Dynamic Programming to Minimize the Maximum Number of Open Stacks

Maria Garcia de la Banda  
Clayton School of Information Technology  
Monash University, 6800, Australia, mbanda@csse.monash.edu.au, www.csse.monash.edu.au/~mbanda

Peter J. Stuckey  
NICTA Victoria Laboratory  
Department of Computer Science & Software Engineering  
University of Melbourne, 3010, Australia, pjs@cs.mu.oz.au, www.cs.mu.oz.au/~pjs

In this paper we give a dynamic programming solution to the problem of minimizing the maximum number of open stacks. 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. It was the winner of the 2005 Constraint Modelling Challenge among 13 submissions tackling the open stacks problem.

Key words: dynamic programming, minimization of maximum number of open stacks, cutting sequencing

History:

1. Introduction.

The Minimization of Open Stacks Problem (MOSP) (Yuen and Richardson 1995) can be described as follows. A factory manufactures a number of different products in batches, i.e., a given product needs to be finished before a different product is manufactured. Each customer of the factory places an order requiring one or more different products. Once a 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 freed to use for 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. The problem is known to be NP-hard (Linhares and Yanasse 2002).

We can formalize the problem as follows. Let \( P \) be a set of products, \( C \) a set of customers, and \( c(p) \) a function which 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 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 \( k \). 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 minimization of open stacks problem aims at finding a schedule for manufacturing the products in \( P \) (i.e., a permutation of the products) which minimizes the maximum number of customers active (or of open stacks) at any time.

**Example 1.** Consider an MSOP defined by the set of customers \( C = \{ c1, c2, c3, c4, c5 \} \), the set of products \( P = \{ p1, p2, p3, p4, p5, p6, p7 \} \) 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 \( ci \) has ordered product \( pj \).

Consider the manufacturing schedule given by sequence \( p7 \, p6 \, p5 \, p4 \, p3 \, p2 \, p1 \) and illustrated by the matrix \( M \) shown in Figure 1(b), where client \( ci \) is active at position \( M_{ij} \) if the position contains either an
2. Previous Work.

The open stacks problem is usually thought of as arising from a cutting stock environment, in which it is described as follows: consider a saw machine which is used to cut large pieces into smaller pieces of different sizes. As each large piece is cut, the smaller pieces are arranged in stacks around the saw machine. Only when all the pieces of the same size have been cut can we ship out the stack, and make use of the space to store a stack of different pieces. The aim in the cutting of the large pieces is to minimize the maximum number of open stacks during the cutting.

Yuen (1991, 1995) provides six heuristics for computing an upper bound to the number of open stacks needed. These heuristics favour in one way or another products whose customers are already open stacks and penalize those whose manufacturing results in newly open stacks. Yuen (1995) also noticed that a product for which all its customers are already open should be scheduled immediately in any heuristic, although he does not prove that this must lead to an optimal solution since he only considers heuristics. Finally, Yuen (1995) also proposed a heuristic which arranges the products before applying any other heuristic, in increasing order of the sum of the height of all stacks in which a product was involved (as a measure of the involvement of a product with other products).

Yuen and Richardson (1995) provide two methods to evaluate the optimality of the heuristics presented by Yuen (1991, 1995). The first method simply compares the trivial lower bound (maximum number of customers that have ordered a particular product) with the upper bound obtained by each heuristic (if they coincide, the heuristic is known to have provided the optimal solution). The second method is based on
an exhaustive backtracking search of the tree formed by all possible schedules. The search is reduced by disregarding products that cannot improve the current maximum, selecting first products which have at least one customer in common with the current stacks, sorting the products (and thus their selection) according to the rearrangement defined in Yuen (1995), and comparing the current result against that obtained using the reverse of the current schedule to avoid the search if the current is already greater or equal than the reversed one.

Yannasse (1997) discusses the relationship between MOSP and other problems such as the minimization of the lifetime (or spread) of open stacks (also called pattern allocation or cutting sequencing), or minimization of stack interruptions (a similar problem but not equivalent to the minimization of spread). The paper also proposes a depth-first, greedy branch and bound approach for solving MOSP. The search uses three different lower bounds (including the trivial one previously mentioned) and a trivial upper bound (the number of customers whose orders have not been sent yet) to disregard nodes. The branch and bound sequences the order in which the customers are completed, rather than the order of the products themselves. From this order one can construct an optimal order of the products straightforwardly.

Faggioli and Bentivoglio (1998) a 3 phase approach to solving MOSP problems. Starting from a greedy heuristic solution somewhat similar to that of Yuen (1995), they improve the solution using a tabu search which considers moving one product to another place in the sequence, and they finish by performing an exhaustive backtracking search similar to that of Yuen and Richardson (1995).

Becceneri et al. (2004) provide a new heuristic (Minimal Cost Node) to compute an upper bound on the number of open stacks. The heuristic uses the customer graph obtained from the problem to identify stacks that can be served without involving the opening of many new stacks. While the accuracy of the heuristic is shown to be good, its computational cost is considerable. The paper also improves on the branch and bound search algorithm provided in Yannasse (1997) by detecting equivalent customer nodes and deleting them from the graph until a given product sequence is found.

Our approach differs from those above mainly in two points. Firstly, rather than using either backtracking or branch and bound search techniques to find the optimal solution, we are able to significantly reduce the search by using dynamic programming thanks to a key insight: that the order in which previous products have been manufactured is not important, and thus they can be considered as a set. And secondly, rather than considering the optimization problem as a hole, we decompose it in a sequence of steps, each attempting to check the satisfiability of the problem for a given number of open stacks. As shown in the evaluation section, such an approach significantly improves efficiency.

