OPERATIONS RESEARCH, Vol. 25, No. 1, January–February 1977

An Algorithm for Two-Dimensional Cutting Problems

NICOS CHRISTOFIDES and CHARLES WHITLOCK

Imperial College, London, England

(Received original January 3, 1974; final, April 7, 1976)

We present a tree-search algorithm for two-dimensional cutting problems in which there is a constraint on the maximum number of each type of piece that is to be produced. The algorithm limits the size of the tree search by deriving and imposing necessary conditions for the cutting pattern to be optimal. A dynamic programming procedure for the solution of the unconstrained problem and a node evaluation method based on a transportation routine are used to produce upper bounds during the search. The computational performance of the algorithm is illustrated by tests performed on a large number of randomly generated problems with constraints of varying "tightness." The results indicate that the algorithm is an effective procedure for solving cutting problems of medium size.

THE PROBLEM of cutting a one-dimensional object (e.g., a length of some material) into smaller pieces—each piece having a given length and value—in such a way so as to maximize the total value of pieces cut is the well-known "knapsack problem." This problem has been examined by a number of authors [8, 12, 13], and methods for its solution have been proposed using either dynamic programming [8] or tree-search techniques [9, 12]. This attention is motivated by many practical problems that can be formulated as knapsack problems, typical cases being the steel bar cutting stock problem [6], the vehicle loading problem [3], and the division of work into time shifts.

The two-dimensional cutting problem requires cutting a plane rectangle into smaller rectangular pieces of given sizes and values to maximize the sum of the values of the pieces cut. This version of the problem appears in the problem of cutting steel or glass plates [10] into required stock sizes to minimize waste. By taking the value of a piece to be proportional to its area, we can formulate the waste minimization problem as one of maximizing the value of the pieces cut. The problem also appears in cutting wood plates to make furniture and paper board to make boxes.

A special case of the general two-dimensional cutting problem is one in which all cuts must go from one edge of the rectangle to be cut to the opposite edge, i.e., the cut has to be of a "guillotine" type. Some cutting pat-

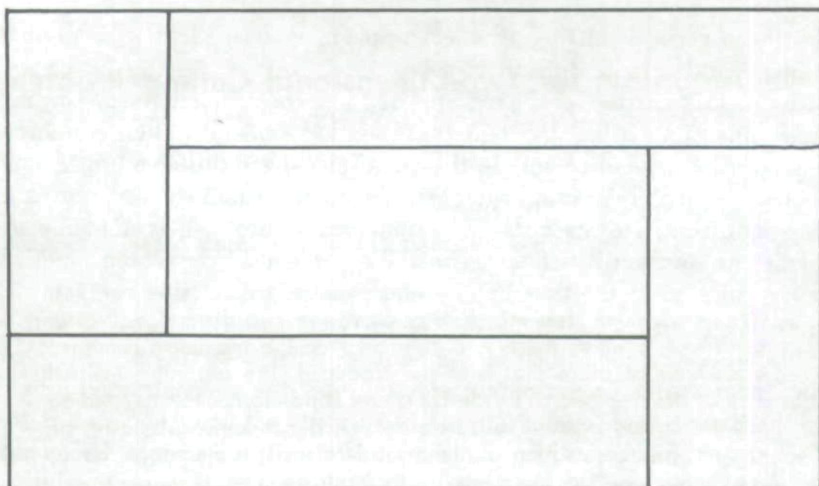

Figure 1(a). Cutting pattern infeasible with guillotine cuts.

terns—e.g., that shown in Figure 1(a)—could not be produced by this type of cut. However, the restriction of "guillotine" cuts appears very often in practice especially for the cutting of paper and glass. Figure 1(b) shows a possible cutting pattern using guillotine cuts where the cuts are numbered in the order in which they could be made, although other sequences are obviously also possible. In this paper only "guillotine" type cuts are considered, and when the word "cut" is used this type of cut is implied.

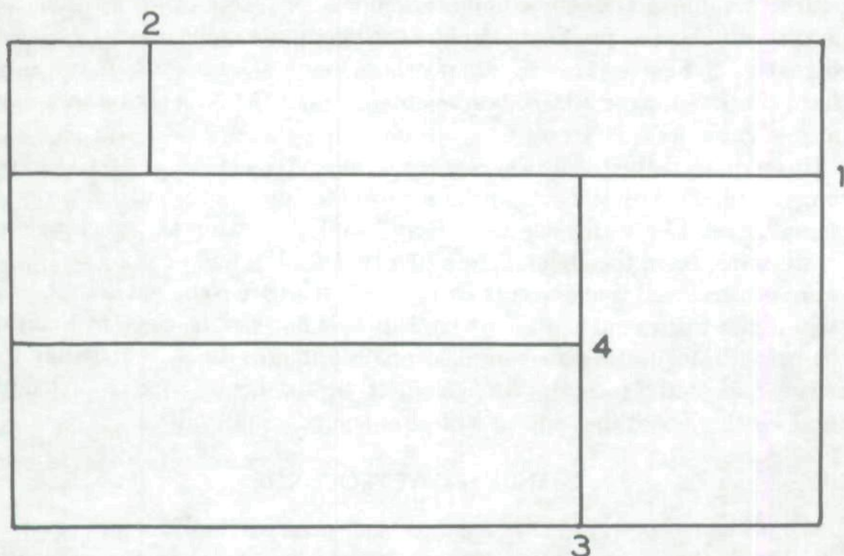

Figure 1(b). Cutting pattern made with guillotine cuts.

