This article was downloaded by: [144.214.88.242] On: 18 September 2017, At: 20:08
Publisher: Institute for Operations Research and the Management Sciences (INFORMS)
INFORMS is located in Maryland, USA

INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Dynamic Programming to Minimize the Maximum Number of Open Stacks

Maria Garcia de la Banda, Peter J. Stuckey,

To cite this article:
Maria Garcia de la Banda, Peter J. Stuckey, (2007) Dynamic Programming to Minimize the Maximum Number of Open Stacks. INFORMS Journal on Computing 19(4):607-617. https://doi.org/10.1287/ijoc.1060.0205

Full terms and conditions of use: http://pubsonline.informs.org/page/terms-and-conditions

This article may be used only for the purposes of research, teaching, and/or private study. Commercial use or systematic downloading (by robots or other automatic processes) is prohibited without explicit Publisher approval, unless otherwise noted. For more information, contact permissions@informs.org.

The Publisher does not warrant or guarantee the article's accuracy, completeness, merchantability, fitness for a particular purpose, or non-infringement. Descriptions of, or references to, products or publications, or inclusion of an advertisement in this article, neither constitutes nor implies a guarantee, endorsement, or support of claims made of that product, publication, or service.

Copyright © 2007, INFORMS

Please scroll down for article—it is on subsequent pages

INFORMS is the largest professional society in the world for professionals in the fields of operations research, management science, and analytics.
For more information on INFORMS, its publications, membership, or meetings visit http://www.informs.org

RIGHTSLINK

INFORMS Journal on Computing
Vol. 19, No. 4, Fall 2007, pp. 607–617
ISSN 1091-9856 | EISSN 1526-5528 | 07 | 1904 | 0607

DOI 10.1287/ijoc.1060.0205
© 2007 INFORMS

Dynamic Programming to Minimize the Maximum Number of Open Stacks

Maria Garcia de la Banda
Clayton School of Information Technology, Monash University,
Melbourne, Victoria 3800, Australia, mbanda@csse.monash.edu.au

Peter J. Stuckey
National ICT Australia (NICTA) Victoria Research Laboratory, Department of Computer Science and
Software Engineering, University of Melbourne, Victoria 3010, Australia, pjs@cs.mu.oz.au

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 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.

Key words: dynamic programming; minimization of maximum number of open stacks; cutting sequencing
History: Accepted by John N. Hooker Jr., Area Editor for Constraint Programming and Optimization; received March 2006; revised August 2006; accepted August 2006. Published online in *Articles in Advance* September 28, 2007.

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

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., 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, and 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 integrated circuit 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 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 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.

Example 1. Consider an MSOP defined by the set of customers $C = \{c_1, c_2, c_3, c_4, c_5\}$, the 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.

RIGHTSLINK

	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)

Figure 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 –**

The MOSP was chosen as the subject of the first Constraint Modelling Challenge posed in May 2005. This paper presents the methodology and results developed by the winning entry. The two main contributions of this paper are as follows. First, it shows how the characteristics of the problem make it naturally expressible in a dynamic-programming formulation. This is crucial since it allows us to reduce the raw search space from $|P|!$ (i.e., every possible permutation of the products) to $2^{|P|}$. Second, this paper shows how further search reductions can be obtained by breaking the optimization problem into a sequence of satisfaction problems, each of which asks whether there is a sequence with exactly N open stacks. (This may be related to the fact that the problem is *fixed parameter tractable*, Linhares and Yanasse 2002.)

In the next section we discuss previous work on the MOSP. In Section 3 we give our call-based $A^\star$ dynamic-programming formulation for the MOSP. Section 4 presents two different preprocessing steps that can significantly reduce the complexity of the problem. Section 5 examines how to compute several upper and lower bounds for an MOSP. Section 6 investigates different search methods, and Section 7 presents an experimental evaluation of the different approaches. In Section 8 we conclude.

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 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 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 is to order the cutting of the smaller piece types 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 favor 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 considers only heuristics. Finally, Yuen (1995) also proposed a heuristic that rearranges 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 that 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 than or equal to the reversed one.

Yannasse (1997) discusses the relationship between the 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 the 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) present a three-phase approach to solving the MOSP. Starting from a greedy heuristic solution somewhat similar to that of Yuen (1995), they improve the solution using a tabu search that 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).