The results of the Constraint Modelling Challenge (2005) provide 13 different approaches to solving the MOSP problem. We defer comparison to this contemporaneous work until after the experimental results section.

3. Dynamic Programming Formulation.

The MOSP problem is naturally expressible in a dynamic programming formulation. To do so we extend the function $c(p)$ which returns the set of customers ordering product $p \in P$, to handle a set of products $S \subseteq P$. That is, we define $c(S) = \bigcup_{p \in S} c(p)$ as a function which returns the set of customers ordering products from set $S \subseteq P$. Let $A \subseteq P$ denote the set of all products scheduled to be manufactured after product $p$. 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 after $p$ and some scheduled before. It is the fact that $a(p, A)$ does not depend on any particular order of the products in $A$ or $P - A - \{p\}$ that makes the problem amenable to dynamic programming. Let $S \subseteq P$ denote the set of products that still need to be manufactured, i.e., those not yet scheduled, and let $stacks_p(S)$ be the
minimum number of stacks required to schedule the products in $S$. Dynamic programming can be used to define $\text{stacks}_P(S)$ as:

$$\text{stacks}_P(S) = \min_{p \in S} \max\{a(p, S - \{p\}), \text{stacks}_P(S - \{p\})\}$$

which computes, for each product $p$, the maximum number of open stacks needed if $p$ was scheduled first (as the maximum between the number of open stacks $a(p, S - \{p\})$ when $p$ is being manufactured, and the number $\text{stacks}_P(S - \{p\})$ once $p$ is finished), and then obtains the minimum of those. 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 $P$.

### 3.1. Basic $A^*$ algorithm

The code in Figure 2 illustrates our $A^*$ call based dynamic programming algorithm, which improves over a naive formulation by taking into account lower and upper bounds.

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 $\text{stack}[S]$), in which case it returns the previously stored result (code shown in light grey). If not, the algorithm tries to find the product that will lead to the minimum number of open stacks if scheduled first by computing $sp$ the value $\max(a(p, S - \{p\}), \text{stacks}(S - \{p\}, L, U))$ for each $p \in S$, and updating the current minimum in $\text{min}$ if required.

Note, however, that it avoids (thanks to the break) considering products whose active set of customers is greater than or equal to the current minimum $\text{min}$, since they cannot improve on the current solution. As a result, the order in which the $S$ products are tried can significantly affect the amount of work performed by the algorithm. In our algorithm, this order follows a simple heuristic which selects (through the index keyword) the product $p$ with the least number of active customers if scheduled immediately. The loop also stops as soon as the current solution is less than or equal to the lower bound, since we are only interested in finding a single solution less than or equal to the lower bound.

```
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\}) \mid p \in T\}
  T := T - \{p\}
  if (a(p, S - \{p\}) ≥ min) break
  sp := \max(a(p, S - \{p\}), \text{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
```

Figure 2 Pseudo-code for $A^*$ call based dynamic programming algorithm. $\text{stacks}(S, L, U)$ returns the minimal number of open stacks required for scheduling the set of products $S$ given a lower bound on the number of stacks of $L$ and an upper bound of $U$. If there is no schedule less than bound $U$ it returns $U + 1$. 

Let \( p_1, p_4, p_6, p_2 \) be the set of active customers just before an element of \( S \) is scheduled, i.e., those for which some of its products have already been manufactured (those in \( P - S \)), and some have not (those in \( S \)). We can reduce the amount of search performed by the code shown in Figure 2 (and thus improve its efficiency) by noticing that any product ordered by customers whose stacks are already open, can always be scheduled first without affecting the optimality of the solution. In other words, for every \( p \in S \) for which \( c(p) \subseteq o(S) \), there must be a solution to \( \text{stacks}_P(S) \) 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 \( \text{stacks}_P(S) \) beginning with \( p \).

**Proof.** Let \( \Pi \) denote a possibly empty sequence of products. In an abuse of notation, and when clear from the context, we will sometimes use sequences as if they were sets. Take any optimal order \( \Pi_1 / \Pi_2 / \Pi_3 \) of \( S \) in which a product \( p' \) is placed before \( p \), and consider altering the order by moving \( p \) to the front thus obtaining \( p_1 \Pi_1 / p_2 \Pi_2 / \Pi_3 \). We show that the active stacks for each product can only decrease.

First, it is clear that the products in \( \Pi_3 \) have the same active set of customers since the set of products manufactured before and after \( \Pi_3 \) remains unchanged. Second, let us consider the changes in the active set of customers for \( p' \), which can be seen as a general representative of products scheduled before \( p \) in the original order. While in the original order the set of active customers at the time \( p' \) is built is \( a(p', \Pi_2 \Pi_3) = c(p') \cup (c(P - \Pi_2 \Pi_3 - \{p'\}) \cap c(\Pi_2 \Pi_3)) \), in the new order the set of active customers is \( a(p', \Pi_2 \Pi_3) = c(p') \cup (c(P - \Pi_2 \Pi_3 - \{p'\}) \cap c(\Pi_2 \Pi_3)) \). Now, for every set of products \( Q \subseteq Q' \) we know that \( c(Q) \subseteq c(Q') \), i.e., increasing the number of products can only increase the number of customers who ordered them. Thus, we have that \( (a) \ c(\Pi_2 \Pi_3) \subseteq c(\Pi_2 \Pi_3) \). By this and the lemma assumptions we have that \( c(p') \subseteq o(S) \subseteq c(P - S) \subseteq c(P - \Pi_2 \Pi_3 - \{p'\}) \) and, therefore, that \( (b) \ c(P - \Pi_2 \Pi_3 - \{p'\}) = c(P - \Pi_2 \Pi_3 - \{p'\}) \). Hence, by (a) and (b) we have that \( a(p', \Pi_2 \Pi_3) \subseteq a(p', \Pi_2 \Pi_3) \). Finally, 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 \( p_1 \Pi_1 / p_2 \Pi_2 / \Pi_3 \) has a minimal number of stacks. □