The more general cutting problem appears less frequently in practical problems and is more difficult to solve.

In practice cutting problems appear in a constrained form, the most usual constraint being the one that restricts the maximum number of pieces of each type to be cut. In the one-dimensional problem upper bound constraints on the variables can easily be included in a dynamic programming algorithm. However, the two-dimensional problem is not amenable to efficient solution by these means. We present a tree-search algorithm for the solution of the two-dimensional constrained cutting problem. The algorithm described limits the size of the tree search by deriving and imposing necessary conditions for the cutting pattern to be optimal and by using the dynamic programming procedure for the unconstrained problem together with a node evaluation method based on a transportation routine as subalgorithms to produce upper bounds during the search. The computational performance of the algorithm is illustrated by tests performed on a large number of randomly generated problems with constraints of varying "tightness." The results indicate that the algorithm is an effective procedure for solving cutting problems of medium size.

1. PROBLEM

The constrained two-dimensional cutting problem can be defined as follows. Let a large rectangle $A_0 = (L_0, W_0)$ (i.e., of length L_0 and width W_0 units) be given, together with a set R of m smaller rectangular pieces $R = \{(l_1, w_1), (l_2, w_2), \cdots, (l_m, w_m)\}$, each piece in R having associated with it a value v_i and a maximum number b_i of pieces of type i that can be cut from A_0. The problem is to find the maximum value of $z = \sum_{i=1}^{m} \xi_i v_i$ so that $\xi_i \leqq b_i$, $i = 1, \cdots, m$, and there exists a series of cuts on A_0 so that ξ_i pieces of type i in R can be cut from A_0, the ξ_i being nonnegative integer variables.

In order to distinguish between the given pieces in set R and the rectangles produced by the cuts on A_0 at any stage during the cutting process, we will refer to the former as "pieces" and the latter as "rectangles." It will be assumed throughout the paper that L_0, W_0, and l_i, W_i, $i = 1, \cdots, m$ are integers and that the cuts on the rectangles are to be made in integer steps along the x or y axes. This limitation is not serious since in practice the actual dimensions can be scaled up. It should also be noted that the orientation of the pieces is considered to be fixed, i.e., a piece of length l and width w is not the same as a piece of length w and width l.

2. ENUMERATIVE PROCEDURE

We will first describe a procedure that generates all possible cutting patterns without duplication and then describe methods for implicitly enumerating these patterns in a general tree-search algorithm.

All possible cutting patterns can be generated by developing a tree, where branchings represent cuts on a rectangle. Thus, the branches emanating from the root-node of the tree correspond to all possible cuts on A_0, and each node at the end of a branch represents the rectangles produced by the corresponding cut on A_0.

To generate all possible cutting patterns (including the original uncut rectangle), we must include among "all possible cuts" an artificial "cut" that leaves the rectangle intact. This "cut" plays an important role in the algorithm and will be referred to as a *0-cut.* Thus, a rectangle that has been "cut" by a 0-cut must not be a candidate for future cutting and must—from that node onward—be considered fixed.

At successive nodes a rectangle is selected from the list of rectangles represented by a node, and branching occurs from that node by making all possible cuts on the chosen rectangle.

Effects of Symmetry

Given a rectangle with dimensions (p, q), there are $p + q - 1$ cuts that can be made on it—along $x = 1, \cdots, (p - 1)$, along $y = 1, \cdots, (q - 1)$, and the 0-cut. If, however, all these cuts are made, producing $p + q - 1$ branches, then the sets of rectangles represented at successor nodes will be duplicated at several nodes because of the appearance of symmetrical cutting patterns. These duplications can easily be removed as follows.

Given a rectangle (p, q), a cut at $x = a$ on (p, q) produces two rectangles (a, q) and $(p - a, q)$. Clearly, these same two rectangles could have been produced by a cut at $x = p - a$, which is symmetrically opposite to the cut at $x = a$ with respect to (p, q). This can be avoided without missing any unique cutting pattern by making cuts only up to half way along the x-side of the rectangle. Thus, the range of x-cut is limited instead of $1 \leqq x \leqq (p - 1)$ to $1 \leqq x \leqq [p/2]$ where $[p/2]$ means "the greatest integer not greater than $p/2$." Similarly, the range of the y-cuts can be redefined to be $1 \leqq y \leqq [q/2]$.

Effects of Cut Ordering

Consider a rectangle (p, q) and suppose that at some node (p, q) is cut at $x = a$, producing two rectangles (a, q) and $(p - a, q)$. Then suppose at some successor node $(p - a, q)$ is cut at b, $a < b \leqq [(p - a)/2]$ producing three rectangles (a, q), (b, q), $(p - a - b, q)$ from (p, q). These same three rectangles could have been produced by first making the cut $x = b$ on (p, q) and then making the cut $x = a$ on $(p - b, q)$. This type of duplication can obviously be removed without missing any unique cutting patterns by introducing an arbitrary cut ordering so that if a rectangle is cut at, say, $x = \alpha$, then all subsequent x-cuts on the two resultant rectangles must be greater than or equal to α. This restriction,

together with the restriction imposed by symmetry in the last section, implies that for the larger of the two resultant rectangles, $(p - \alpha, q)$, the range of x-cuts is now limited to $\alpha \leqq x \leqq [(p - \alpha)/2]$ and, in particular, if $[(p - \alpha)/2] < \alpha$ no further x-cut on that rectangle need be made. For the smaller of the two resultant rectangles, (α, q), the restriction imposed by the cut ordering implies that no further x-cut is possible.

