The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation

BERNARD CHAZELLE

***Abstract*—We study implementations of the bottom-left heuristic for two-dimensional bin-packing. To pack N rectangles into an infinite vertical strip of fixed width, the strategy considered here places each rectangle in turn as low as possible in the strip in a left-justified position. For reasons of simplicity and good performance, the bottom-left heuristic has long been a favorite in practical applications; however, the best implementations found so far require a number of steps $O(N^3)$. In this paper, we present an implementation of the bottom-left heuristic which requires linear space and quadratic time. The algorithm is fairly practical, and we believe that even for relatively small values of N, it gives the most efficient implementation of the heuristic, to date. It proceeds by first determining all the possible locations where the next rectangle can fit, then selecting the lowest of them. It is optimal among all the algorithms based on this exhaustive strategy, and its generality makes it adaptable to different packing heuristics.**

***Index Terms*—Bin packing, bottom-left heuristic, computational geometry, geometric pattern matching, operations research, operating systems, scheduling.**

I. INTRODUCTION

MANY problems from operations research or in operating systems involve finding efficient packings of rectangles into a given rectangular area [6]. For this problem known as two-dimensional packing, Baker, Coffman, and Rivest [2] gave a combinatorial model where a rectangular bin R with an open top is to be packed with N rectangles $R_1 \cdots$, R_N, so as to minimize the total bin height. In addition to the requirement that distinct rectangles should not overlap, this model considers only *orthogonal* and *oriented* packings. An orthogonal packing has all the edges parallel to the bottom or vertical edges of the bin R, and it is oriented if rectangles cannot be rotated at all. More precisely, the rectangles are given by a list $L = \{(x_1, y_1), \cdots, (x_N, y_N)\}$, where R_i has width x_i and height y_i. So, in order to have an oriented packing, we must ensure that the edges of length x_i are parallel to the bottom edge of R.

Since the problem of finding an optimal packing is NP-hard, various approximation methods have been proposed [1]–[4]. In this paper we turn our attention to the bottom-left (BL) heuristic. The strategy chosen consists of placing a rectangle into its lowest possible location, and left-justifying it. We then iterate on this process for each rectangle in turn with the order given by L. It has been shown in [2] that although poorly ordered lists L can lead to arbitrarily bad packings relative to an optimal packing, simply ordering L by decreasing widths guarantees a total bin height at most three times the optimal height. In practice, experience has shown that the BL strategy tends to perform fairly well, and its utter simplicity makes it particularly attractive in many applications areas. Unfortunately, its conceptual simplicity does not carry over to the implementation level, and only naive $O(N^4)$ or at best $O(N^3)$ algorithms had as yet been discovered.

We should mention that although shelf-heuristics offer better worst-case performance than the BL-heuristic [3], the latter can be an interesting alternative when the former do not fare well, since it produces packings of a totally different type.

We propose here an optimal scheme for reporting all the possible locations where the next rectangle can be placed. This method requires linear time for each rectangle, and it can be specialized to implement various packing heuristics. In particular we will give a precise description of an implementation of the BL heuristic based on this method, which has an $O(N)$-space, $O(N^2)$-time complexity. The class of heuristics to which this method applies corresponds to the packing procedures which preserve *bottom-left stability*. We say that a rectangle is packed in a bottom-left stable (BL-stable) position if it cannot move downwards or slide to the left. Note that the BL heuristic satisfies this condition. The paper is organized as follows: after an initial, intuitive presentation of the algorithm, we describe the supporting data structure, then we show in detail how to find all the possible locations for a new rectangle to be packed. Next we explain how to update the data structure, once the rectangle has been placed in the bin. Finally, we analyze the space and time requirements of the algorithm, and prove that it is indeed optimal among all the algorithms preserving bottom-left stability.

II. THE DESCRIPTION OF THE BL ALGORITHM

Throughout this section we will consider only packing heuristics which preserve bottom-left stability. At any stage of the heuristic, the bin contains a set of empty spaces (or holes) H_0, H_1, $\cdots$, which can be viewed as polygons with horizontal and vertical edges. There is always one such hole H_0 present in the bin, i.e., the unbounded area lying above all the rectangles placed so far (Fig. 2).