**Example 2.** Consider the open stacks problem of Example 1. Let us assume that the set of products \( S = \{p_1, p_2, p_3, p_4, p_6\} \) is scheduled after \( P - S = \{p_5, p_7\} \) have been scheduled. Then, the active customers after \( p_5 \) and \( p_7 \) have been manufactured is \( o(S) = \{c_1, c_4, c_5\} \) and an optimal schedule can begin with \( p_3 \) since \( c(p_3) \subseteq o(S) \). An optimal schedule is shown in Figure 3. □

We can modify the pseudo code of Figure 2 to take advantage of Lemma 1 by adding the line

\[
\text{if } (\exists p \in S. c(p) \subseteq o(S)) \text{ return } \text{stacks}(S - \{p\}, L, U)
\]

before the while loop.

<table>
<thead>
<tr>
<th>( p_7 )</th>
<th>( p_5 )</th>
<th>( p_3 )</th>
<th>( p_1 )</th>
<th>( p_4 )</th>
<th>( p_6 )</th>
<th>( p_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 )</td>
<td>X</td>
<td>X</td>
<td>−</td>
<td>X</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>( c_4 )</td>
<td>X</td>
<td>−</td>
<td>X</td>
<td>−</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>( c_5 )</td>
<td>.</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>.</td>
</tr>
</tbody>
</table>

Figure 3 An optimal schedule for the products \( S = \{p_1, p_2, p_3, p_4, p_6\} \) assuming \( \{p_5, p_7\} \) have already been scheduled.

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 (those sets for which at the end of the computation \( min \) is still set to \( U + 1 \)). We will make use of these sets later.

A call to function \( \text{stacks}(P, L, U) \) returns the minimal number of stacks required to schedule the products in set \( P \) assuming lower bound \( L \) and upper bound \( U \). Extracting the optimal schedule found from the array of stored answers \( stack[] \) is straightforward, and standard for dynamic programming.

### 3.2. Scheduling non-opening products first

Let \( o(S) = c(P - S) \cap c(S) \) be the set of active customers just before an element of \( S \) is scheduled, i.e., those for which some of its products have already been manufactured (those in \( P - S \)), and some have not (those in \( S \)). We can reduce the amount of search performed by the code shown in Figure 2 (and thus improve its efficiency) by noticing that any product ordered by customers whose stacks are already open, can always be scheduled first without affecting the optimality of the solution. In other words, for every \( p \in S \) for which \( c(p) \subseteq o(S) \), there must be a solution to \( \text{stacks}_P(S) \) 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 \( \text{stacks}_P(S) \) beginning with \( p \).

**Proof.** Let \( \Pi \) denote a possibly empty sequence of products. In an abuse of notation, and when clear from the context, we will sometimes use sequences as if they were sets. Take any optimal order \( \Pi_1 / \Pi_2 / \Pi_3 \) of \( S \) in which a product \( p' \) is placed before \( p \), and consider altering the order by moving \( p \) to the front thus obtaining \( p_1 \Pi_1 / p_2 \Pi_2 / \Pi_3 \). We show that the active stacks for each product can only decrease.

First, it is clear that the products in \( \Pi_3 \) have the same active set of customers since the set of products manufactured before and after \( \Pi_3 \) remains unchanged. Second, let us consider the changes in the active set of customers for \( p' \), which can be seen as a general representative of products scheduled before \( p \) in the original order. While in the original order the set of active customers at the time \( p' \) is built is \( a(p', \Pi_2 \Pi_3) = c(p') \cup (c(P - \Pi_2 \Pi_3 - \{p'\}) \cap c(\Pi_2 \Pi_3)) \), in the new order the set of active customers is \( a(p', \Pi_2 \Pi_3) = c(p') \cup (c(P - \Pi_2 \Pi_3 - \{p'\}) \cap c(\Pi_2 \Pi_3)) \). Now, for every set of products \( Q \subseteq Q' \) we know that \( c(Q) \subseteq c(Q') \), i.e., increasing the number of products can only increase the number of customers who ordered them. Thus, we have that \( (a) \ c(\Pi_2 \Pi_3) \subseteq c(\Pi_2 \Pi_3) \). By this and the lemma assumptions we have that \( c(p') \subseteq o(S) \subseteq c(P - S) \subseteq c(P - \Pi_2 \Pi_3 - \{p'\}) \) and, therefore, that \( (b) \ c(P - \Pi_2 \Pi_3 - \{p'\}) = c(P - \Pi_2 \Pi_3 - \{p'\}) \). Hence, by (a) and (b) we have that \( a(p', \Pi_2 \Pi_3) \subseteq a(p', \Pi_2 \Pi_3) \). Finally, 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 \( p_1 \Pi_1 / p_2 \Pi_2 / \Pi_3 \) has a minimal number of stacks. □