The consequence of cut ordering as explained above is to eliminate from explicit consideration different sequences of cuts when these lead to the same final cutting pattern. A similar kind of restriction can be imposed on the y-cuts.

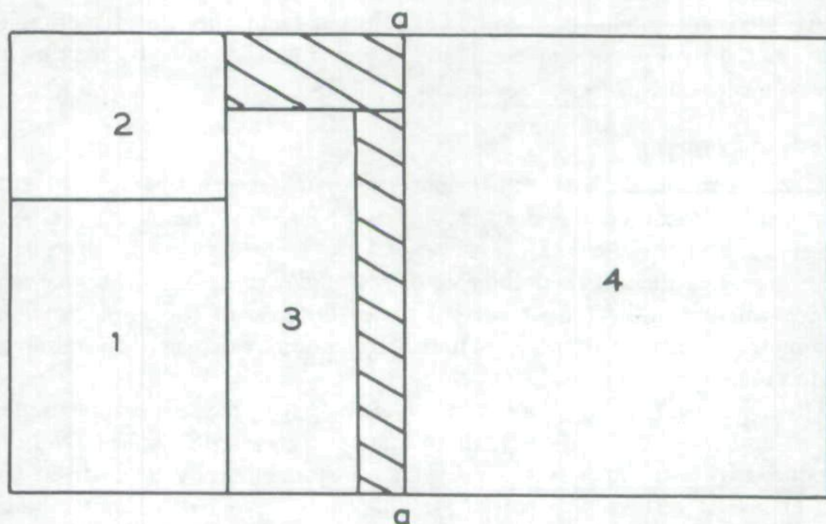

Figure 2(a). Nonnormalized cutting pattern.

Normal Cuts

If a rectangle (p, q) is to be cut by, say, an x-cut at some position x, then in the final cutting pattern there must be some combination of the lengths l_i of the available pieces, for which $w_i \leqq q$, whose sum must be exactly x. If this were not so (*i.e.*, as shown in Figure 2(a) for the x-cut marked $a - a$), then an alternative cut at position $x' < x$ could also lead to a final cutting pattern (as shown in Figure 2(b)), involving the same pieces and hence having the same value as the pattern in Figure 2(a). The pattern of Figure 2(b) will be called *normal*, and it is apparent that for any pattern there is a normal equivalent. It should, however, be pointed out that, contrary to the cut restrictions imposed previously, normality is a property of a cutting pattern that is relative to the set of pieces available for cutting. It should also be noted that a direct consequence of normality is that "waste"—such as the shaded area in Figure 2(b)—is never

cut away by the algorithm but is left attached to a rectangle from which some piece will later be produced.

Thus, one can (without loss of optimality) limit the x-cuts only to those values of x leading to normal patterns; and similarly for the y-cuts. The feasible x-cuts on a rectangle (p, q), given the set R of pieces (l_i, w_i) to be cut, must then be at values of x that are the elements of some set S^q.

The sets S^q can be calculated for all values of q by a single iterative equation in one pass. First assume that the m pieces are ordered in non-decreasing values of w_i. We define a function $f_r(x)$ as follows.

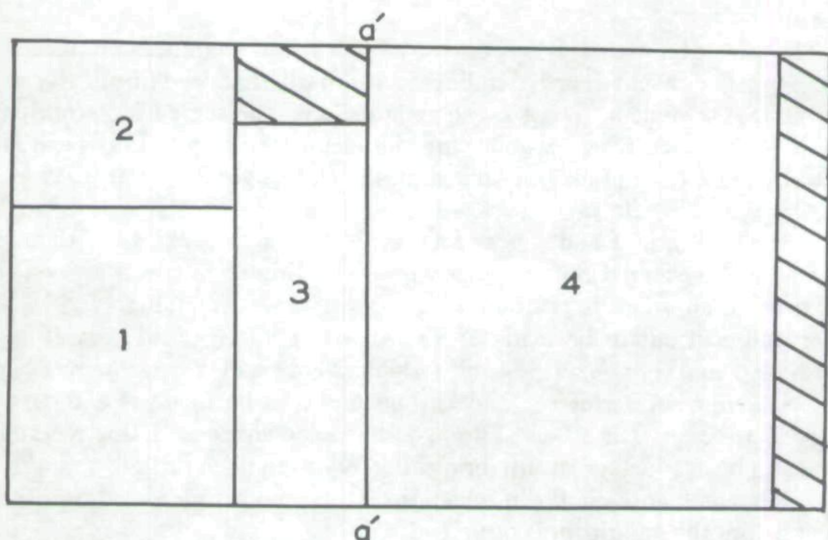

Figure 2(b). Normalized form of cutting pattern in Figure 2(a).