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

RIGHTSLINK

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

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 described mainly in two points. First, rather than using either backtracking or branch-and-bound search techniques to find the optimal solution, we reduce the search significantly by using dynamic programming thanks to a key insight: The order in which previous products have been manufactured is not important, and thus they can be considered as a set. Second, rather than considering the optimization problem as a whole, 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. We defer comparison to this contemporaneous work until after the experimental results section.

3. Dynamic-Programming Formulation

The MOSP 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 that returns the set of customers ordering products from set $S \subseteq P$. Let $a(p, A)$, where $A \subset P - \{p\}$, denote the set of active customers at the time product p is manufactured assuming that the set A of products is manufactured after P and $P - A - \{p\}$ is manufactured before p. Then $a(p, A) = c(p) \cup (c(A) \cap c(P - A - \{p\}))$, i.e., the active customers are those who ordered p, plus those whose orders include some products scheduled after p and some scheduled before. That $a(p, A)$ does not depend on any particular order of the products in A or $P - A - \{p\}$ 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*(S) be the minimum number of stacks required to schedule the products in S. Dynamic programming can be used to define *stacks*(S) as

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

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 *stacks*$(S - \{p\})$ 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 $A^\star$ Algorithm

The code in Figure 2 illustrates our $A^\star$ call-based dynamic-programming algorithm, which improves over a naïve 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 *stack*$[S]$), in which case it returns the previously stored result (code shown in light gray). If not, the algorithm tries to find the product that will lead to the minimum number of open stacks if scheduled first by computing in *sp* the value $\max(a(p, S - \{p\}), \mathsf{stacks}(S - \{p\}, L, U))$ for each $p \in S$, and updating the current minimum in *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 *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 that selects the product p

```
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_{p∈T} a(p, S − {p})
    T := T − {p}
    if (a(p, S − {p}) ≥ min) break
    sp := max(a(p, S − {p}), stacks(S − {p}, L, U))
    if (sp < min) min := sp
  stack[S] := min
  if (min > U) FAIL := FAIL ∪ {S}
  else SUCCESS := SUCCESS ∪ {S}
  return min
```

Figure 2 **Pseudocode for $A^\star$ Call-Based Dynamic-Programming Algorithm**

Notes. stacks(S, L, U) returns the minimum 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 $U + 1$.

RIGHTSLINK

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

with the fewest active customers if scheduled immediately. The index construct index $\min_{p\in S} e(p)$ returns the p in S that causes the expression $e(p)$ to take its minimum value (it is similarly defined for max). 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.

The dark gray code stores in *SUCCESS* the sets that resulted in finding a solution within the bounds, and in *FAIL* those that 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 stacks(P, L, U) returns the minimum 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 Nonopening 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 *stacks*(S) that starts with p.

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

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

First, it is clear that the products in Π_3 have the same active set of customers since the set of products manufactured before and after Π_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 $a(p', \Pi_2 p \Pi_3) = c(p') \cup (c(P - \Pi_2 p \Pi_3 - \{p'\}) \cap c(\Pi_2 p \Pi_3))$, 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) $c(\Pi_2 p \Pi_3) \subseteq c(\{p'\} \cup \Pi_2 p \Pi_3)$. By this and the lemma assumptions we have that $c(p) \subseteq o(S) \subseteq c(P-S) \subseteq c(P - \Pi_2 p \Pi_3 - \{p'\})$ and, therefore, that (b) $c(P - \Pi_2 p \Pi_3 - \{p'\}) = c(P - \Pi_2 \Pi_3 - \{p'\})$. Hence, by (a) and (b) we have that $a(p', \Pi_2 \Pi_3) \subseteq a(p', \Pi_2 p \Pi_3)$. 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 $p\Pi_1 p' \Pi_2 \Pi_3$ has a minimum number of stacks. □

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

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

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

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

before the **while** loop.

3.3. Looking Ahead

We can further reduce the search performed by 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 *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 $N(c)$ be the set of nodes adjacent to c in G.