**Example 2.** Consider the open stacks problem of Example 1. Let us assume that the set of products \( S = \{p_1, p_2, p_3, p_4, p_6\} \) is scheduled after \( P - S = \{p_5, p_7\} \) have been scheduled. Then, the active customers after \( p_5 \) and \( p_7 \) have been manufactured is \( o(S) = \{c_1, c_4, c_5\} \) and an optimal schedule can begin with \( p_3 \) since \( c(p_3) \subseteq o(S) \). An optimal schedule is shown in Figure 3. □

We can modify the pseudo code of Figure 2 to take advantage of Lemma 1 by adding the line

\[
\text{if } (\exists p \in S, c(p) \subseteq o(S)) \text{ return } \text{stacks}(S - \{p\}, L, U)
\]

before the while loop.
3.3. Looking ahead

We can further reduce the search performed by \textit{stacks}(S,L,U) by computing a lower bound to the number of stacks required to schedule the products in \( S \) based on looking ahead to see how many stacks will be needed to close the already opened stacks. Let us define the \textit{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, 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 \( a_G(c) \) be the set of nodes adjacent to \( c \) in \( G \).

\textbf{Lemma 2.} The minimal number of stacks required for a set of products \( S \) is at least \( b(S) = |o(S)| + \min\{|a_G(c) - o(S)| \mid c \in e(S)\} \).

\textit{Proof.} Before the products in \( S \) are scheduled, the open stacks are \( o(S) \). Consider first an active customer \( c \in o(S) \). In order to close \( c \) we need to have open stacks for all customers \( c' \in a_G(c) \) adjacent to \( c \) (including \( c \)), since \( c \) and \( c' \) share some product \( p \) which needs to be completed before we can close \( c \). This at least requires us to open the customers in \( a_G(c) - o(S) \) which are not already open. Similarly, consider a non-active customer \( c \in c(S) - o(S) \). We could also close \( c \) simply by opening all the customers in \( a_G(c) - o(S) \), but that would not allow us to close any customer currently open. Hence, the bound holds. \( \square \)

\textbf{Example 3.} Consider the customer graph for Example 1 which is shown in Figure 4, and consider scheduling the set \( S = \{ p_2, p_3, p_4, p_5, p_6, p_7 \} \) after \( \{ p_1 \} \). The open stacks for \( S \) are \( o(S) = \{ c_1, c_2 \} \). The adjacent non-open customers to each customer \( c_1, c_2, c_3, c_4 \) and \( c_5 \) are respectively \( \{ c_4, c_5 \}, \{ c_3, c_4 \}, \{ c_3, c_4, c_5 \} \) and \( \{ c_4, c_5 \} \). Hence \( b(S) = |\{ c_1, c_2 \}| + 2 = 4 \). This is a lower bound on a schedule for \( S \), since closing any customer requires at least this many open stacks.

If we consider scheduling \( S = \{ p_1, p_2, p_3, p_4, p_5, p_6 \} \) after \( \{ p_7 \} \) we have that \( o(S) = \{ c_1, c_4 \} \) and the adjacent non-open customers to each customer \( c_1, c_2, c_3, c_4 \) and \( c_5 \) are respectively \( \{ c_2, c_5 \}, \{ c_1, c_2, c_3 \}, \{ c_2, c_3 \}, \{ c_2, c_3, c_5 \} \) and \( \{ c_5 \} \). Hence \( b(S) = 3 \). \( \square \)

We can use this lower bound to improve the \( A^* \) programming algorithm given in Figure 2 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. Note that using \( a'(p,A) \) rather than \( a(p,A) \) can significantly reduce the search space.

\textbf{Example 4.} Consider the problem of Example 1. In the initial where \( S = \{ p_1, p_2, p_3, p_4, p_5, p_6, p_7 \} \), while the calculation of \( a(p,S - \{ p \}) \) gives 2, 1, 2, 3, 2, 2 respectively for \( p = p_1, p_2, p_3, p_4, p_5, p_6, p_7 \), the calculation of \( a'(p,S - \{ p \}) \) gives 4, 3, 3, 3, 3, 3, 3. Hence, the code will try one of \( p_2, p_3, p_4, p_5, p_6, p_7 \) as the first scheduled product before it tries \( p_1 \). Since this will determine an overall schedule requiring 3 stacks it will never examine any schedules that commence with \( p_1 \) or indeed any of the other products since they cannot improve the schedule found already. \( \square \)

4. 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.

\begin{figure}[h]
\centering
\includegraphics{example1_graph.png}
\caption{Customer graph for the problem of Example 1.}
\end{figure}
Solving the reduced problem gives an optimal value for $P$. Optimal solutions to the reduced problem can be extended to give optimal solutions to $P$ by simply placing each $p$ immediately after any of the $p'$s that subsumed it.

This preprocessing step, which was also noted in Becceneri et al. (2004), can be proved by using Lemma 1. Simply note that if $c(p) \subseteq c(p')$, then any order for $S$ that includes $p'$ but not $p$ must have $c(p) \subseteq o(S)$. Because the problem is the same when considering the reverse order, the same holds for orders in which $p$ appears immediately before $p'$.

**Example 5.** Consider the open stacks problem from Example 1. Since $c(p2) \subseteq c(p4)$ and $c(p6) \subseteq c(p4)$, both $p2$ and $p6$ can be removed. Inserting them after $p4$ in any optimal order for the reduced set of products, gives an optimal order for the original problem. This is illustrated in Figure 5.\[\blacksquare\]

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 Yuen and Richardson (1995). Although we thought this was too unrealistic to occur, it does occur in several benchmarks, including some of the mildly difficult ones.