If there exists one or more integer-valued vectors $\bar{\xi}$ that satisfy

$$x = \sum_{i=1}^{r} \bar{\xi}_i l_i, \qquad 0 \leqq \bar{\xi}_i \leqq b_i, \tag{1}$$

then set $f_r(x) = w_{i*}$, where

$$i^* = \min_{\bar{\xi}} [\max \{i \mid \bar{\xi}_i \neq 0\}]. \tag{2}$$

If no such vector $\bar{\xi}$ satisfying (1) exists, then set $f_r(x) = \infty$.

It follows from the above definition of $f_r(x)$ that if $f_m(x) \leqq q$, then x is the sum of lengths l_i of some combination of pieces (satisfying the b_i constraints) all of whose widths w_i are less than q, and x must therefore be in the set S^q. Thus, once the tableau $f_m(x)$ is generated, the sets S^q can be produced for any q.

The tableaux $f_r(x)$, $1 \leqq r \leqq m$, $0 \leqq x \leqq L_0$ can be generated simply by the recursion: $f_i(x) = \min \{f_{i-1}(x), \max \{w_i, \min_k (f_{i-1}(x - kl_i))\}\}$,

where $1 \leqq k \leqq \min\{b_i, [x/l_i]\}$, k integer, $x \geqq l_i$; $f_i(x) = f_{i-1}(x)$, $x < l_i$; and $f_0(x) = \infty$ for all x.

The corresponding sets T^p for the y-cuts can be generated in a similar manner.

Description of Enumerative Algorithm

We now describe an algorithm that generates all normal cutting patterns on a rectangle $A_0 = (L_0, W_0)$, given a set R of pieces to be cut. In this algorithm each node represents a state of the rectangle after cutting has taken place, and a tree branching from one node to another represents a cut.

The state at a node n is described by the list L of rectangles produced by the sequence of cuts corresponding to the path that leads from the root of the tree to node n. In list L each rectangle is represented by a four-part label (p, q, x, y), where p and q are the rectangle's length and width and x and y are integers that can take values in the ranges $0 \leqq x \leqq [p/2] + 1$, $1 \leqq y \leqq [q/2] + 1$.

The meaning of x and y is as follows. If $1 \leqq x \leqq [p/2]$, then the next cut to be considered on rectangle (p, q)—if this rectangle is chosen for cutting—is an x-cut at position x. If $x = [p/2] + 1$ and $1 \leqq y \leqq [q/2]$, then the next cut to be made on (p, q) is a y-cut at position y. If $x = [p/2] + 1$ and $y = [q/2] + 1$, all feasible x-cuts and y-cuts on rectangle (p, q) have been performed and the next cut to be made is a 0-cut. If $x = 0$, we infer that a 0-cut on (p, q) has been made and this rectangle is not to be cut further by any branching following node n.

A few comments on the mechanics of the search procedure are necessary before the algorithm is described.

(i) Only one rectangle is cut at a node, and the procedure is exhaustive because of the introduction of the 0-cut, as explained earlier.

(ii) When a rectangle (p, q) is produced by an x-cut, future x-cuts on (p, q) must be at values of $x \geqq X$, where X is calculated as described earlier. Similarly, Y is calculated if rectangle (p, q) was producd by a y-cut.

(iii) The state of the search is represented by:

 (a) The list L of four-part labels corresponding to rectangles produced by the cuts so far. This list is updated for forward and backward branchings.

 (b) Say that at node j a rectangle (p_j, q_j) from list L was chosen for cutting and the four-part label for that rectangle was (p_j, q_j, x_j, y_j). Then a vector $Q(j)$ of six-part labels $(p_j, q_j, x_j, y_j, X_j, Y_j)$ is stored for all j, $0 < j < n$, where X_j and Y_j are as explained in (ii) above.

A diagrammatic description of the algorithm is shown in Figure 3, and a detailed description is given in the appendix.

3. TREE-SEARCH ALGORITHM

Section 2 gave an enumerative procedure that could generate all normal cutting patterns with respect to a set $R = \{(l_i, w_i), i = 1, \cdots, m\}$ and maximum numbers b_i, $i = 1, \cdots, m$ of pieces to be cut. The procedure generates all cutting patterns without symmetric duplications and without explicitly considering different sequences of cuts when these lead to the same cutting pattern. However, up to now no consideration has been given to the fact that the pieces in R have values v_i associated with them, and the purpose of this section is to examine ways of limiting the tree search described earlier in order to solve the optimization problem of Section 1.

Upper Bound on Value of Cutting Pattern at Node

In the normalized optimal cutting pattern there is exactly one piece from the set R fitted into each rectangle of this pattern because (as mentioned earlier) the "waste"—i.e., rectangles with nothing fitted in them—is not cut away by the algorithm itself. Thus, at some node of the tree when the list of rectangles already cut is given by the set L, let those rectangles that have had a 0-cut made on them form the subset $H_0 \subseteq L$. The rectangles in H_0 will not be cut at any node below the current node and in the final cutting pattern will have some piece fitted in them. Hence, an upper bound on the value obtainable from the rectangles in H_0 can be obtained at any node of the tree by allocating pieces to these rectangles from the set R in an optimal fashion. This allocation can be done quite simply as follows:

Form a matrix $[a_{ik}]$ with m rows corresponding to the pieces in R and u columns corresponding to the u (say) rectangles (p_k, q_k) in H_0. Set $a_{ik} = v_i$, if $l_i \leqq p_k$ and $w_i \leqq q_k$ and $a_{ik} = -\infty$ otherwise.