Lemma 2. *The minimum number of stacks required for a set of products S is at least $b(S) = |o(S)| + \min\{|N(c) - c(P-S)| \mid c \in c(S)\}$.*

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

RIGHTSLINK

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

stacks for all customers $c' \in N(c)$ adjacent to c (including c), since c and c' share some product p that needs to be completed before we can close c. This at least requires us to open the customers in $N(c) - c(P - S)$ that have not already been opened $c(P - S)$. Similarly, consider a nonactive customer $c \in c(S) - o(S)$. We could also close c simply by opening all the customers in $N(c) - c(P - S)$, but that would not allow us to close any customer currently open. Hence, the bound holds. □

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 nonopened customers to each customer c_1, c_2, c_3, c_4, and c_5 are respectively $\{c_4, c_5\}$, $\{c_3, c_4\}$, $\{c_3, c_4\}$, $\{c_3, c_4, c_5\}$, 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 nonopened 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 $b(S) = 3$.

We can use this lower bound to improve the $A^\star$ 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.

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, 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.

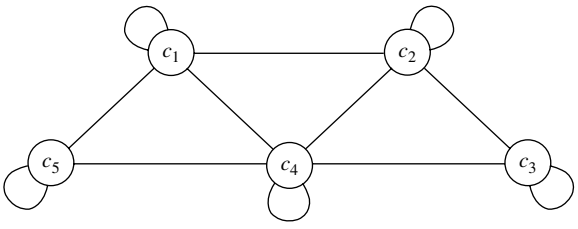

Figure 4 **Customer Graph for the Problem of Example 1**

	p_5	p_7	p_3	p_1	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	.	.	.	.

Figure 5 **An Optimal Order for $\{p_1, p_3, p_4, p_5, p_7\}$ of the Problem of Example 1 with p_2 and p_6 Added Immediately After p_4 to Give an Optimal Order for the Original Problem**

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. 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 $c(p_2) \subseteq c(p_4)$ and $c(p_6) \subseteq c(p_4)$, both p_2 and p_6 can be removed. Inserting them after p_4 in any optimal order for the reduced set of products, gives an optimal order for the original problem. This is illustrated in Figure 5.

Our second preprocessing step is more obvious: if P can be partitioned into two sets $P = P_1 \cup P_2$ such that $c(P_1) \cap c(P_2) = \varnothing$, 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 $A^\star$ 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 by Becceneri et al. (2004), although only informal arguments on correctness are given.

RIGHTSLINK

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

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

LEMMA 4. *If $d = \min\{|N(c)| \mid c \in C\}$ 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* of c, denoted c' can be obtained in two ways. One way is to remove an entire customer $c \in C$. In this case we define $c'(p) = c(p) - \{c\}$ for each $p \in P$. The other way is by merging two adjacent customers, i.e., two customers for which there exists a $p \in P$ such that $\{c_1, c_2\} \subseteq c(p)$. In this case we define

$$c'(p) = \begin{cases} (c(p) - \{c_2\}) \cup \{c_1\} & \text{if } c_2 \in c(p), \\ c(p) & \text{otherwise.} \end{cases}$$

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 minimum open stacks for a problem defined by $c(p)$, and m' be the minimum 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 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 implementation of the Becceneri et al. (2004) algorithm and a greedy clique finder that does not do contractions but tries to find maximal cliques starting from the set of customers that order each individual product.

Now consider how to improve the upper bound. For this we experimented with 11 greedy heuristics (whose general form is shown in Figure 6) that, 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 as follows:

```
heuristic(S)
  min := 0
  while (S ≠ ∅)
    heuristically select p
    S := S − {p}
    if (a(p, S − {p}) > min) min := a(p, S − {p})
  return min
```

Figure 6 (Meta) Pseudocode for Greedy Heuristics

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

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

(B) This heuristic selects the product p that, 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 among its products as follows: 2^{-m} for the first scheduled product, 2^{-m+1} for the second, $\ldots$, 2^{-2} for the second to last, and 2^{-1} for the last product. As a result, products that initiate or are close to the activation of the stack are not favored, while those that are near the end or actually close the stack, are favored:

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

(E) This heuristic selects the product p that, 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 will denote (F)) of Becceneri et al. (2004) that 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.