5. **Bounds.**

Our $A^*$ programming algorithm uses both upper and lower bounds on the number of open stacks needed to solve $P$ to reduce the number of subsets visited. As mentioned before, trivial lower and upper bounds are $L = \max\{|c(p)| \mid p \in P\}$ (the maximum number of customers who ordered the same product) and $U = |C|$ (the total number of customers).

Several authors have considered how to improve the lower bound. In particular, the following Lemmas define lower bounds previously used in the literature:

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

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

Given an open stacks problem where $P$ is the set of products, $C$ the set of customers, and $c(p)$ the set of customers who ordered product $p \in P$, a minor $c'(p)$ can be obtained in two ways. One way is to remove an entire customer $c \in C$. This is achieved by, for each $p \in P$, replacing $c(p)$ by $c'(p) = c(p) - \{c\}$. The other way is by merging two adjacent customers, i.e. two customers for which there exists a $p \in P$ such that $c_1, c_2 \in c(p)$. This is achieved by replacing $c(p)$ by $c'(p) = (c(p) - \{c_2\}) \cup \{c_1\}$ if $c_2 \in c(p)$, or by $c'(p) = c(p)$ otherwise. Given the customer graph $G = (V, E)$ of the open stacks problem, these two operations correspond to an edge contraction and a node elimination from $G$, respectively.

**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 $m' \leq m$.

Becceneri et al. (2004) define a heuristic arc contraction approach (HAC) based on the Lemmas 4 and 5. In particular, they apply any number of minor steps and use the size of the minimum degree node as a lower bound for the original problem. Note that this process can be stopped as soon as the remaining customer graph is a clique $Q$, since by Lemma 3 its minimum number of open stacks is $|Q|$. We built an
heuristic($S$)
min := 0
while ($S \neq \emptyset$)
  heuristically select $p$
  $S := S - \{p\}$
  if ($a(p, S - \{p\}) > \text{min}$) min := $a(p, S - \{p\})$
return min

Figure 6 (Meta) Pseudo-code for greedy heuristics.

implementation of Becceneri et al.’s algorithm and a greedy clique finder that doesn’t do contractions but tries to find maximal cliques starting from each product set of customers.

Let us now consider how to improve the upper bound. For this we experimented with eleven greedy heuristics (whose general form is shown in Figure 6) which at each stage of computation $S$ select a product $p$ according to different criteria. The five most successful heuristics over all benchmark instances we found were:

(A) This heuristic (defined by Yuen (1995) as heuristic 3) favours products who supply many already active stacks and do not open new stacks. In other words, it selects the product $p$ which, if scheduled immediately, maximizes the number of previously active stacks requiring $p$ minus the number of stacks made active by $p$:

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

(B) This heuristics selects the product $p$ which, if scheduled immediately, minimizes the number of active stacks $a(p, S - \{p\})$ and breaks ties in favor of products that close a greater number of stacks (are the last product in those stacks) $|c(p) - c(S - \{p\})|:

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

(C) This heuristic is similar to the one above except for the fact that all active stack numbers less than the current min are considered equivalent:

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

(D) This heuristic is similar to (B) but breaks ties by maximizing a cost given by $\sum_{c \in c(p)} 2^{-n(S,c)}$ 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. As a result, products which initiate or are close to the activation of the stack are not favoured, while those which are near the end or actually close the stack, are favoured.

$\text{index min}_{p \in S}(a(p, S - \{p\}), -\sum_{c \in c(p)} 2^{-n(S,c)})$

(E) This heuristic selects the product $p$ which, if scheduled immediately, minimizes the maximum of the number of active stacks required using the improved formula $a'(p, A) = \max(\{a(p, A), b(A)\})$.

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

We also implemented the minimal cost node heuristic (which we’ll denote (F)) of Becceneri et al. (2004) which does not follow the general greedy format of Figure 6 since it selects arcs (not products) in the customer graph to determine a product order.

As mentioned in the Introduction, our A* programming algorithm is particularly effective when called with $L = U = n$, where it only explores schedules which use exactly $n$ active stacks. This is almost certainly related to the fact that the problem is fixed parameter tractable (Linhares and Yannasse 2002), that is if we fix $n$ there is a polynomial time algorithm for the decision problem. This immediately suggests an extended search procedure (which we will call stepwise($L, U$)) that successively tries each possible value from the lower $L$ to the upper $U$ bound. The code in Figure 7 implements this approach. The loop stops as soon as a solution $min$ for stacks($P, try, try$) that is equal than $try$ is found, since that is known to be the optimal value. Note that $min$ cannot be less than $try$ since we are going upwards from the lower bound. If $min$ is greater than $try$, before the next value is tried, we must reset the stack[$S$] value of any $S \in FAIL$, i.e., the value of those $S$ for which no solution was found equal than $try$ and, therefore, were set by stacks($S, try, try$) to stack[$S$] = $try + 1$.

The code in Figure 8 modifies the stepwise search by using binary search. The code repeatedly tries to find a solution using the midpoint of the current range as $try$. If no solution lower or equal than $try$ is found, it iterates using as new range the values above $try$ after (as for the stepwise search) resetting the value of the sets $S \in FAIL$. If, on the other hand, a solution is found, it attempts to find a better solution using the range below. To do this it must first reset all stack[$S$] computations ($S \in FAIL \cup SUCCESS$) performed for the current iteration stacks($P, try, try$) (but not those previously calculated and used by this computation), since they could be too high or too low.