The best feasible allocation of pieces to rectangles is then given by a solution to the transportation problem:

$$\max z = \textstyle\sum_{k=1}^{u} \sum_{i=1}^{m} a_{ik} x_{ik} \quad (3)$$

$$\textstyle\sum_{i=1}^{m} x_{ik} \leqq 1 \quad (4)$$

$$\textstyle\sum_{k=1}^{u} x_{ik} \leqq b_i \quad (5)$$

$$x_{ik} \geqq 0.$$

The solution to this problem is made very simple by the special structure of the $[a_{ik}]$ matrix, and the problem can be solved by a particularly efficient version of the transportation routines [2].

The rectangles in L that have not had 0-cuts made on them are liable, at future branchings, to be cut further into smaller rectangles and hence

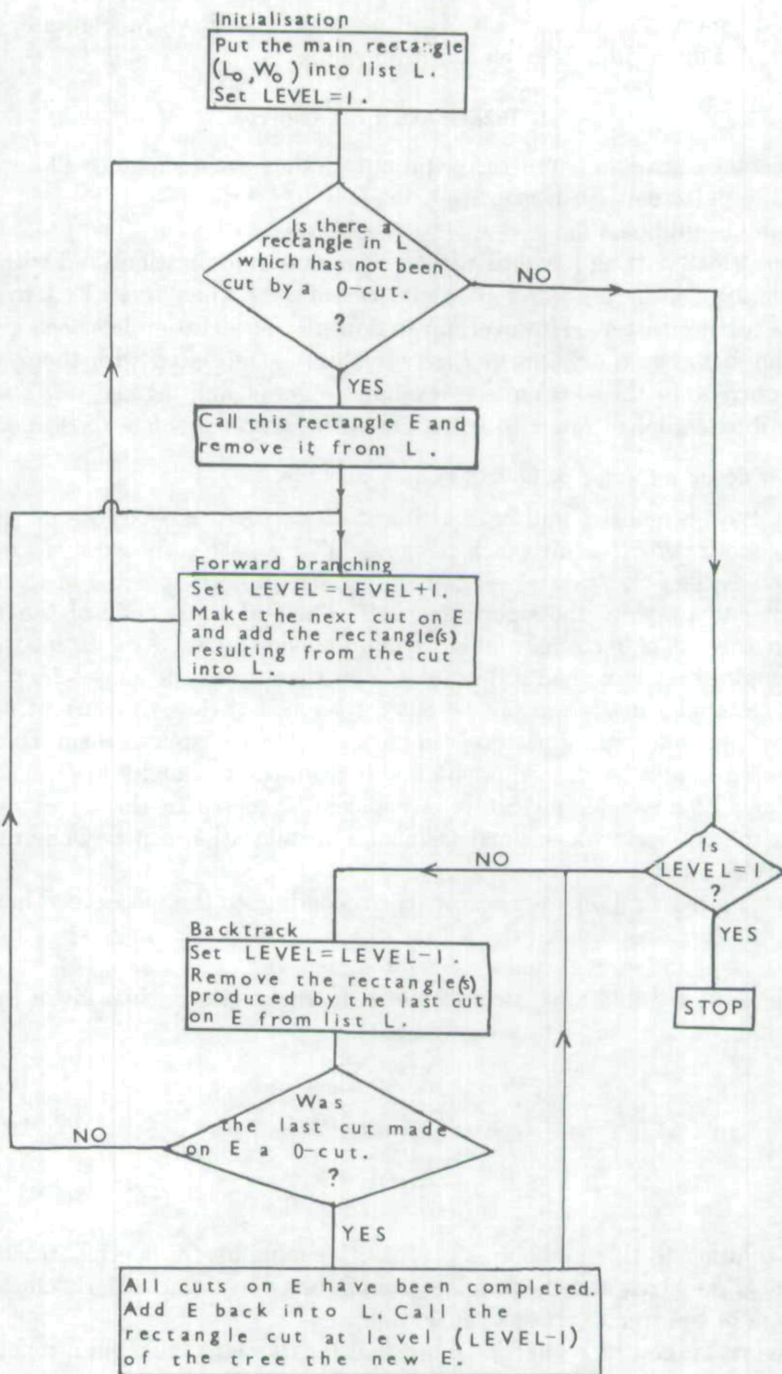

Figure 3 Flow chart of the algorithm.

may be allocated several pieces from R in the final solution. It is therefore not possible to use the transportation routine to calculate an upper bound for these rectangles, as this only allocates one piece per rectangle. However, an upper bound on the value obtainable from each rectangle in $L - H_0$ can be derived by solving, for every such rectangle, the unconstrained two-dimensional cutting problem using the first of the dynamic programming procedures given by Gilmore and Gomory [8]. The second method given in [8] is incorrect, as shown in [11].

The procedure for calculating an upper bound at any node is therefore to solve the transportation problem for all rectangles in H_0 and to solve the unconstrained two-dimensional cutting problem for all other rectangles.

