Minimizing the maximum number of open stacks by customer search

Geoffrey Chu and Peter J. Stuckey

NICTA Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au

Abstract. We describe a new exact solver for the minimization of open stacks problem (MOSP). By combining nogood recording with a branch and bound strategy based on choosing which customer stack to close next, our solver is able to solve hard instances of MOSP some 5-6 orders of magnitude faster than the previous state of the art. We also derive several pruning schemes based on dominance relations which provide another 1-2 orders of magnitude improvement. One of these pruning schemes largely subsumes the effect of the nogood recording. This allows us to reduce the memory usage from an potentially exponential amount to a constant ∼2Mb for even the largest solvable instances. We also show how relaxation techniques can be used to speed up the proof of optimality by up to another 3-4 orders of magnitude on the hardest instances.

1 Introduction

The *Minimization of Open Stacks Problem* (MOSP) [10] can be described as follows. A factory manufactures a number of different products in batches, i.e., all copies of a given product need to be finished before a different product is manufactured, so there are never two batches of the same product. Each customer of the factory places an order requiring one or more different products. Once one product in a customer's order starts being manufactured, a stack is opened for that customer to store all products in the order. Once all the products for a particular customer have been manufactured, the order can be sent and the stack is freed for use by another order. The aim is to determine the sequence in which products should be manufactured to minimize the maximum number of open stacks, i.e., the maximum number of customers whose orders are simultaneously active. The importance of this problem comes from the variety of real situations in which the problem (or an equivalent version of it) arises, such as cutting, packing, and manufacturing environments, or VLSI design. Indeed the problem appears in many different guises in the literature, including: graph path-width and gate matrix layout (see [3] for a list of 12 equivalent problems). The problem is known to be NP-hard [3].

We can formalize the problem as follows. Let P be a set of products, C a set of customers, and $c(p)$ a function that returns the set of customers who have ordered product $p \in P$. Since the products ordered by each customer $c \in C$ are placed in a stack different from that of any other customer, we use c to denote

	p_1	p_2	p_3	p_4	p_5	p_6	p_7
c_1	X	.	.	.	X	.	X
c_2	X	.	.	X	.	.	.
c_3	.	X	.	X	.	X	.
c_4	.	.	X	X	.	X	X
c_5	.	.	X	.	X	.	.

(a)

	p_7	p_6	p_5	p_4	p_3	p_2	p_1
c_1	X	–	X	–	–	–	X
c_2	.	.	.	X	–	–	X
c_3	.	X	–	X	–	X	.
c_4	X	X	–	X	X	.	.
c_5	.	.	X	–	X	.	.

(b)

Fig. 1. (a) An example $c(p)$ function: $c_i \in c(p_j)$ if the row for c_i in column p_j has an X. (b) An example schedule: c_i is active when product p_j is scheduled if the row for c_i in column p_j has an X or a –.

both a client and its associated stack. We say that customer c is active (or that stack c is open) at time k in the manufacturing sequence if there is a product required by c that is manufactured before or at time k, and also there is a product manufactured at time k or afterwards. In other words, c is active from the time the first product ordered by c is manufactured until the last product ordered by c is manufactured. The MOSP aims at finding a schedule for manufacturing the products in P (i.e., a permutation of the products) that minimizes the maximum number of customers active (or of open stacks) at any time. We call a problem with n customers and m products an $n \times m$ problem.

Example 1. Consider a 5×7 MOSP for the set of customers $C = \{c_1, c_2, c_3, c_4, c_5\}$, and set of products $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, and a $c(p)$ function determined by the matrix M shown in Figure 1(a), where an X at position M_{ij} indicates that client c_i has ordered product p_j.

Consider the manufacturing schedule given by sequence $[p_7, p_6, p_5, p_4, p_3, p_2, p_1]$ and illustrated by the matrix M shown in Figure 1(b), where client c_i is active at position M_{ij} if the position contains either an X (p_j is in the stack) or an – (c_i has an open stack waiting for some product scheduled after p_j). Then, the active customers at time 1 are $\{c_1, c_4\}$, at time 2 $\{c_1, c_3, c_4\}$, at time 3 $\{c_1, c_3, c_4, c_5\}$, at time 4 $\{c_1, c_2, c_3, c_4, c_5\}$, at time 5 $\{c_1, c_2, c_3, c_4, c_5\}$, at time 6 $\{c_1, c_2, c_3\}$, and at time 7 $\{c_1, c_2\}$. The maximum number of open stacks for this particular schedule is thus 5.□

The MOSP was chosen as the subject of the first Constraint Modelling Challenge [6] posed in May 2005. Many different techniques were explored in the 13 entries to the challenge. The winning entry by Garcia de la Banda and Stuckey [2] concentrated on two properties of the MOSP problem. First, the permutative redundancy found in the MOSP problem leads naturally to a dynamic programming approach [2]. This approach is also largely equivalent to the constraint programming approach described in [5] where permutative redundancies are pruned using a table of no-goods. Second, by using a branch and bound method, the upper bound on the number of open stacks can be used to prune branches in various ways. These two techniques are very powerful and led to a solver that was an order of magnitude faster than any of the other entries in the 2005 MOSP challenge. The *Limited Open Stacks Problem*, $LOSP(k)$, is the decision version of the problem, where we determine if for some fixed k there is an order of products that requires at most k stacks at any time. The best approach of [2] solves

the MOSP problem by repeatedly solving $LOSP(k)$ and reducing k until this problem has no solution.