To place the next rectangle $R_i = (l, h)$, the algorithm proceeds by examining each hole H_j in turn and finding whether

Manuscript received December 28, 1981; revised September 19, 1982. This work was supported in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551, monitored by the U.S. Air Force Office of Scientific Research.

The author was with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213. He is now with the Department of Computer Science, Brown University, Providence, RI 02912.

0018-9340/83/0800-0697$01.00

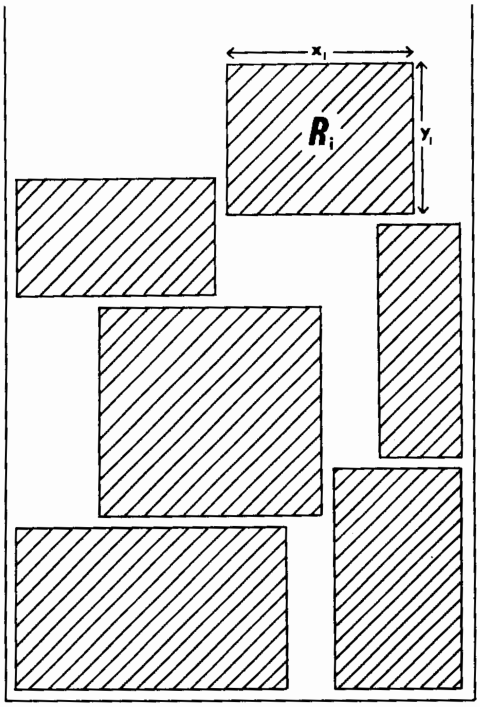


Fig. 1. An orthogonal bin packing.

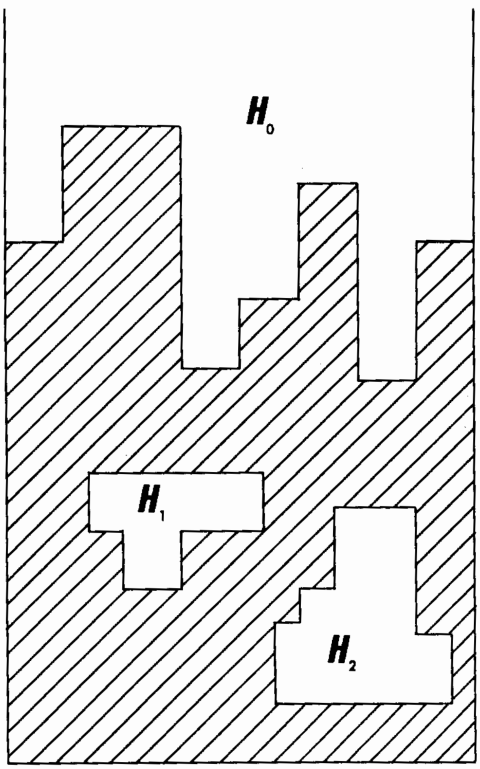


Fig. 2. The general configuration.

R_i fits into it. To visualize the process, we can view the rectangle R_i as a mechanical device consisting of two horizontal bars of length l, held together by a spring pushing outwards (Fig. 3). As a result, the height of the rectangles is as big as possible under the constraint that it must fit within the upper and lower edges of the hole which it is being tested against. As illustrated in Fig. 3, the searching scheme consists of sliding the device from left to right, observing the variation in height and waiting for it to exceed h. Whenever this happens, we can report the current location of the rectangle as a feasible candidate.

It appears that the main problems to solve are as follows.

1) Which data structure should be chosen to represent the holes?

2) How to slide the bars inside the holes?

3) How to update the data structure once a choice for placing R_i has been made?

4) What is the size of the data structure?

A. The Data Structure

We essentially wish to represent each hole with a description of its boundary, so as to allow for left-to-right traversals. Since we are more concerned with simplicity and clarity than with constant factor optimization, we prefer to choose a slight redundant representation. Each hole is represented by a doubly-linked list of its vertices, in the order in which they appear in a traversal of the boundary of the hole. In addition, we must distinguish certain vertices and edges which play a special role.