If at any node the value z^* of the upper bound is greater than the value $\bar{z}$ of the best solution so far, and if the number of each piece i used to generate this value is not greater than the constraint b_i, then an improved solution has been obtained. z^* can then replace $\bar{z}$, and backtracking can occur. If on the other hand $z^* \leqq \bar{z}$, then further branching from the current node can be discountinued and backtracking can again take place.

If neither of these cases occurs, then forward branching can take place until a terminal node is reached (i.e., a node at which all rectangles in the list L have had 0-cuts made on them), in which case the value of the solution of the transportation problem is the value of the cutting pattern corresponding to that node.

We note that the solution to the unconstrained problems need not be calculated at each node but that a single dynamic programming table corresponding to the solution for the initial rectangle (L_0, W_0) and calculated once at the beginning of the tree search could be used to obtain directly the unconstrained solutions to any rectangle (p, q), $1 \leqq p \leqq L_0$, $1 \leqq q \leqq W_0$ in the list L. The major part of the computation of the upper bound is therefore the solution of the transportation problem of (3), (4), and (5), and this solution need take place only at nodes resulting from some 0-cut, i.e., only when the set H_0 of rectangles changes.

Branching Strategies

The choice of which rectangle from the list L of available rectangles is to be cut has been left unspecified. There are many ways in which this rectangle can be chosen, and a number of these have been tried.

(i) One simple method is to select that rectangle n with the minimum x-dimension p_n and if more than one such rectangle exists, then choose among these the one with the smallest y-dimension q_n. This corresponds to choosing the "smallest" rectangle; or similarly, the "largest" rectangle could be chosen.

(ii) An alternative method is to select that rectangle for which the unconstrained problem gives the highest value. This is another simple and computationally inexpensive method since the value

of the unconstrained solution is used in the calculation of the bounds.

(iii) A slightly more complex branching strategy that aims to obtain a feasible solution early on in the search is as follows. At each node of the tree the number r_i' of pieces of type i that have been allocated to rectangles by the transportation subroutine is available. Also available is the number r_i'' of pieces of type i used by all the unconstrained dynamic programming solutions to the rectangles still available for cutting (i.e., the ones that have not as yet had a 0-cut made on them). If $r_i = r_i' + r_i''$, then obviously $r_i > b_i$ for at least one $i = 1, \cdots, m$; otherwise, a feasible solution would have been obtained and backtracking would have occurred. Let i^* be that piece i for which $(r_i - b_i)$ is maximum. It is then reasonable to try to reduce the number of pieces used for any type i for which $r_i > b_i$ and in particular that of type i^* since this would produce the largest step toward feasibility. With this in mind, one could then choose to cut the rectangle that uses the largest number of pieces of type i^* in the unconstrained solution. One hopes that, after this rectangle is cut, fewer pieces of type i^* will be used.

The computational results given in the next section have been obtained from a computer program using branching strategy (iii).

4. COMPUTATIONAL RESULTS

The algorithm described in the last two sections was tested on a large number of randomly generated two-dimensional cutting problems and, as can be seen from Table I, the method can be used to solve practical problems of quite reasonable size. This algorithm has solved actual wood cutting problems in the manufacture of furniture.

The random problems were produced as follows. Given the initial rectangle A_0 with area $\alpha_0 = L_0 \cdot W_0$, then m random numbers α_i, $i = 1, \cdots, m$, corresponding to the areas of the m rectangles in R, were generated by sampling from the uniform distribution in the range 0 to 0.25 α_0. The x dimensions l_i of the rectangles in R were generated by sampling from the uniform distribution in the range 0 to α_i and rounded upward to the nearest integer, and the y dimensions w_i were then calculated using the formula $w_i = \alpha_i/l_i$, again rounding the number upward.

The values v_i of the rectangles in R were generated using the function: $v_i = r_i\alpha_i$, where r_i is a uniformly distributed random number in the range 1 to 3. Once more the values v_i were rounded upward.

The constraints b_i on each piece in R were deliberately chosen so that the unconstrained solution obtained from the dynamic program was infeasible. Table I gives details of the size of the problems tested and the time needed for their solution. The computer code was written in FORTRAN IV for

TABLE I
PERFORMANCE OF ALGORITHM

m	(L_0, W_0)	No. of problems solved	Times (CDC 7600 sec)			No. of nodes in tree		
			Max.	Av.	Min.	Max.	Av.	Min
5	(40, 70)	4	1.1	0.96	0.79	1141	835	654
5	(53, 65)	3	0.79	0.73	0.76	194	131	85
5	(50, 100)	6	2.32	1.60	1.31	1943	1024	170
6	(15, 10)	4	14.31	7.15	1.25	11051	5310	1085
7	(40, 70)	4	15.23	7.48	1.18	18013	8866	1109
10	(40, 70)	6	25.43	15.29	0.7	18602	12175	230
20	(40, 70)	5	181.0	130.18	66.14	57284	38807	22184

the CDC 7600 computer and was run using the FTN compiler of that machine. The total memory requirements of the code used for the solution of all problems in Table I was 30 K words.