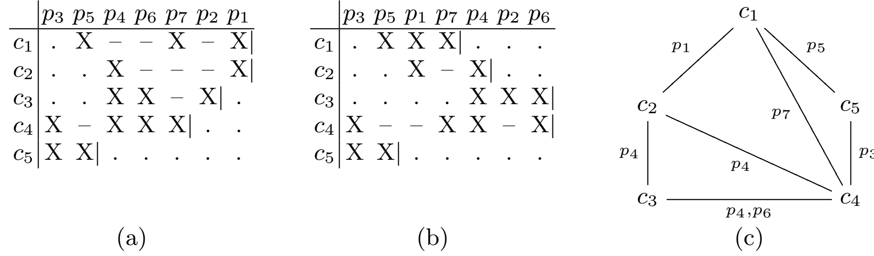
The search strategy used in this winning entry (branching on which product to produce next), is actually far from optimal. As was first discussed in [8] and shown in [9], branching on which customer stack to close next is never worse than branching on which product to produce next, and is usually much better, even when the number of customers is far greater than the number of products. This is the result of a simple dominance relation. In this paper we show that combining this search strategy with nogood recording produces a solver that is some 5-6 orders of magnitude faster than the winning entry to the Modelling Challenge on hard instances. We also derive several other dominance rules that provide a further 1-2 orders of magnitude improvement. One rule in particular largely subsumes the effect of the nogood recording. This allows us to reduce the memory usage from an potentially exponential amount to a constant ∼2Mb for even the largest solvable instances. We also utilise relaxation techniques to speed up the proof of optimality for the hardest instances by a further 3-4 orders of magnitude. With all the improvements, our solver is able to solve all the open instances from the Modelling Challenge within 10 seconds!

2 Searching on Customers

Our solver employs a branch and bound strategy for finding the exact number of open stacks required for an MOSP instance. The MOSP instance is treated as a series of satisfaction problems LOSP(k), where at each stage, we ask whether there is a solution that uses no more than k stacks. If a solution is found, we decrease k and look for a better solution.

We briefly define what a dominance relation is. A dominance relation $\triangleright$ is a binary relation defined on the set of partial problems generated during a search algorithm. For a satisfaction problem, if P_i and P_j are partial problems corresponding to two subtrees in the search tree, then $P_i \triangleright P_j$ imply that if P_j has a solution, then P_i must have a solution. This means that if we are only interested in the satisfiability of the problem, and P_i dominates P_j, then as long as P_i is searched, P_j can be pruned.

The customer based search strategy is derived from following idea. Given a product order $U = [p_1, p_2, \ldots p_n]$, define a customer close order $T = [c_1, c_2, \ldots c_m]$ as the order in which customer stacks can close given U. Construct U' such that we first schedule all the products needed by c_1, then any products required by c_2, then those required by c_3, etc. It is easy to show that if U is a solution, then so is U'. Clearly this can be converted to a dominance relation. It is sufficient to search only product orderings where every product is required by the next stack to close. This can be achieved by branching on which customer stack to close next, and then scheduling exactly those products which are needed. The correctness of this search strategy is proved in [8] and [7]. In fact, it can be shown that the customer based search strategy never examines more nodes than the search strategy based on choosing which product to produce next (even when the strongest look ahead pruning of [2] is used).



	p_3	p_5	p_4	p_6	p_7	p_2	p_1
c_1	.	X	–	–	X	–	X|
c_2	.	.	X	–	–	–	X|
c_3	.	.	X	X	–	X|	.
c_4	X	–	X	X	X|	.	.
c_5	X	X|	.	.	.	.	.

(a)

	p_3	p_5	p_1	p_7	p_4	p_2	p_6
c_1	.	X	X	X|	.	.	.
c_2	.	.	X	–	X|	.	.
c_3	.	.	.	.	X	X	X|
c_4	X	–	–	X	X	–	X|
c_5	X	X|	.	.	.	.	.

(b)

Fig. 2. (a) A schedule corresponding to customer order $[c_5, c_4, c_3, c_2, c_1]$. (b) A schedule corresponding to customer order $[c_5, c_1, c_2, c_3, c_4]$. (c) The customer graph (ignoring self edges) with edges labelled by the products that generate them.

Example 2. Consider the schedule U shown in Figure 1(b), the customers are closed in the order $\{c_4, c_5\}$ when p_3 is scheduled, then $\{c_3\}$ when p_2 is scheduled, then $\{c_2, c_1\}$ when p_1 is scheduled. Consider closing the customers in the order $T = [c_5, c_4, c_3, c_2, c_1]$ compatible with U. This leads to a product ordering, for example, of $U' = [p_3, p_5, p_4, p_6, p_7, p_2, p_1]$. The resulting scheduling is shown in Figure 2(a). It only requires 4 stacks (and all other schedules with this closing order will use the same maximum number of open stacks).