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. The code is illustrated in Figure 9. The procedure corresponds to the backstep in the binarychop procedure. The backwards search procedure has another advantage: we can stop at any time with a (non-optimal) solution. Note that since
stacks can actually return a value smaller than try we do not just decrease the try by one each time, but to under the minimum we last found.

\[
\text{backwards}(L, U) \\
\text{try} := U \\
\text{while} (\text{try} \geq L) \\
\text{FAIL} := \text{SUCCESS} := \emptyset \\
\text{min} := \text{stacks}(P, \text{try}, \text{try}) \\
\text{if} (\text{min} > \text{try}) \text{return} \ gmin \\
\text{gmin} := \text{min} \\
\text{try} := \text{min} - 1 \\
\text{for} (S \in \text{FAIL} \cup \text{SUCCESS}) \text{stack}[S] := 0
\]

Figure 9 Pseudo-code for backwards stepwise search for optimal solution.

7. Experimental Results.

We tested our approach on all the problem instances provided for the Modelling Challenge. All experiments were run on a Pentium IV 3.4Ghz with 2GB RAM running Linux Fedora Core 3. 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. The dynamic programming software was compiled with gcc 3.4.2 using -O3. Timings are calculated as the sum of user and system time given by getrusage, since 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.

7.1. The Challenge Instances to MOSP

The instances provided for the Modelling Challenge were divided into two categories depending on whether their results were requested individually for an instance, or in aggregate form for a collection of MOSP instances. Table 1 provides results on aggregate instances, i.e., each of its files consists of a collection of MOSPs. The nomenclature of the files is that used in Modelling Challenge (2005), where the suffix of the filename \( n \_m \) indicates instances in the suite have \( n = |C| \) customers and \( m = |P| \) products. The runs for aggregate files are performed using all \( A^* \) improvements, all preprocessing steps, all lower and upper bounds heuristics (using the best value found), and the backwards stepwise search approach. The timing results for each file are aggregates over 10 runs of each individual benchmark in the suite.

For each file we give the mean best solution, the total time (in milliseconds) per instance as mean, median and maximum, the search (calls to \text{stacks} that reach the while loop) required to find the optimal, and the search required to find and prove optimality (the optimal value was found for every instance in this table). Since the program is deterministic the search results are the same on each run of a benchmark.

The measure of search effort is the number of calls to \text{stacks} that do not immediately return, because of cache hit, \( S = \emptyset \) or the definite choice optimization of Lemma 1. 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. 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)

One can see that most aggregate instances are quite easy. The median search effort required to find the optimal is never greater than 30 which basically means that, often, one of the first schedules tried is optimal. Note that sometimes we can require less calls to \text{stacks} than there are products to prove optimality, this is because of preprocessing to remove redundant products. Clearly, some individual examples are much more difficult, and these dominate the results for that file. Only wbo\_30\_30 and wbop\_30\_30 have a significant number of difficult instances.
The second set of results are shown in Table 2, where the remaining instances are run individually under the same conditions as those in Table 1. Here we show the name of the instance instance, its number of clients and products, the best value found, whether the best value found is proved optimal or not, its total time (in milliseconds) as the average over ten runs, the search effort to find the optimal, and the total search effort to prove optimality.

Our program found the optimal solutions for all instances except SP2, SP3 and SP4 which hit the search limit (set at \(2^{33} = 33554432\) calls to \texttt{stacks}). Again only a few of the instances are difficult, in particular