Table II gives the exact details of three of the problems solved, includ-

TABLE II
DETAILS OF THREE TEST PROBLEMS

Problem No. 1. Initial rectangle $A_0 = (15, 10)$, $m = 7$

Details of pieces:

Dimension vector: $[l_i, w_i] = [(8,4), (3,7), (8,2), (3,4), (3,3), (3,2), (2,1)]$
Value vector: $[v_i] = [66, 35, 24, 17, 11, 8, 2]$
Constraint vector: $[b_i] = [2, 1, 3, 5, 2, 2, 1]$
Solution value: constrained by $[b_i]$, 244 (unconstrained, 249)
Solution time: 2.47 sec. *Number of nodes in tree:* 3,794

Problem No. 2. Initial rectangle $A_0 = (40, 70)$, $m = 10$

Details of pieces:

Dimension vector: $[l_i, w_i] = [(21, 22), (31, 13), (9, 35), (9, 24), (30, 7), (11, 13), (10, 14), (14, 8), (12, 8), (13, 7)]$
Value vector: $[v_i] = [582, 403, 315, 216, 210, 143, 140, 110, 94, 90]$
Constraint vector: $[b_i] = [1, 1, 3, 3, 2, 3, 1, 3, 3, 3]$
Solution value: constrained by $[b_i]$, 2892 (unconstrained, 3006)
Solution time: 24.07 sec. *Number of nodes in tree:* 18,602

Problem No. 3. Initial rectangle $A_0 = (40, 70)$, $m = 20$

Details of pieces:

Dimension vector: $[l_i, w_i] = [(31, 43), (30, 41), (29, 39), (28, 38), (27, 37), (26, 36), (25, 35), (24, 34), (33, 23), (22, 32), (31, 21), (29, 18), (17, 27), (15, 24), (16, 25), (15, 24), (23, 14), (21, 12), (19, 11), (9, 17)]$
Value vector: $[v_i] = [500, 480, 460, 440, 420, 410, 400, 380, 360, 340, 320, 300, 280, 240, 260, 240, 220, 180, 160, 140]$
Constraint vector: $[b_i] = [4, 2, 4, 4, 3, 4, 3, 4, 4, 3, 3, 3, 2, 2, 4, 1, 4, 3, 4, 1]$
Solution value: constrained by $[b_i]$, 1860 (unconstrained, 2240)
Solution time: 66.14 sec. *Number of nodes in tree:* 22,184

ing the sizes, values, and constraints of the pieces in R, and the values of the constrained and unconstrained solutions. Figure 4 shows the cutting pattern that gives the constrained optimal solution to problem 2 in Table II.

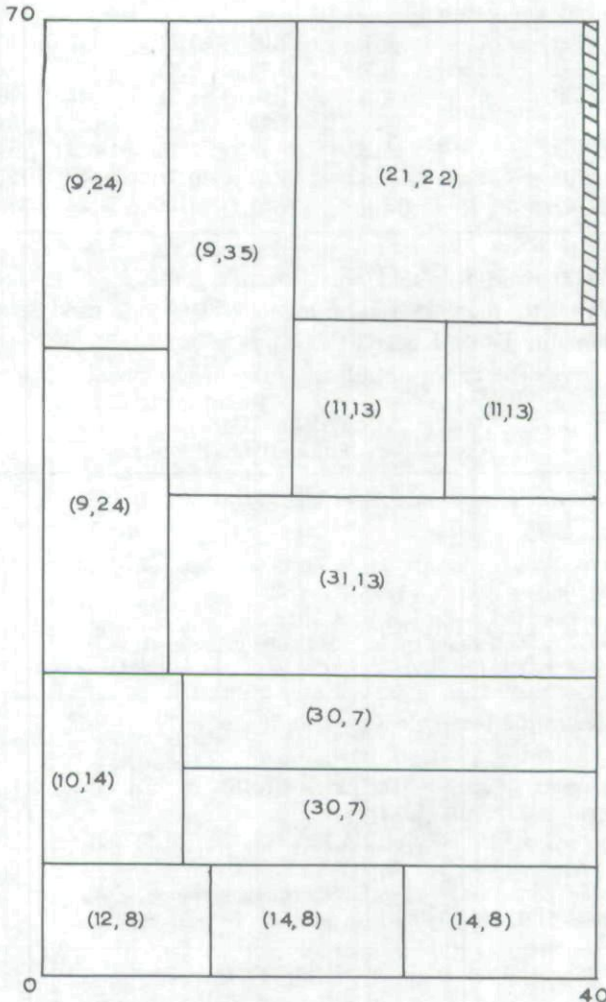

Figure 4. Optimal cutting pattern for Problem 2.

APPENDIX

Let c_i be the number of pieces of type i in R used in the upper bound calculation described in Section 3. Thus, c_i is the sum of the number of pieces used in the solution of the transportation problem for H_0 and the number of pieces used in the dynamic programming solution for the re-

maining rectangles. The description of the complete algorithm, including the calculation of the bounds, is then as follows:

Initialization:

1.1 Set: $n = 1$, $L = \{(L_0, W_0, 1, 1)\}$, $\bar{z} = 0$.

Calculation of the bound:

2.1 Calculate the value of z^* and c_i, $i = 1, \cdots, m$ for the list L.

2.2 If $z^* \leqq \bar{z}$, set $K = 0$ and go to 6.1; otherwise, continue.

2.3 If $b_i \geqq c_i$, for all $i = 1, \cdots, m$, set $\bar{z} = z^*$, $K = 0$ and go to 6.1; otherwise, continue.

Choose a rectangle for cutting:

2.4 Choose a rectangle (p_n, q_n, x_n, y_n) from the list L with $x_n \neq 0$. If none exists set $K = 0$ and go to 6.1; otherwise, set $K = 1$ and go to step 2.5.