Define the *customer graph* $G = (V, E)$ for an open-stacks problem as: $V = C$ and $E = \{(c_1, c_2) \mid \exists p \in P, \{c_1, c_2\} \subseteq c(p)\}$, that is, a graph in which nodes represent customers, and two nodes are adjacent if they order the same product. Note that, by definition, each node is self-adjacent. Let $N(c)$ be the set of nodes adjacent to c in G. The customer graph for the problem of Example 1 is shown in Figure 2(c).

Rather than thinking in terms of products then, it is simpler to think of the MOSP problem entirely in terms of the customer graph. All functions $c(p)$ that produce the same customer graph have the same minimum number of stacks. Thus the products are essentially irrelevant. Their sole purpose is to create edges in the customer graph. Thus we can think of the MOSP this way. For each customer c, we have an interval during which their stack is open. If there is an edge between two nodes in the customer graph, then their intervals must overlap. If a customer close order that satisfies these constraints are found, an equivalent product ordering using the same number of stacks can always be found.

We define our terminology. At each node there is a set of customer stacks S that have been closed. The stacks which have been opened (not necessarily currently open) are $O(S) = \cup_{c \in S} N(c)$. The set of currently open stacks is given by $O(S) - \{c \in O(S) | N(c) \subseteq O(S)\}$. For each c not in S, define $o(c, S) = N(c) - O(S)$, i.e. the new stacks which will open if c is the next stack to close.

Define $open(c, S) = |o(c, S)|$ and $close(c, S) = |\{d | o(d, S) \subseteq o(c, S)\}|$, i.e. the new stacks that will open and the number of new stacks that will close respectively if we close c next.

Suppose that customers S are currently closed, then closing c requires opening $o(c, S)$, so the number of stacks required is $|O(S) - S \cup o(c, S)|$. If we are solving LOSP(k) and $|O(S) - S \cup o(c, S)| > k$ then it is not possible

to close customer c next, and we call the sequence S ++ $[c]$ *violating*. Note $|O(S) - S \cup o(c, S)| \geq open(c, S)$.

We define the *playable* sequences as follows: the empty sequence ϵ is playable; S ++ $[c]$ is playable if S is playable and S ++ $[c]$ is not *violating*.

A *solution* S of the LOSP(k) is a playable sequence of all the customers C. This leads to an algorithm for MOSP by simply solving LOSP(k) for k varying from $|C| - 1$ (there is definitely a solution with $|C|$) to 1, and returning the smallest k that succeeds.

```
MOSP(C, N)
  for (S ∈ 2^C) prob[S] := false
  for (k ∈ |C| − 1, . . . , 1)
    if (¬ playable(∅,C,k,N) return k + 1

playable(S, R, k, N)
  if (prob[S]) return false
  if (R = ∅) return true
  for (c ∈ R)
    if (|O(S) − S ∪ o(c, S)| ≤ k)
      if (playable(S ∪ {c}, R − {c}, k, N)) return true
  prob[S] := true
  return false
```

This simple algorithm tries to schedule the customers in all possible orders using at most k stacks. The memoing of calls to playable just records which partial sets of customers, S, have been examined already by setting $prob[S] = true$. If a set has already been considered it either succeeded and we won't look further, or if it failed, we record the nogood and return $false$ when we revisit it.

The algorithm can be improved by noting that if $o(c_i, S) \subseteq o(c_j, S)$ and $i < j$, then clearly, we can always play i before j rather than playing j immediately, since closing j will close i in any case. Hence move j can be removed from the candidates R considered for the next customer to close.

Example 3. Reexamining the problem of Example 1 using the closing customer schedule of $[c_5, c_1, c_2, c_3, c_4]$ results in many possible schedules (all requiring the same maximum number of open stacks). One is shown in Figure 2(b). This uses 3 open stacks and is optimal since e.g. product p_4 always requires 3 open stacks.

3 Improving the search on customers

In this section we consider ways to improve the basic search approach by exploiting several other dominance relations.

3.1 Definite Moves

Suppose S ++ $[q]$ is playable and $close(q, S) \geq open(q, S)$, then if there is any solution extending S there is a solution extending S ++ $[q]$. This means that

we can prune all branches other than q. Intuitively speaking, q is such a good move at this node that it is always optimal to play it immediately.

Theorem 1. *Suppose S ++ $[q]$ is playable and $close(q, S) \geq open(q, S)$, then if $U' = S$ ++ R is a solution, there exists a solution $U = S$ ++ $[q]$ ++ R'.*

Proof. Suppose there was a solution $U' = S$ ++ $[c_1, c_2, \ldots, c_m, q, c_{m+1}, \ldots, c_n]$. We claim that $U = S$ ++ $[q, c_1, \ldots, c_m, c_{m+1}, \ldots, c_n]$ is also a solution. The two sequences differ only in the placement of q. The number of stacks which are open at any time before, or any time after the set of customers $\{c_1, c_2, \ldots, c_m, q\}$ are played is identical for U and U', since it only depending on the set of customers closed and not the order. Thus if U' is a solution, then U has less than or equal to k open stacks at those times. Since S ++ $[q]$ is playable, the number of open stacks when q is played in U is also less than or equal to k. Finally, the number of open stacks when S ++ $[q, c_1, c_2, \ldots, c_i]$ has been played in U is always less than or equal to the number of open stacks when S ++ $[c_1, c_2, \ldots, c_i]$ has been played in U', because we have at most $open(q, S)$ extra stacks open, but at least $close(q, S)$ extra stacks closed. Thus the number of open stacks at these times are also less than or equal to k and U is a solution.