RIGHTSLINK

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

```
stepwise(L, U)
  for try := L to U
    FAIL := ∅
    min := stacks(P, try, try)
    if (min = try) return min
    for (S ∈ FAIL) stack[S] := 0
```

Figure 7 Pseudocode for Stepwise Search for Optimal Solution

6. Search Strategies

As mentioned in Section 1, our $A^\star$ dynamic-programming algorithm is particularly effective when called with $L = U = n$, where it explores only schedules that use exactly n active stacks. This is almost certainly related to the fact that the problem is *fixed parameter tractable* (Linhares and Yanasse 2002), i.e., 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 to *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 to *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 than or equal to *try* is found, it iterates using as a 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)

```
binarychop(L, U)
  gmin := U + 1
  while (L ≤ U)
    FAIL := SUCCESS := ∅
    try := (L + U) div 2
    min := stacks(P, try, try)
    if (min ≤ try)
      gmin := min
      U := min − 1
      for (S ∈ FAIL ∪ SUCCESS) stack[S] := 0
    else
      L := try + 1
      for (S ∈ FAIL) stack[S] := 0
  return gmin
```

Figure 8 Pseudocode for Binary Chop Search for Optimal Solution

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

Figure 9 Pseudocode for Backwards Stepwise Search for Optimal Solution

(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 (nonoptimal) solution. Note that since stacks can actually return a value smaller than *try* we do not just decrease *try* by one each time, but instead we reduce it to one less than the minimum we last found.

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 3.4 Ghz with 2 GB 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 the 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 the Constraint 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

RIGHTSLINK

Table 1 Results for Aggregate Instances from the Constraint Modelling Challenge

File	Mean best	Total time per instance (ms)			Search to find optimal			Total search effort		
		Mean	Median	Max.	Mean	Median	Max.	Mean	Median	Max.
problem_10_10	8.03	0.09	0	2	7.01	7	23	8.21	7	86
problem_10_20	8.92	0.13	0	4	10.45	10	46	15.66	11	419
problem_15_15	12.87	0.37	0	7	13.75	13	89	39.19	14	584
problem_15_30	14.02	2.49	1	43	25.17	22	487	260.09	23	6,031
problem_20_10	15.88	0.52	0	4	12.99	10	92	36.96	28	179
problem_20_20	17.97	5.83	1	86	29.31	19	561	428.12	20	6,634
problem_30_10	23.95	1.63	1	9	17.59	10	178	57.62	52	280
problem_30_15	25.97	7.48	3	46	35.45	15	384	282.63	113	1,634
problem_30_30	28.32	718.36	3	10,074	999.80	30	64,954	31,473.85	30	379,396
problem_40_20	36.38	96.89	8	607	90.69	20	1,329	2,454.55	147	14,757
Shaw_20_20	13.68	12.43	11	42	45.80	19	474	812.76	667	3,020
wbo_10_10	5.92	0.14	0	1	9.82	10	11	14.00	10	60
wbo_10_20	7.35	0.33	0	6	20.27	19	57	47.98	19	629
wbo_10_30	8.20	1.27	0	21	26.32	27	30	147.53	28	1621
wbo_15_15	9.35	1.11	1	7	15.57	15	31	103.60	71	579
wbo_15_30	11.58	28.34	2	213	85.82	30	1,936	2,496.88	30	17,724
wbo_20_10	12.90	0.51	0	3	11.74	10	25	40.26	40	96
wbo_20_20	13.69	8.72	7	42	38.48	20	338	540.33	363	2,894
wbo_30_10	20.05	1.78	2	5	14.52	10	70	60.79	58	117
wbo_30_15	20.96	8.08	7	30	28.54	15	155	280.28	253	859
wbo_30_30	22.56	1,108.10	306	8,686	608.63	30	14,230	41,707.21	16,392	319,162
wbop_10_10	6.75	0.10	0	1	9.82	10	10	14.22	10	42
wbop_10_20	8.07	0.56	0	8	21.02	19	105	69.78	20	715
wbop_10_30	8.55	1.07	1	25	28.98	28	70	113.22	29	2,464
wbop_15_15	10.37	0.77	0	6	15.25	15	32	71.92	15	313
wbop_15_30	12.15	18.63	2	197	162.52	30	2,890	1,593.28	30	18,177
wbop_20_10	14.28	0.49	0	3	11.82	10	28	32.30	25	85
wbop_20_20	14.87	7.99	2	58	40.24	20	659	473.01	20	3,428
wbop_30_10	22.48	1.21	1	5	10.78	10	22	39.48	39	83
wbop_30_15	22.38	7.50	5	38	25.13	15	166	249.48	156	1,070
wbop_30_30	23.84	986.12	87	8,770	1,113.31	30	35,735	31,250.81	2,973	300,677
wbp_10_10	7.28	0.08	0	1	7.60	7	18	11.68	8	70
wbp_10_20	8.71	0.17	0	2	11.93	12	36	25.07	13	330
wbp_10_30	9.31	0.20	0	2	14.03	14	21	25.43	15	268
wbp_15_15	11.05	0.56	0	6	13.82	13	54	59.75	15	509
wbp_15_30	13.09	5.02	1	60	28.75	23	308	539.16	26	6,580
wbp_20_10	15.12	0.52	0	3	11.97	10	45	41.12	40	96
wbp_20_20	15.41	7.26	2	83	50.58	19	1,000	468.31	31	5,136
wbp_30_10	23.18	2.03	2	8	23.68	10	100	73.33	66	185
wbp_30_15	22.98	10.56	7	45	57.48	15	651	377.62	270	1,348
wbp_30_30	24.46	1,203.78	5	17,097	2,071.78	30	132,556	45,069.34	61	765,944