2.5 Set $X_n = x_n$ and $Y_n = y_n$.

2.6 Set $Q(n) = (p_n, q_n, x_n, y_n, X_n, Y_n)$ and $L = L - \{(p_n, q_n, x_n, y_n)\}$.

Forward branching (x-cut):

3.1 Set $x = x_n$.

3.2 If $x \geqq [p_n/2] + 1$, set $x_n = x + 1$ and go to 4.1.

3.3 If $x \in S^{q_n}$, go to 3.4; otherwise, set: $x = x + 1$ and go to 3.2.

3.4 Set $L = L \cup \{(x, q_n, x, 1), (p_n - x, q_n, x, 1)\}$.

3.5 Set $x_n = x + 1$, $n = n + 1$, and go to 2.1.

Forward branching (y-cut):

4.1 Set $y = y_n$.

4.2 If $y \geqq [q_n/2] + 1$, go to 5.1.

4.3 If $y \in T^{p_n}$ go to 4.4; otherwise, set $y = y + 1$ and go to 4.2.

4.4 Set $L = L \cup \{(p_n, y, 1, y), (p_n, q_n - y, 1, y)\}$.

4.5 Set $y_n = y + 1$, $n = n + 1$ and go to 2.1.

Forward branching (0-cut):

5.1 Set $L = L \cup \{(p_n, q_n, 0, y_n)\}$.

5.2 Set $x_n = 0$, $n = n + 1$ and go to 2.1.

Backtracking:

6.1 If $n = 1$, stop; all normal cutting patterns have been generated.

6.2 If $K = 0$, go to 6.4.

6.3 $L = L \cup \{(p_n, q_n, X_n, Y_n)\}$.

6.4 Set $n = n - 1$.

6.5 If $x_n > [p_n/2] + 1$, go to 6.5.2. If $x_n = 0$, go to 6.5.3; otherwise, continue.

(*Last cut made was an x-cut*):

6.5.1 Set $L = L - \{(x_n - 1, q_n, x_n - 1, 1).(p_n - x_n + 1, q_n, x_n - 1, 1)\}$ and go to 3.1.

(*Last cut made was a y-cut*):

6.5.2 Set $L = L - \{(p_n, y_n - 1, 1, y_n - 1), (p_n, q_n - y_n + 1, 1, y_n - 1)\}$ and go to 4.1.

(Last cut made was a 0-cut):

6.5.3 Set $L = L - \{(p_n, q_n, 0, y_n)\}$, $K = 1$ and go to 6.1.

ACKNOWLEDGMENT

The authors wish to thank the two referees for their useful suggestions.

REFERENCES

1. R. Art, "An Approach to the Two-Dimensional, Irregular Cutting Stock Problem," I.B.M. Cambridge Scientific Centre Report, No. 320-2006, 1966.
2. J. F. Desler, and S. L. Hakimi, "A Graph-Theoretic Approach to a Class of Integer-Programming Problems," *Opns. Res.* **17,** 1017–1033 (1969).
3. S. Eilon and N. Christofides, "The Loading Problem," *Management Sci.* **17,** 259–268 (1971).
4. L. F. Escudero and E. Garbayo, "The Cutting Stock Problem: Application of Combinational Techniques and Mixed Integer Programming," presented at 8th Mathematical Programming Symposium, Stanford University, August 1973.
5. P. M. Ghare and L. E. Walters, "A Branch-and-Bound Algorithm for the Multi-Dimensional Knapsack Problem," presented at a joint meeting of the 33rd national meeting of the Operations Research Society of America, and American meeting of the Institute of Management Science, 1968.
6. P. C. Gilmore and R. E. Gomory, "A Linear Programming Approach to the Cutting-Stock Problem," *Opns. Res.* **9,** 849–859 (1961).
7. P. C. Gilmore and R. E. Gomory, "Multistage Cutting Problems of Two and More Dimensions," *Opns. Res.* **13,** 94–120 (1965).
8 P. C. Gilmore and R. E. Gomory, "The Theory and Computation of Knapsack Functions," *Opns. Res.* **15,** 1045–1075 (1967).
9. H. Greenberg and R. L. Hegerich, "A Branch Search Algorithm for the Knapsack Problem," *Management Sci.* **16,** 327–332 (1970).
10. S. Hahn, "On the Optimal Cutting of Defective Glass Sheets," I.B.M. New York Scientific Center Report No. 320–2916, 1967.
11. J. C. Herz, "A Recursive Computing Procedure for Two-Dimensional Stock Cutting," *IBM J. Res. Dev.* **16,** 462–469 (1972).
12. G. Ingargiola and J. Korsh, "Reduction Algorithm for 0–1 Single Knapsack Problems," *Management Sci.* **20,** 460–463 (1973).
13. H. M. Salkin and C. A. DeKluyver, "The knapsack Problem: A Survey," *Nav. Res. Log. Quart.* **22,** 127–144 (1975).

Copyright 1977, by INFORMS, all rights reserved. Copyright of Operations Research is the property of INFORMS: Institute for Operations Research and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.