<table>
<thead>
<tr>
<th>File</th>
<th>mean</th>
<th>best</th>
<th>Total time per instance (ms)</th>
<th>Search to find optimal</th>
<th>Total search effort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>med.</td>
<td>max</td>
<td>mean</td>
<td>med.</td>
</tr>
<tr>
<td>problem_10_10</td>
<td>8.03</td>
<td>0.09</td>
<td>0</td>
<td>2</td>
<td>7.01</td>
</tr>
<tr>
<td>problem_10_20</td>
<td>8.92</td>
<td>0.13</td>
<td>0</td>
<td>4</td>
<td>10.45</td>
</tr>
<tr>
<td>problem_15_15</td>
<td>12.87</td>
<td>0.37</td>
<td>0</td>
<td>7</td>
<td>13.75</td>
</tr>
<tr>
<td>problem_15_30</td>
<td>14.02</td>
<td>2.49</td>
<td>1</td>
<td>43</td>
<td>25.17</td>
</tr>
<tr>
<td>problem_20_10</td>
<td>15.88</td>
<td>0.52</td>
<td>0</td>
<td>4</td>
<td>12.99</td>
</tr>
<tr>
<td>problem_20_20</td>
<td>17.97</td>
<td>5.83</td>
<td>1</td>
<td>86</td>
<td>29.31</td>
</tr>
<tr>
<td>problem_30_10</td>
<td>23.95</td>
<td>1.63</td>
<td>1</td>
<td>9</td>
<td>17.59</td>
</tr>
<tr>
<td>problem_30_15</td>
<td>25.97</td>
<td>7.48</td>
<td>3</td>
<td>46</td>
<td>35.45</td>
</tr>
<tr>
<td>problem_30_30</td>
<td>28.32</td>
<td>718.36</td>
<td>3</td>
<td>10074</td>
<td>999.80</td>
</tr>
<tr>
<td>problem_40_20</td>
<td>36.38</td>
<td>96.89</td>
<td>8</td>
<td>607</td>
<td>90.69</td>
</tr>
<tr>
<td>Shaw_20_20</td>
<td>13.68</td>
<td>12.43</td>
<td>11</td>
<td>42</td>
<td>45.80</td>
</tr>
<tr>
<td>wbo_10_10</td>
<td>5.92</td>
<td>0.14</td>
<td>0</td>
<td>1</td>
<td>9.82</td>
</tr>
<tr>
<td>wbo_10_20</td>
<td>7.35</td>
<td>0.33</td>
<td>0</td>
<td>6</td>
<td>20.27</td>
</tr>
<tr>
<td>wbo_10_30</td>
<td>8.20</td>
<td>1.27</td>
<td>0</td>
<td>21</td>
<td>26.32</td>
</tr>
<tr>
<td>wbo_15_15</td>
<td>9.35</td>
<td>1.11</td>
<td>1</td>
<td>7</td>
<td>15.57</td>
</tr>
<tr>
<td>wbo_15_30</td>
<td>11.58</td>
<td>28.34</td>
<td>2</td>
<td>213</td>
<td>85.82</td>
</tr>
<tr>
<td>wbo_20_10</td>
<td>12.90</td>
<td>0.51</td>
<td>0</td>
<td>3</td>
<td>11.74</td>
</tr>
<tr>
<td>wbo_20_20</td>
<td>13.69</td>
<td>8.72</td>
<td>7</td>
<td>42</td>
<td>38.48</td>
</tr>
<tr>
<td>wbo_30_10</td>
<td>20.05</td>
<td>1.78</td>
<td>2</td>
<td>5</td>
<td>14.52</td>
</tr>
<tr>
<td>wbo_30_15</td>
<td>20.96</td>
<td>8.08</td>
<td>7</td>
<td>30</td>
<td>28.54</td>
</tr>
<tr>
<td>wbo_30_30</td>
<td>22.56</td>
<td>1108.10</td>
<td>306</td>
<td>8686</td>
<td>608.63</td>
</tr>
<tr>
<td>wbo_10_10</td>
<td>6.75</td>
<td>0.10</td>
<td>0</td>
<td>1</td>
<td>9.82</td>
</tr>
<tr>
<td>wbo_10_20</td>
<td>8.07</td>
<td>0.56</td>
<td>0</td>
<td>8</td>
<td>21.02</td>
</tr>
<tr>
<td>wbo_10_30</td>
<td>8.55</td>
<td>1.07</td>
<td>1</td>
<td>25</td>
<td>28.98</td>
</tr>
<tr>
<td>wbo_15_15</td>
<td>10.37</td>
<td>0.77</td>
<td>0</td>
<td>6</td>
<td>15.25</td>
</tr>
<tr>
<td>wbo_15_30</td>
<td>12.15</td>
<td>18.63</td>
<td>2</td>
<td>197</td>
<td>162.52</td>
</tr>
<tr>
<td>wbo_20_10</td>
<td>14.28</td>
<td>0.49</td>
<td>0</td>
<td>3</td>
<td>11.82</td>
</tr>
<tr>
<td>wbo_20_20</td>
<td>14.87</td>
<td>7.99</td>
<td>2</td>
<td>58</td>
<td>40.24</td>
</tr>
<tr>
<td>wbo_30_10</td>
<td>22.48</td>
<td>1.21</td>
<td>1</td>
<td>5</td>
<td>10.78</td>
</tr>
<tr>
<td>wbo_30_15</td>
<td>22.38</td>
<td>7.50</td>
<td>5</td>
<td>38</td>
<td>25.13</td>
</tr>
<tr>
<td>wbo_30_30</td>
<td>23.84</td>
<td>986.12</td>
<td>87</td>
<td>8770</td>
<td>1113.31</td>
</tr>
<tr>
<td>wbp_10_10</td>
<td>7.28</td>
<td>0.08</td>
<td>0</td>
<td>1</td>
<td>7.60</td>
</tr>
<tr>
<td>wbp_10_20</td>
<td>8.71</td>
<td>0.17</td>
<td>0</td>
<td>2</td>
<td>11.93</td>
</tr>
<tr>
<td>wbp_10_30</td>
<td>9.31</td>
<td>0.20</td>
<td>0</td>
<td>2</td>
<td>14.03</td>
</tr>
<tr>
<td>wbp_15_15</td>
<td>11.05</td>
<td>0.56</td>
<td>0</td>
<td>6</td>
<td>13.82</td>
</tr>
<tr>
<td>wbp_15_30</td>
<td>13.09</td>
<td>5.02</td>
<td>1</td>
<td>60</td>
<td>28.75</td>
</tr>
<tr>
<td>wbp_20_10</td>
<td>15.12</td>
<td>0.52</td>
<td>0</td>
<td>3</td>
<td>11.97</td>
</tr>
<tr>
<td>wbp_20_20</td>
<td>15.41</td>
<td>7.26</td>
<td>2</td>
<td>83</td>
<td>50.58</td>
</tr>
<tr>
<td>wbp_30_10</td>
<td>23.18</td>
<td>2.03</td>
<td>2</td>
<td>8</td>
<td>23.68</td>
</tr>
<tr>
<td>wbp_30_15</td>
<td>22.98</td>
<td>10.56</td>
<td>7</td>
<td>45</td>
<td>57.48</td>
</tr>
<tr>
<td>wbp_30_30</td>
<td>24.46</td>
<td>1203.78</td>
<td>5</td>
<td>17097</td>
<td>2071.78</td>
</tr>
</tbody>
</table>
the SP benchmarks which were specifically designed to have large path width. Our code 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 calculated using lower bound heuristics and using stepwise are 18, 15 and 22 respectively.