3.2 Better Moves

While definite moves are always worth playing we can find similarly that one move is always better than another. If both S ++ $[q]$ and S ++ $[r, q]$ are playable and $close(q, S \cup \{r\}) \geq open(q, S \cup \{r\})$ then if there is a solution extending S ++ $[r]$, there exists a solution extending S ++ $[q]$. This means that we do not need to consider move r at this node. Intuitively, q is so much better than r that rather than playing r now, it is always better to play q first.

Theorem 2. *Suppose S ++ $[q]$ and S ++ $[r, q]$ are playable and $close(q, S \cup \{r\}) \geq open(q, S \cup \{r\})$ then if $U' = S$ ++ $[r]$ ++ R is a solution there exists a solution $U = S$ ++ $[q]$ ++ R'.*

Proof. Suppose there was a solution $U' = S$ ++ $[r, c_1, c_2, .., c_m, q, c_{m+1}, \ldots, c_n]$. The conditions imply that if r is played now, q becomes a definite move. By the same argument as above, $U'' = S$ ++ $[r, q, c_1, \ldots, c_n]$ is also a solution. Now if we swap q with r, the number of new stacks opened before r increases by at most $open(q, S)$, but the number of new stacks closed before r increases by exactly $close(q, S)$. Also, playing q after S does not break the upperbound by our condition. Thus $U = S$ ++ $[q, r, c_1, \ldots, c_n]$ is also a solution. □

Although "better move" seems weaker than "definite move" as it prunes only one branch at a time rather than all branches but one, it is actually a generalisation, as by definition any "definite move" is "better" than all other moves. Our implementation of "better move" subsumes "definite move" so we will simply consider them as one improvement.

3.3 Old Move

Let $S = [s_1, s_2, \ldots, s_n]$. Suppose $U = [s_1, s_2, .., s_m, q, s_{m+1}, \ldots, s_n]$ is playable and we have previously examined the subtree corresponding to state $[s_1, s_2, .., s_m, q]$. Then we need not consider sequences starting with $U' = S \ ++ \ [q]$ because we will have already considered equivalent sequences earlier when searching from state $[s_1, s_2, .., s_m, q]$.

Theorem 3. *Let $S = [s_1, s_2, \ldots, s_n]$. Suppose that $S' = [s_1, s_2, .., s_m, q, s_{m+1}, \ldots, s_n]$ is playable, then if $U' = S \ ++ \ [q] \ ++ \ R$ is a solution then $U = S' \ ++ \ R$ is a solution.*

Proof. S' is playable by assumption so the number of open stacks at any time during S' is less than or equal to k. At any point after S', the number of open stacks are identical for U and U' since it only depends on the set of closed customers and not the order. Hence U is also a solution.

At any node, if it is found that at some ancestor node, the q branch has been searched and U is playable, then q can immediately be pruned. This pruning scheme was mentioned in [8], but it was incorrectly stated there. The author of [8] failed to note that the condition that U is playable is in fact crucial, because if U was not playable, then the set $S \ ++ \ [q]$ would have been pruned via breaking the upper bound and would not have in fact been previously explored, thus pruning it now would be incorrect.

Naively, it would appear to take $O(|C|^3)$ time to check the "old move" condition at each node. However, it is possible to do so in $O(|C|)$ time. At each node we keep a set $Q(S)$ of all the old moves. i.e. the set of moves q such that we can find $S' = [s_1, s_2, .., s_m, q, s_{m+1}, \ldots, s_n]$ which is playable, and such that move q has already been searched at the node $S'' = [s_1, .., s_m]$. Note that by definition, when a move r has been searched at the current node, r will be added to $Q(S)$. It is easy to calculate $Q(S \ ++ \ [s_{n+1}])$ when we first reach that child node. First, $Q(S \ ++ \ [s_{n+1}]) \subseteq Q(S)$, since if $S' = [s_1, s_2, .., s_m, q, s_{m+1}, \ldots, s_n, s_{n+1}]$ is playable then by definition so is $S' = [s_1, s_2, .., s_m, q, s_{m+1}, \ldots, s_n]$. Second, to check if each $q \in Q(S)$ is also in $Q(S \ ++ \ [s_{n+1}])$, we simply have to check whether the last move in $S' = [s_1, s_2, .., s_m, q, s_{m+1}, \ldots, s_n, s_{n+1}]$ is playable, as all the previous moves are already known to be playable since $q \in Q(S)$. Checking the last move takes constant time so the total complexity is $O(|C|)$. There are some synergies between the "better move" improvement and the "old move" improvement. If $q \in Q(S)$ and q is better than move r, then we can add r to $Q(S)$ as well. This allows "old move" to prune sets that we have never even seen before.