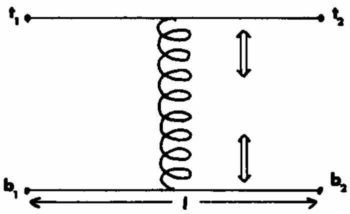
Definition: An edge of a hole is said to be *leftmost* if both of its adjacent edges are horizontal and lie to its right, and neither of the angles they form with the edge is reflex. On the contrary, a *notch* is an edge which displays a reflex angle at each of its endpoints. It is said to be a *left* (respectively *right*, *upper*, *lower*) notch if its two adjacent edges lie to its left (respectively right, above, below). Finally, a vertex with a reflex angle, adjacent to a vertical edge above it and a horizontal edge to the right, is called a *falling corner*.

The definitions above precisely identify the only items which create difficulties in sliding our mechanical device. For this reason, we introduce the concept of a *nice* hole, defined as a hole with no left, right, or upper notch, and with at most one falling corner. Although holes tend not to be nice, in general, we will show that any hole can be partitioned into a collection of nice subholes, arranged in a tree fashion.

Lemma 1: A hole cannot have right-notches or upper-notches, and it has at most one falling corner.

Proof: The first part of the lemma is trivial, since the heuristic must preserve BL-stability. Suppose now that a hole contains two falling corners. Let $a_1a_2a_3 \cdots b$ be the part of the boundary connecting up the two falling corners a_1 and b, given in clockwise order (Fig. 5). We can always assume that a_1 and b are the only falling corners on the polygonal line $a_1 \cdots b$. Since there is no upper-notch, a_3 lies below a_2. Also, to prevent the presence of a right-notch, a_2 and a_3 must be the vertices with minimum X-coordinate among $a_2, a_3, \cdots, b$. This implies that a_3 is a falling corner; hence $a_3 = b$. We conclude by observing that either the rectangle with the edge a_1a_2 or the rectangle with the edge a_2b must be instable, that is, cannot be in such a position using a heuristic preserving BL-stability. □

With this result in hand, we can best explain the partitioning of each hole by completing the description of the data structure associated with each hole. Call $L_1, L_2, \cdots$ the leftmost edges of the hole and $Q_1, Q_2, \cdots$ the top endpoints of the left notches.



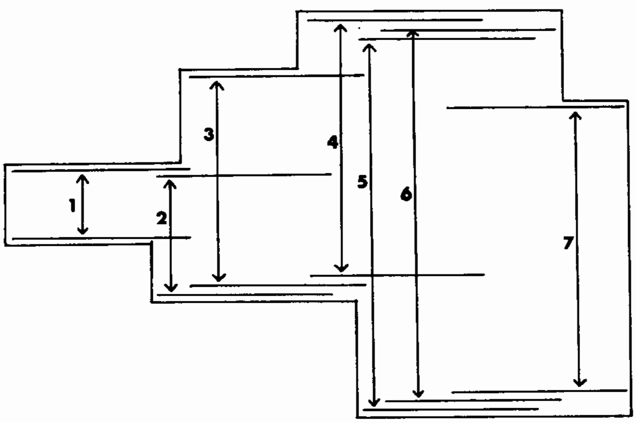


Fig. 3. Testing feasbility with a "spring" device.

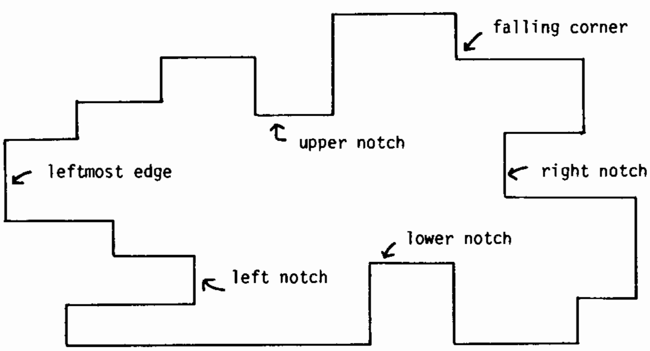


Fig. 4. The special edges of a hole.

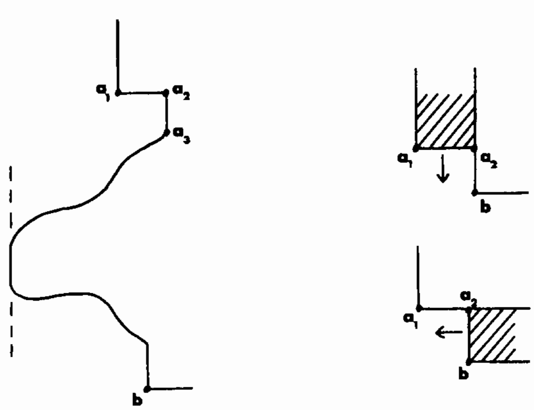


Fig. 5. The proof of Lemma 1.