performed using all $A^\star$ improvements, all preprocessing steps, all lower and upper bound 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 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). Because 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 stacks that do not immediately return, because of cache hit, $S = \varnothing$, or the definite choice optimization of Lemma 1. Note that because 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

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

RIGHTSLINK

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

Table 2 Results for the Individual Instances from the Constraint Modelling Challenge

Instance	$\|C\|$	$\|P\|$	Best value	Proved optimal?	Runtime (ms)	Search for best	Total search
Miller_20_40	20	40	13	✓	610	40	39,656
GP1	50	50	45	✓	8.4	42	42
GP2	50	50	40	✓	11.2	48	48
GP3	50	50	40	✓	12.6	50	50
GP4	50	50	30	✓	10.5	37	37
GP5	100	100	95	✓	84.8	208	208
GP6	100	100	75	✓	138.0	100	100
GP7	100	100	75	✓	118.8	99	99
GP8	100	100	60	✓	174.3	96	96
NWRS1	10	20	3	✓	0.2	8	8
NWRS2	10	20	4	✓	0.1	11	11
NWRS3	15	25	7	✓	0.0	13	13
NWRS4	15	25	7	✓	0.1	15	15
NWRS5	20	30	12	✓	1.4	20	20
NWRS6	20	30	12	✓	1.0	23	23
NWRS7	25	60	10	✓	3.0	32	32
NWRS8	25	60	16	✓	2,118.6	40	86,869
SP1	25	25	9	✓	26.4	17	1,269
SP2	50	50	19	×	1,650.0	25,785	—
SP3	75	75	36	×	1 hour	949,523	—
SP4	100	100	56	×	4 hours	3,447,816	—

optimal. Note that sometimes we can require fewer calls to stacks than there are products to prove optimality, 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, 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 best solution found, and the total search effort to prove optimality (or—if optimality was not proved).

Our program found the optimal solutions for all instances except SP2, SP3, and SP4, which hit the search limit (set at $2^{25} = 33{,}554{,}432$ calls to stacks). Again, only a few of the instances are difficult, in particular the SP benchmarks that were specifically designed to have large path width. Our code finds a solution of size 19 for SP2 using 25,785 calls to stacks before hitting the limit trying 18 stacks. The runtime shown for SP2, SP3, and SP4 is the time to find the best solution. The best lower bounds we have discovered for SP2, SP3, and SP4 calculated using lower-bound heuristics and using stepwise are 18, 15, and 22, respectively.