3.4 Upperbound heuristic

In this section, we describe an upperbound heuristic which was found to be very effective on our instances. A good heuristic for finding an optimal solution is useful from a practical point of view if no proof of optimality is required. It is also a crucial component for the relaxation techniques described in the next

Table 1. Comparison of upperbound heuristic, versus complete search on some difficult problems. Times in milliseconds

Instance	Orig. time	Heur. time	Speedup
100-100-2	4136.3	39.8	104.0
100-100-4	4715.5	43.3	108.8
100-100-6	8.2	12.8	0.6
100-100-8	9.2	6.3	1.5
100-100-10	1.6	1.3	1.3
125-125-2	1159397.3	385.1	3010.3
125-125-4	2593105.1	398.9	6500.1
125-125-6	8975.9	424.9	21.1
125-125-8	187.8	146.1	1.3
125-125-10	22.2	8.3	2.7

subsection which can give several orders of magnitude speedup on the proof of optimality for hard problems.

In [2] the authors tried multiple branching heuristics in order to compute and upper bound, but only applied them in a greedy fashion, effectively searching only 1 leaf node for each. We can do much better by performing an incomplete search where we are willing to explore a larger number of nodes, but still much fewer than a complete search. Simple ways of doing this using our complete search engine include, sorting the choices according to some criteria, and only trying the first m moves for some m. Or trying all the moves which are no worse than the best by some amount e, etc.

One heuristic that is extremely effective is to only consider the moves where we close a customer stack that is currently open, the intuition being that if a stack is not even open yet, there is no point trying to close it now. Although this seems intuitively reasonable, it is in fact not always optimal. In practice however, an incomplete search using this criteria is very fast, and finds the optimal solution almost all the time, and several orders of magnitude faster than the complete search for some hard instances. The reason for its strength comes from its ability to exploit a not quite perfect dominance relation. Almost all the time, subtrees where we close a stack that is not yet open is dominated by one where we close a currently open stack, and thus we can exploit this to prune branches similarly to what we did in Section 3. The dominance is not always true however, so using such a pruning rule makes it an incomplete, heuristic search. The procedure ub_MOSP is identical to that for MOSP except that the line **for** $(c \in R)$ is replaced by **for** $(c \in R \cap O(S))$.

See Table 1 for a brief comparison of the times required to find the optimal solution.

3.5 Relaxation

Relaxation has been used in [4] in the context of a local search method. The idea there was to try to relax the problem in such a way that solution density is increased and thus better solutions can be found quicker. However, those methods are of no help for proving optimality. In this section we show how relaxation can be used to speed up the proof of optimality.

As was seen in the experimental results in [2], the sparser instances of MOSP are substantially harder than denser instances of MOSP given the same number of customers and products. This can be explained by the fact that in sparser instances, each customer has far fewer neighbours in the customer graph, thus many more moves would fall under the upper bound limit at each node and both the depth and the branching factor of the search tree are dramatically increased compared to a dense instance of the same size.

However, the sparsity of these instances also leads to a potential optimization. Since the instance is sparse and the optimum is low (e.g. 20-50 for a 125×125 problem) it is possible that not all of the constraints are actually required to force the lower bound. It is possible that there is some small "unsatisfiable core" of customers which are producing the lower bound. If such an unsatisfiable core exists and can be identified, we can potentially remove a large number of customers from the problem and make the proof of optimality much quicker. It turns out that this is often possible.

First, we will show how we can relax the MOSP instance. Naively, we can simply delete an entire node in the customer graph and remove all edges containing that node. This represents the wholesale deletion of some constraints and of course is a valid relaxation. However, we can do much better using the following result from [1] (although only informal arguments are given for correctness)

Lemma 1. *If G' is some contraction of G, where G represents the customer graph of an MOSP instance, then G' is a relaxation of G.*

So by using this lemma, we can get some compensation by retaining some of the edges when we remove a node. Next we need to identify the nodes which can be removed/merged without loosening the lower bound on the problem.

The main idea is that the longer a customer's stack is open in the optimal solutions, the more likely it is that that customer is contributing to the lower bound, since removal of such a customer would mean that there is a high chance that one of the optimal solutions can reduce to one needing one fewer stack. Thus we want to avoid removing such customers. Instead we want to remove or merge customers whose stacks are usually open for a very short time. One naïve heuristic is to greedily remove nodes in the customer graph with the lowest degree. Fewer edges coming out of a node presumably means that the stack is open for a shorter period of time on average.

A much better heuristic comes from the following idea. Suppose there exist a node c such that any neighbour of c is also connected to most of the neighbours of c, then when c is forced open by the closure of one of those neighbours, that neighbour would also have forced most of the neighbours of c to open, and thus c will be able to close soon afterwards and will only be open for a short time. The condition that most neighbours of c are connected to most other neighbours of c is in fact quite common for sparse instances due to the way that the customer graph is generated from the products (each product produces a clique in the graph). To be more precise, in our implementation, the customers are ranked according to:

$$F(c) = \sum_{c' \in N(c)} |N(c) - N(c')|/|N(c)| \tag{1}$$

This is a weighted average of how many neighbours c' of c are not connected to each neighbour of c. The weights represents the fact that neighbours with fewer neighbours are more likely to close early and be the one that forces c to open. We merge the node c with the highest value of $F(c)$ with the neighbouring node c' with the highest value of $|N(c) - N(c')|$, as that node stands to gain the highest number of edges.