Besides being doubly linked with its adjacent vertices in the hole, each vertex Q_j will have "special" double links with (Fig. 6)

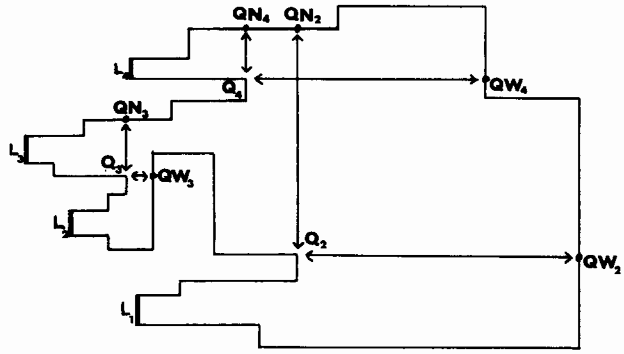


Fig. 6. Setting special links in a hole.

1) QN_j, the point on the boundary which lies immediately above Q_j on a vertical line.

2) QW_j, the point on the boundary which lies immediately to the right of Q_j on a horizontal line.

So, to summarize, every vertex on the boundary of each hole has exactly two double links, except for the special vertices QN_j, QW_j, and Q_j which have respectively 3, 3, and 4 such links. The reason for singling out these vertices will now become apparent.

Consider the boundary of the hole along with the segments of the kind Q_jQN_j. These segments partition the hole into a certain number of subholes, all of which are *nice*. This is a direct consequence of Lemma 1 and of the fact that the partitioning consists essentially in "removing" all left-notches in the hole. It is now apparent that each subhole can be identified as the node of a tree which realizes a partition of the hole into nice subholes. Let us now look at some properties of the data structure in order to motivate the introduction of all the links present in it. From the absence of left- or right-notches, it follows that each subhole has exactly one leftmost (respectively rightmost[1]) edge which is also the vertical edge in the subhole with minimum (respectively maximum) X-coordinate. Putting the previous results together, we conclude that each subhole can be swept across from left to right by a vertical line, starting at its leftmost edge L_i, so that the intersection with the line will always be exactly one segment. Moreover, since there is no upper-notch and at most one falling corner, the top endpoint of the segment can only move upwards as the line moves to the right, except possibly at the only falling corner existing. See Fig. 7 for an illustration of the two kinds of subholes.

With the links $Q_i \leftrightarrow QN_i$ set above, moving the vertical line from left to right, keeping track of the endpoints of the intersecting segment, is a trivial matter. Partitioning the hole into subholes permits us to carry out this procedure for the entire hole, by iterating on this process for each of its subholes, and without duplicating any work. Now that the setting of the links $Q_i \leftrightarrow QN_i$ has been motivated, we can next informally explain the reason for introducing the vertices QW_i by observing that, instead of sweeping a vertical line, we really need to sweep our spring device of width l. Thus, it is important to know which obstacles the right part of the device can encounter when this

[1] A *rightmost* edge is defined symmetrically like a leftmost edge with respect to the right.