Table 3 Comparative Results over the Entire Benchmark Suite

Search method	Total calls to stacks	Total time (secs.)
$A^\star$	56,231,534	1,386
stepwise	29,887,854	880
binarychop	25,351,370	715
backwards	21,271,366	572
backwards–definite	33,992,526	1,260
backwards–$a'(p, A)$	169,638,021	441
backwards–redundant	30,166,640	850
backwards–red–def	70,759,348	2,231
backwards–partition	21,275,426	573
backwards–upper	21,298,294	570
backwards–lower	21,452,365	575

7.2. The Effect of the Optimizations

In this section 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 solve each instance optimally (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 is reduced 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 is reduced 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 16,305 redundant

RIGHTSLINK

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

Table 4 Comparison of the Heuristics over the 5,803 Benchmarks of the Suite

Heuristic	(A)	(B)	(C)	(D)	(E)	(F)
best	3,046	3,596	3,615	3,840	5,073	5,446
unique	29	1	7	43	159	514
optimal	2,733	3,231	3,248	3,458	4,608	4,986
sum	98,167	96,798	96,769	96,462	94,592	94,093

products out of 101,385 total products, a 16% reduction in size on average. Given that each extra product could in the worst case double the search space, this is vital. This is masked by using the definite choice optimization, if both are removed the program fails to solve NWRS8, which has 20 redundant products out of 60.

There are 113 instances where the products are separable (which surprised us somewhat), with 2.42 separate parts on average. In most cases the result of separating is not much better than not, as the separable partitions are usually tiny singletons. But there are examples such as Warwick_1711 where the search space is reduced from 1,853 calls to stacks to 183, even though the separable parts are size 1, 1, and 5 out of 29 nonredundant products.

The effect of the upper-bound heuristics is not too great once we use backwards. They improve the number of sets in 884 cases, but the percentage improvement is negligible overall (0.0012%) because 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 5,964 partitions of products for 5,803 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 5,803), and the total *sum* of the heuristic results is shown. Clearly the minimal-cost node heuristic of Becceneri 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 2,718 benchmarks of 5,803, while the HAC approach gives the optimal on 3,380.

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

7.3. Comparing with Other Results in the Competition

A full report of the competition is available from the competition website, Constraint 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). Theirs 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 different possible early and late sets and solving these sub-problems independently. Finally, they also use a local-search technique to explore around every new better solution in order to find good solutions early. Overall, our run times were around two 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 $stacks(S)$ given at the beginning of Section 3, using a bottom-up dynamic-programming approach that requires all $2^{|P|}$ subproblems to be solved. As we can see from the median calls to stacks required, this is very wasteful for the MOSP. This entry solved all instances with fewer 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 that was unable to solve most instances; a number of local-search methods that 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, and Miller_20_40 (and of course SP2, SP3, and SP4). Overall, our results are usually two 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.

RIGHTSLINK

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

8. Conclusion

The call-based $A^\star$ dynamic-programming formulation of minimizing open stacks gives a very effective algorithm for solving these problems. It can be improved by searching backwards for a minimum 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 $a'(p, A)$ to get a better lower bound on the 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

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

References

Becceneri, J. C., H. H. Yannasse, N. Y. Soma. 2004. A method for solving the minimization of the maximum number of open stacks problem within a cutting process. *Comput. Oper. Res.* **31** 2315–2332.

Constraint Modelling Challenge. 2005. Constraint modelling challenge 2005, http://www.dcs.st-and.ac.uk/~ipg/challenge/.

Faggioli, E., C. A. Bentivoglio. 1998. Heuristic and exact methods for a cutting sequencing problem. *Eur. J. Oper. Res.* **110** 564–575.

Linhares, A., H. H. Yanasse. 2002. Connections between cutting-pattern sequencing, VLSI design, and flexible machines. *Comput. Oper. Res.* **29** 1759–1772.

Yannasse, H. H. 1997. On a pattern sequencing problem to minimize the maximum number of open stacks. *Eur. J. Oper. Res.* **100** 454–463.

Yuen, B. J. 1991. Heuristics for sequencing cutting patterns. *Eur. J. Oper. Res.* **55** 183–190.

Yuen, B. J. 1995. Improved heuristics for sequencing cutting patterns. *Eur. J. Oper. Res.* **87** 57–64.

Yuen, B. J., K. V. Richardson. 1995. Establishing the optimality of sequencing heuristics for cutting stock problems. *Eur. J. Oper. Res.* **84** 590–598.

Downloaded from informs.org by [144.214.88.242] on 18 September 2017, at 20:08 . For personal use only, all rights reserved.

RIGHTSLINK