Although we have a good heuristic for finding nodes to merge, it is quite possible to relax too much to the point that the relaxed problem has a solution lower than the true lower bound of the original problem, in which case it will be impossible to prove the true lower bound using this relaxed problem. Thus it is important that we have a quick way of finding out when we have relaxed too much. This is where the very fast and strong upperbound heuristic of the previous subsection is needed. The overall relaxation algorithm is as follows:

```
relax_MOSP(C, N)
  ub := ub_MOSP(C, N)                              % ub is an upper bound
  (C', N') := (C, N)
  while (|C'| > ub)
    (C', N') := merge_one_pair(C', N')             % relax problem
  while ((C, N) ≠ (C', N'))
    relax_ub := ub_MOSP(C', N')
    if (relax_ub < ub)                             % too relaxed to prove lb
      (C', N') := unmerge_one_pair(C', N')         % unrelax problem
    else
      lb := MOSP(C', N')                           % compute lowerbound
      if (lb < ub)                                 % too relaxed to prove lb
        (C', N') := unmerge_one_pair(C', N')       % unrelax problem
      else return ub                               % lb = ub
  return MOSP(C, N)                                % relaxation failed!
```

As can be seen, the upperbound heuristic is necessary to find a good (optimal) solution quickly. It is also used to detect when we are too relaxed as quickly as possible so that we can unrelax. If the upperbound heuristic is sufficiently good, we will quickly be able to find a relaxation that removes as many customers as possible without being too relaxed. If the upperbound heuristic is weak however, we could waste a lot of time searching in a problem that is in fact too relaxed to ever give us the true lowerbound. In practice, we have found that our upperbound heuristic is quite sufficient for the instances we tested it on.

There are a few optimisations we can make to this basic algorithm. Firstly, when an unmerge is performed, we can attempt to extend the last solution found to a solution of this less relaxed problem. If the solution extends, then it is still too relaxed and we need to unmerge again. This saves us having to actually look for a solution to this problem. Secondly, naively, when we perform an unrelax,

Table 2. Results on the open problems from the Constraint Modelling Challenge 2005, comparing versus the the winner of the challenge [2]. Times in milliseconds.

.

	[2]			This paper		
	Best	Nodes	Time	Optimal	Nodes	Time
SP2	19	25785	1650	19	1236	7
SP3	36	949523	∼3600000	34	84796	410
SP4	56	3447816	∼14400000	53	1494860	9087

we can simply unmerge the last pair of nodes that were merged. However, we can do better. One of the weaknesses of the current algorithm is that the nodes to be merged are chosen greedily using equation (1). If this happens to choose a bad relaxation that lowers the lowerbound early on, then we will not be able to remove any more customers beyond that point. We can fix this to some extent by choosing which pair of nodes to unmerge when we unrelax. We do this by considering each of the problems that we get by unmerging each pair of the current merges. If the last solution found does not extend to a solution for one of these, then we choose that unmerge, as this unrelaxation gives us a chance to prove the true lowerbound. If the last solution extends to a solution for all of them, we unmerge the last pair as per usual. This helps to get rid of early mistakes in merging and is useful on several of our instances.

4 Experimental evaluation

In this section we demonstrate the performance of our algorithm, and the effect of the improvements. The experiments were performed on a Xeon Pro 2.4GHz processor with 2Gb of memory. The code implementing the approaches were compiled using g++ with -O3 optimisation.

4.1 Modelling Challenge instances

Very stringent correctness tests were performed in view of the large speedups achieved. All versions of our solver were run on the 5000+ instances used in the 2005 model challenge [6], as well as another 100,000 randomly generated instances of size 10×10 to 30×30 and various densities. The answers were identical with the solver of [2].

We compare our solver with the previous state of the art MOSP solver, on which our solver is based. The results clearly show that our solver is orders of magnitude faster than the original version. Getting an exact speedup is difficult as almost all of the instances that the original version can solve are solved trivially by our solver in a few milliseconds, whereas instances that our solver finds somewhat challenging are completely unsolvable by the original version.

Our solver was able to solve all the open problems from the Modelling Challenge: SP2, SP3, and SP4. Table 2 compares these problems with the best results from the Challenge by [2]. The nodes and times (in milliseconds) for [2] are for finding the best solution they can. The times for our method are for the full solve including proof of optimality (using all improvements).

4.2 Harder Random instances

Of the 5000+ instances used in the 2005 challenge, only SP2, SP3 and SP4 take longer than a few milliseconds for our solver to solve. Thus we generate some difficult random instances for this experiment. First we specify the number of customers, number of products and the average number of customers per product. We then calculate a density that will achieve the specified average number of customers per product. The customer vs product table is then randomly generated using the calculated density to determine when to put 1's and 0's. As a post condition, we throw away any instance where the customer graph can be decomposed into separate components. This is done because we want to compare on instances of a certain size, but if the customer graph decomposes, then the instance degenerates into a number of smaller and relatively trivial instances.

We generate 5 instances for each of the sizes 30×30, 40×40, 50×50, 75×75, 100×100, 125×125, 100×50, 50×100, and average number of customer per product values of 2, 4, 6, 8, 10, for a total of 200 instances.