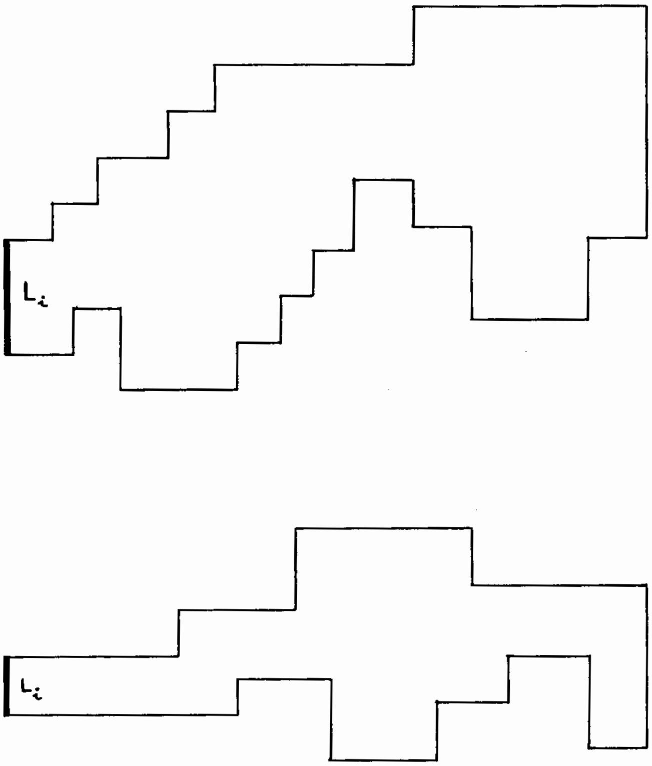


Fig. 7. The two kinds of subholes.

part starts leaving the subhole and the device is still partially inside it. Since both upper- and right-notches are ruled out, QN_i, as defined, is the first and only possible obstacle which will stop the motion of the device.

In conclusion, we have defined a data structure F to represent each hole in the bin as a collection of subholes, each associated in a one-to-one correspondence with a leftmost edge L_i. More precisely, to each L_i is associated two doubly-linked lists $FT(L_i)$ and $FB(L_i)$, containing the vertical upper and lower endpoints, respectively, of the vertical segment sweeping the sub-hole from left to right (Fig. 8). Thus, we have

$$F = \{FT(L_i), FB(L_i) \mid \text{All leftmost edges } L_i \text{ in each hole of the bin.}\}$$

The purpose of partitioning each hole is to allow us to search a hole for packing locations without duplicating any work.

B. Computing All Possible Positions for a New Rectangle

The most difficult task is to slide the lower bar b_1b_2 of the spring device along the edges of the list $FB(L_i)$. Indeed, the possible presence of lower-notches among them may cause the bar b_1b_2 to slide upwards as well as downwards. As long as b_1b_2 slides upwards, it is easy to keep looking at a distance l ahead of b_2 in order to determine the next obstacle forcing b_1b_2 into an upward motion. It follows that sliding the lower bar of the device upwards can be done in linear time. When there is no obstacle causing the bar to slide up, however, we must determine which horizontal edge of $FB(L_i)$ the segment b_1b_2 will next fall upon—see position (2) in Fig. 3. To do so, we use a priority queue Q to make this edge readily available at all

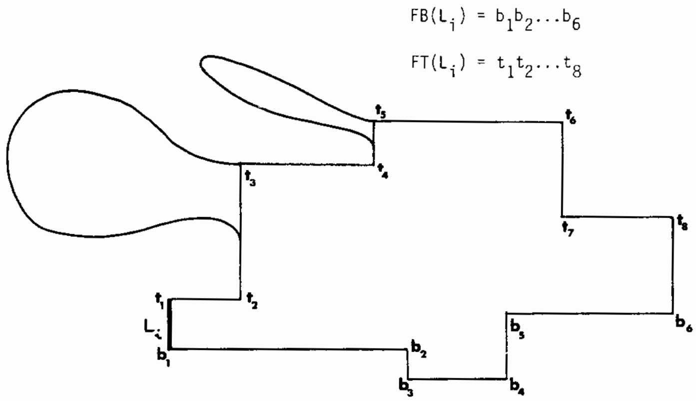


Fig. 8. The data structure F.

times. When this edge becomes a supporting edge for b_1b_2, it must be deleted from Q. Note also that Q must be updated when b_1b_2 slides to the right. Later on, we will show how these updates can be done in linear time. To slide the upper bar t_1t_2 is similar, yet much simpler since the bar can only move upwards, except for one possible encounter with a falling corner. In the case of the unbounded hole, we can assume the presence of a straight horizontal line located at a distance $2h$ over the highest rectangle in the bin. This assumption is made in order to reset the original conditions and thus ensure uniform treatment. Before proceeding with the algorithm, we introduce some notation.

The X- (respectively Y-) coordinate of a point M is denoted $x(M)$ (respectively $y(M)$), so that the point can be referred to as the pair $(x(M), y(M))$. Two sets of points S, S' satisfy the relation $S \leq_x S'$ if $x(M) \leq x(M')$ for any point M (respectively M') in S (respectively S'). Similarly we define the relation $\leq_y$. If d is a number, $M +_x d$ stands for the point $(x(M) + d, y(M))$. Similarly, $M +_y d$ stands for the point $(x(M), y(M) + d)$. Line (S) is the infinite line passing through the segment S.

