 |
| NWRS7 | 10 | Yes | 199.7 | 0 | 17544 |
| NWRS8 | 16 | Yes | 9.33 | 0 | 484 |
| SP1 | 9 | Yes | 199.2 | 0 | 35195 |
| SP2 | 20 | No | 299.2 | 254 | 5455 |
| SP3 | 38 | No | 283.7 | 1 | 1450 |
| SP4 | 59 | No | 292.7 | 1 | 503 |

Table 1: Individual results

| | % solved | mean best | time per instance (sec) | | | Total effort p. i. (bts) | | |
|---|---|---|---|---|---|---|---|---|
| File | optimally | value found | mean | median | max | mean | median | max |
| problem_10_10.dat | 100 | 8.03 | 0.01 | 0.04 | 0.8 | 2.17 | 12 | 120 |
| problem_10_20.dat | 100 | 8.92 | 4.28 | 0.02 | 0.09 | 0.80 | 6 | 40 |
| problem_15_15.dat | 100 | 12.8 | 0.03 | 0.11 | 1.20 | 4.6 | 23 | 209 |
| problem_15_30.dat | 100 | 14.02 | 0.009 | 0.04 | 0.09 | 0.96 | 7 | 13 |
| problem_20_10.dat | 100 | 15.87 | 1.40 | 0.74 | 209.46 | 154.77 | 96 | 30182 |
| problem_20_20.dat | 100 | 17.97 | 0.11 | 0.24 | 2.21 | 7.77 | 18 | 172 |
| problem_30_10.dat | 94 | 23.95 | 25.96 | 2.21 | 299.75 | 1091.26 | 445 | 26361 |
| problem_30_15.dat | 99 | 25.97 | 7.27 | 0.89 | 297.30 | 272.94 | 83 | 12271 |
| problem_30_30.dat | 100 | 28.32 | 0.17 | 0.15 | 1.87 | 5.45 | 14 | 60 |
| problem_40_20.dat | 96.4 | 36.38 | 20.55 | 0.53 | 298.50 | 341.08 | 70 | 6211 |
| ShawInstances.txt | 100 | 13.68 | 0.36 | 0.32 | 1.32 | 20 | 22 | 68 |
| wbo_10_10 | 100 | 5.925 | 0.01 | 0.03 | 0.04 | 2.5 | 5 | 13 |
| wbo_10_20 | 100 | 7.35 | 0.007 | 0.02 | 0.02 | 1.6 | 4 | 6 |
| wbo_10_30 | 100 | 8.2 | 0.006 | 0.02 | 0.02 | 1.3 | 4 | 6 |
| wbo_15_15 | 100 | 9.35 | 0.08 | 0.10 | 0.30 | 11 | 16 | 52 |
| wbo_15_30 | 100 | 11.58 | 0.09 | 0.10 | 0.81 | 14 | 14 | 143 |
| wbo_20_10 | 100 | 12.9 | 2.1 | 0.30 | 112 | 317 | 41 | 18, 812 |
| wbo_20_20 | 98.8 | 13.69 | 5.17 | 0.25 | 299.09 | 868.4 | 24 | 55225 |
| wbo_30_10 | 90 | 20.05 | 47.76 | 3.33 | 299.16 | 2039.38 | 148 | 17155 |
| wbo_30_15 | 90.8 | 20.96 | 40.86 | 2.47 | 299.34 | 2075.06 | 226 | 26696 |
| wbo_30_30 | 90 | 22.58 | 42.66 | 1.54 | 299.71 | 2108.03 | 247 | 24227 |
| wbop_10_10 | 100 | 6.75 | 0.005 | 0.01 | 0.02 | 0.73 | 2 | 6 |
| wbop_10_20 | 100 | 8.08 | 0.006 | 0.03 | 0.06 | 1.83 | 4 | 20 |
| wbop_10_30 | 100 | 8.55 | 0.003 | 0.02 | 0.02 | 0.83 | 4 | 6 |
| wbop_15_15 | 100 | 10.37 | 0.04 | 0.09 | 0.35 | 4.83 | 10 | 60 |
| wbop_15_30 | 100 | 12.15 | 0.04 | 0.10 | 0.32 | 5.48 | 12 | 47 |
| wbop_20_10 | 100 | 14.28 | 0.26 | 0.15 | 3.60 | 26.15 | 20 | 401 |
| wbop_20_20 | 100 | 14.87 | 0.25 | 0.21 | 4.28 | 20.52 | 21 | 435 |
| wbop_30_10 | 92.5 | 22.48 | 32.67 | 1.03 | 299.72 | 1513.8 | 50 | 14966 |
| wbop_30_15 | 95 | 22.38 | 25.26 | 1.12 | 299.17 | 1813.35 | 83 | 26968 |
| wbop_30_30 | 98.5 | 23.84 | 9.56 | 0.55 | 299.61 | 442.45 | 71 | 16291 |
| wbp_10_10 | 100 | 7.28 | 0.01 | 0.03 | 0.13 | 4.13 | 5 | 68 |
| wbp_10_20 | 100 | 8.71 | 0.005 | 0.02 | 0.03 | 0.97 | 4 | 8 |
| wbp_10_30 | 100 | 9.31 | 0.003 | 0.02 | 0.02 | 0.53 | 3 | 5 |
| wbp_15_15 | 100 | 11.05 | 1.023 | 0.11 | 34.36 | 292.50 | 14 | 10152 |
| wbp_15_30 | 100 | 13.09 | 0.023 | 0.060 | 0.371 | 3.075 | 11 | 54 |
| wbp_20_10 | 100 | 15.13 | 8.08 | 0.35 | 146.07 | 1386.15 | 18 | 32319 |
| wbp_20_20 | 98.8 | 15.41 | 7.61 | 0.18 | 298.72 | 1705.12 | 39 | 101327 |
| wbp_30_10 | 90 | 23.20 | 46.07 | 2.55 | 299.71 | 2094.25 | 80 | 15758 |
| wbp_30_15 | 85 | 23.03 | 49.72 | 0.90 | 299.92 | 2816.48 | 75 | 29906 |
| wbp_30_30 | 91.4 | 24.47 | 34.70 | 0.99 | 299.73 | 2545.60 | 149 | 93548 |

Table 2: Aggregate results