### 7.2. The Effect of the Optimizations.

In this subsection we briefly describe the effect of the preprocessing approaches, lower and upper bounds approaches and searching approaches. We use as baseline the backwards search methodology with all improvements, i.e., the definite choices of Lemma 1, the improved lower bounds calculation ($a'(p, A)$), and the preprocessing and global bounds improvements.

Table 3 compares 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 for backwards with some optimizations disabled individually.

Regarding the different search strategies, the backwards strategy is a clear winner (regardless of whether we consider search effort, or runtime), followed by the binarychop and stepwise search strategies, in that order. As mentioned before, this is due to the fact that the most expensive stack number to try is often the stack number below the optimal, and those above the optimal were usually easier than those below.

Regarding the effect of individual optimizations on the backwards strategy, it is clear that 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 have not 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.
Table 3 Comparative results over the entire benchmark suite.

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

Table 4 Comparison of the heuristics over the 5803 benchmarks.

<table>
<thead>
<tr>
<th>heur</th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
<th>(E)</th>
<th>(F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>best</td>
<td>3046</td>
<td>3596</td>
<td>3615</td>
<td>3840</td>
<td>5073</td>
<td>5446</td>
</tr>
<tr>
<td>unique</td>
<td>29</td>
<td>1</td>
<td>7</td>
<td>43</td>
<td>159</td>
<td>514</td>
</tr>
<tr>
<td>optimal</td>
<td>2733</td>
<td>3231</td>
<td>3248</td>
<td>3458</td>
<td>4658</td>
<td>4986</td>
</tr>
<tr>
<td>sum</td>
<td>98167</td>
<td>96798</td>
<td>96769</td>
<td>96462</td>
<td>94592</td>
<td>94093</td>
</tr>
</tbody>
</table>

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 non redundant 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 than a tiny fraction. Comparatively, the heuristics rank in the order (A) to (F) (worst to best). Of 5964 partitions of products for 5803 instances, Table 4 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 instance (of 5803), and the total sum of the heuristic results is shown. Clearly the minimal cost node heuristic of Beccheneri et al. (2004) is the best, but using multiple heuristics can still substantially improve the result, since they all provide the unique best answer in at least one case.

Although the lower bounds approaches are very successful at finding good lower bounds, the only time they can improve the backwards search 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 bettered by the clique approach. The clique lower bound gives the optimal answer in 2718 benchmarks of 5803, while the HAC approach gives the optional on 3380.

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

7.3. Comparing with Other Results in the Competition.

A full report of the competition is available from the competition website Modelling Challenge (2005). Here we briefly compare our results to the other 12 entries. Our approach solved more instances than any other
entry except that of Shaw and Laborie who solved exactly the same set (everything except SP2, SP3 and SP4). Their approach is a constraint programming approach with tabling which is quite similar to a dynamic programming approach. They used the definite choice optimization of Lemma 1, clique lower bounds and developed a more complex search strategy based on splitting products into early and late sets and solving scheduling these problems independently. Finally, they also explored around every better solution found using a local search technique to find good solutions early. Overall, our run times were around 2 orders of magnitude faster than theirs, and the search space explored 2 to 10 times less.

The competition had another dynamic programming entry, which more or less implemented the direct definition of $\text{stacks}_P(S)$ given at the beginning of section 3, using a bottom-up dynamic programming approach which requires all $2^{|P|}$ subproblems to be solved. As we can see from the median calls to $\text{stacks}$ required, this is very wasteful for MOSP. This entry solved all instances with less than 30 products optimally, about 2 to 10 times slower than our solution. It did not attempt the larger instances (because of the space required).

No entry in the competition provided an optimal solution for SP2, SP3 and SP4 while our best solutions for them were bettered three times: The local search method of Truchet, Bourdon and Codognet found a solution of 55 for SP4, while the heuristic solution construction method of Miller, based on reasoning over the customer graph, found solutions of 35 for SP3 and 54 for SP4.

The competition entries included a mixed integer programming model which was unable to solve most instances; a number of local search methods which often gave quite good results; and many constraint programming models, which were usually unable to solve the difficult instances in wbo 30_30, wbop 30_30, Miller 20_40 (and of course SP2, SP3 and SP4). Overall, our results are usually 2 orders of magnitude faster than any other entrant except the other dynamic programming solution, and require an order of magnitude less search (although the comparison of different measures of “search” makes this less meaningful).

There are many fascinating results, theorems and models contained in the challenge report, and no doubt many of the techniques could be tried out with our dynamic programming formulation.

8. Conclusion.

The call-based $A^*$ dynamic programming formulation of minimizing open stacks give a very effective algorithm for solving these problems. It can be improved by searching backwards for a minimal solution, probably because of the fact that the problem is fixed parameter tractable. While we have experimented with many different optimizations, the key improvements we discovered were always scheduling subsumed products immediately (Lemma 1) and using $d'(p, A)$ to get a better lower bound on rest of the schedule. There is certainly scope for improving the dynamic programming approach, particularly by better dynamic lower bounding using the customer graph.

Acknowledgments

We would like to thank the organizers of the Constraint Modelling Challenge 2005, Barbara Smith and Ian Gent, for their hard work in providing a really interesting challenge. We would also like to thank Thierry Benoist for pointing out an inaccurate description of $b(S)$ in our original version of this paper.

References