Ideally, we want to measure speedup by comparing total solve time. However, as mentioned before, the instances that our solver finds challenging are totally unsolvable by the original. Table 4.2 is split into two parts. Above the horizontal line are the instances where the original managed to prove optimality. Here, nodes, time (in milliseconds) and speedup are for the total solve. Below the line the original cannot prove optimality. Here, nodes, time and speedup are for the finding a solution that is at least as good as the solver of [2] could find. The column δOpt shows the average distance this solution is from the optimal. Note that our approach finds and proves the optimal in all cases although the statistics for this are not shown in the table. Time to find an equally good solution is not necessarily a good indication of the speedup achievable for the full solve, as other factors like branching heuristics come into play. However, the trend is quite clear. The original solver is run with its best options. Our MOSP algorithm is run with "better move", "old move" and nogood recording turned on (but no relaxation). Both solvers have a node limit of 2^{25} iterations.

Note that because a single move can close multiple stacks, it is possible to completely solve an instance using fewer nodes than there are customers. This occurs frequently in the high density instances. Thus the extremely low node counts shown here are not errors. The speedup is around 2-3 orders of magnitude for the smallest problems (30×30), and around 5-6 orders of magnitude for the hardest problems that the original version can solve (40×40). The speedup appears to grow exponentially with problem size. We cannot get any speedup numbers for the harder instances since the original cannot solve them. However, given the trend in the speedup, it would not be surprising if the speedup for a full solve on the hardest instances solvable by our solver (100×100) was in the realms of 10^{10} or more.

Next we examine the effect of each of our improvements individually by disabling them one at a time. The three improvements we test here are "better move", "old move" and nogood recording. We use only the instances which are solvable without the improvements and non-trivial, i.e. the 100×100 instances and the easier 125×125 instances. For each improvement, we show the relative slowdown compared to the version with all three optimisations on.

Table 3. Comparing customer search versus [2] on harder random instances. Search is to find the best solution found by [2] with node limit 2^{25}.

	[2]			This paper		
Instance	δOpt	Nodes	Time(ms)	Nodes	Time(ms)	Speedup
30-30-2	0	14318	480	408	4.4	109
30-30-4	0	48232	1981	158	2.0	979
30-30-6	0	89084	2750	56	0.9	3136
30-30-8	0	83558	2010	18	0.4	5322
30-30-10	0	18662	506	8	0.2	2335
40-40-2	0	669384	192917	1472	14.9	12942
40-40-4	0	3819542	227087	556	6.1	36959
40-40-6	0	11343235	625892	217	2.9	218062
40-40-8	0	8379706	334392	49	0.8	403272
40-40-10	0	3040040	98194	20	0.4	229305
50-50-2	0	12356205	1300311	6344	65.8	19758
50-50-4	0.2	5612259	446409	219	2.7	164728
50-50-6	0.2	7949026	510831	45	0.8	636409
50-50-8	0.2	525741	28337	15	0.4	75274
50-50-10	0	16809	784	7	0.2	3411
75-75-2	0.8	2485310	420935	7030	76.8	5484
75-75-4	2.6	3507703	666784	63	1.4	486669
75-75-6	1.2	4412286	756032	59	1.4	548132
75-75-8	1.2	4121046	519778	19	0.6	841336
75-75-10	0.6	1198282	120087	15	0.5	244128
100-100-2	2.2	3008009	765131	481	9.4	81653
100-100-4	4.8	6777140	2017286	145	3.3	619257
100-100-6	4	1269071	347970	39	1.4	241145
100-100-8	4.4	1686045	414456	31	1.1	363468
100-100-10	1.6	4195494	789039	15	0.7	1097494
125-125-2	1.8	7418402	3276210	36672	436.8	7500
125-125-4	3.8	3412379	1559691	916	20.4	76286
125-125-6	6	6076996	2643707	57	2.3	1144180
125-125-8	6.2	942290	321050	28	1.5	217007
125-125-10	3.4	170852	45798	24	1.3	35647
50-100-2	0.2	90076	9971	97	1.6	6290
50-100-4	1	1973322	139300	23	0.6	220776
50-100-6	0.6	1784	116	13	0.4	301
50-100-8	0	97	9	5	0.2	47
50-100-10	0	99	8	3	0.2	39
100-50-2	0	2393401	438220	11117	133.7	3279
100-50-4	0.4	14211006	3499389	183260	1592.3	2198
100-50-6	1.2	5326088	1395417	1569	21.4	65163
100-50-8	0.6	1522796	408908	3506	45.8	8932
100-50-10	1	3594743	906559	524	10.6	85710

As Table 4.2(a) shows, both "better move" and "old move" can produce up to 1 to 2 orders of magnitude speedup on the harder instances. The lower speedups are from instances that are already fairly easy and solvable in seconds. The results from disabling the nogood recording are very interesting. It is known from previous work, e.g. the DP approach of [2] and the CP approach of [5] that nogood recording or equivalent techniques produce several orders of magni-

Table 4. (a) Comparing the effects of each optimisation in turn, and (b) comparing the effects of relaxation.

Instance	Better	Old	Nogood	No relax(ms)	Relax(ms)	Removed	Speedup
100-100-2	25.2	63.7	1.21	603120	370	51.2	1630.6
100-100-4	8.94	5.95	1.01	266205	4798	20.8	55.5
100-100-6	1.90	1.71	0.91	10344	3909	8	2.6
100-100-8	1.54	1.30	0.66	551	712	2.4	0.8
100-100-10	2.96	2.75	1.00	46	94	0.6	0.5
125-125-2	—	—	—	59642993	3284	62.2	18161.8
125-125-4	—	—	—	29768563	251634	24.8	118.3
125-125-6	11.9	3.10	0.98	810678	167384	9	4.8
125-125-8	1.65	1.17	0.98	18781	11978	5	1.6
125-125-10	1.27	0.98	0.64	768	1041	3	0.7
	(a)				(b)		

tude speedup. However, these approaches require (in the worst case) exponential memory usage for the nogood table. It appears however that once we have the "old move" improvement, we can actually turn off nogood recording without a significant loss of performance. In fact, some instances run faster. Thus our "old move" improvement largely subsumes the effect of the huge nogood tables used in the DP [2] or CP [5] approaches and reduces the memory usage from an exponential to a linear amount. The solver of [2] uses up all 2Gb of main memory in ∼5 min with nogood recording. However, our new solver using "old move" pruning uses a constant amount of memory $<$ 2Mb even for 125×125 problems.

4.3 Relaxation

In the following set of experiments, we demonstrate the effectiveness of our relaxation technique. For each of our largest instances, we show in Table 4.2(b) the total runtime (in milliseconds) without relaxation, with relaxation, and the number of customers that was successfully removed without changing the lower bound, as well as the speedup for relaxation. Both versions are run with the customer search strategy, "better move" and "old move" improvements.

As can be seen from the results in Table 4.2(b), relaxation is most effective for sparse instances where we can get up to 3-4 orders of magnitude improvement. There is a slight slowdown for several dense instances but that is because they are trivial to begin with (take $<$ 1s). The sparser the instance, the more customers can be removed without changing the lower bound and the greater the speedup from the relaxation technique. For the hardest instances, 125-125-2, it is often possible to remove some 60-70 of the 125 customers without changing the bound. This reduces the proof of optimality that normally takes 10+ hours into mere seconds. The 125-125-4 instances are now comparatively harder, since we are only able to remove around 25 customers and get a speedup of ∼100. Relaxation is largely ineffective for the denser instances like 125-125-8,10. However, dense instances are naturally much easier to solve anyway, so we have speedup where it is needed the most.

Our relaxation techniques are also useful if we only wish to prove a good lower bound rather than the true lower bound. For example, if we only insist on proving a lower bound that is 5 less than the true optimum, then ∼45 customers

can be removed from the 125-125-4 instances and the bound can be proved in seconds. This is again several orders of magnitude speedup compared to using a normal complete search to prove such a bound. In comparison, although the HAC lower bound heuristic of [1] uses virtually no time, it gives extremely weak lower bounds for the 125-125-4 instances, which are some 30 stacks below the optimum and are of little use.

5 Conclusion

In this paper we show how combining nogood recording with the customer based search strategy of [8] yields a solver that is 5-6 orders of magnitude faster than the previous state of the art MOSP solver. We show how exploiting several dominance relations leads to the the "definite move", "better move" and "old move" improvements. These produce a further 1-2 orders of magnitude improvement. The "old move" improvement in particular is able to subsume the effect of pruning using the extremely large nogood table. This allows us to reduce the memory usage of our solver from an amount exponential in the size of the problem to a constant ∼2Mb. Finally we show how relaxation techniques can be used to speed up the proof of optimality of the hardest instances by another 3-4 orders of magnitude.

Acknowledgments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

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. M. Garcia de la Banda and P.J. Stuckey. Dynamic programming to minimize the maximum number of open stacks. *INFORMS JOC*, 19(4):607–617, 2007.
3. A. Linhares and H.H. Yanasse. Connections between cutting-pattern sequencing, VLSI design, and flexible machines. *Computers & Operations Research*, 29:1759–1772, 2002.
4. S. Prestwich. Increasing solution density by dominated relaxation. *Modelling and Reformulating Constraint Satisfaction Problems, 4th Int. Workshop*, 2005.
5. P. Shaw and P. Laborie. A constraint programming approach to the min-stack problem. In *Constraint Modelling Challenge 2005* [6].
6. B. Smith and I. Gent. Constraint modelling challenge report 2005. `http://www.cs.st-andrews.ac.uk/~ipg/challenge/ModelChallenge05.pdf`.
7. N. Wilson and K. Petrie. Using customer elimination orderings to minimise the maximum number of open stacks. In *Constraint Modelling Challenge 2005* [6].
8. H.H. Yannasse. On a pattern sequencing problem to minimize the maximum number of open stacks. *EJOR*, 100:454–463, 1997.
9. H.H. Yannasse. A note on generating solutions of a pattern sequencing problem to minimize the maximum number of open orders. *Technical Report LAC-002/98, INPE, São José dos Campos, SP, Brazil*, 1998.
10. B.J. Yuen and K.V. Richardson. Establishing the optimality of sequencing heuristics for cutting stock problems. *EJOR*, 84:590–598, 1995.