Consider a subhole in the bin with its associated lists $FB(L_i)$ and $FT(L_i)$. Assume that we conceptually remove the polygonal line corresponding to $FT(L_i)$ so that the subhole becomes an infinite vertical strip with a base delimited by $FB(L_i)$. Assuming then, for the sake of illustration, that there is a force of gravity acting vertically (downwards) on the bar b_1b_2 of length l, we can define C as the locus of the endpoint b_1 for all possible locations of the bar b_1b_2. More precisely, C is the set of points on the polygonal line defined by $FB(L_i)$ such that, if b_1 is placed at any point of C, the segment b_1b_2 fits entirely inside the strip defined above (Fig. 9). Similarly, we define D to be the locus of the endpoint t_1 of the upper bar t_1t_2, when the gravity now acts upwards and $FB(L_i)$ has been removed so that the strip now stretches to infinity downwards. Note that C and D may intersect.

The remainder of the exposition consists of two parts. First we assume that the polygonal lines C and D are available for each subhole in the bin, and we show how to use this information to compute placements for the rectangle R to be packed. Then only, we actually describe the algorithm for computing C and D.

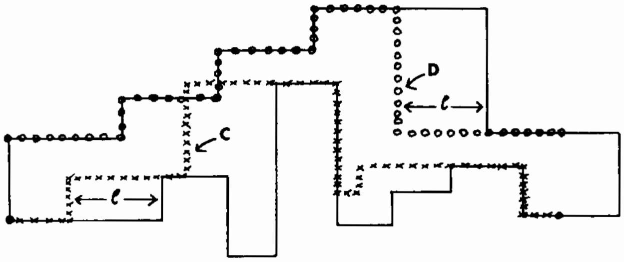


Fig. 9. The loci C and D.

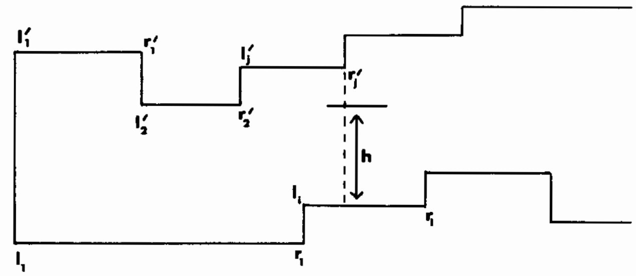


Fig. 10. Reporting feasible positions.

l) Placing a New Rectangle in the Bin: Assuming that both C and D are available as lists of vertices with increasing X-coordinates, it is easy to determine all the BL-stable locations of the rectangle to pack. To do so, we simply compare the height of the polygonal lines C and D, and report the positions where the height of t_1t_2 exceeds that of b_1b_2 by at least h. We can assume that C and D are given by the list of their horizontal edges from left to right (Fig. 10).

$$C = \{(l_1, r_1), \cdots, (l_m, r_m)\}$$
$$D = \{(l'_1, r'_1), \cdots, (l'_p, r'_p)\}$$

We can always assume that r_m and r'_p have the same X-coordinate. Indeed, if this is not the case, the line over-extending to the right can always be trimmed. We first give the algorithm *PLACING* for determining the set E of possible positions for b_1, the lower-left corner of the rectangle (l, h), inside the subhole. Following that description, we analyze the correctness and the complexity of the algorithm.

The list E computed by the function *PLACING* consists of significant points on C, each assigned a label *yes* or *no* to indicate whether they belong or not to the desired set of possible positions for b_1. This set consists, in general, of a number of disjoint polygonal lines which can be easily obtained from E by joining together points with *yes* labels. We omit the details which are straightforward. The function *PLACING* returns the vertices of the polygonal line E in left-to-right order. To best understand the algorithm, we should view it as a set of two coroutines playing symmetrical roles. Thus we may concentrate only on the first while statement. In this case, the vertical edge $r'_jl'_{j+1}$ lies above l_ir_i, therefore a new height difference $y(l'_{j+1}) - y(l_i)$ must be tested against h (Fig. 10). It is clear that the algorithm runs in time $O(m + p)$.

In conclusion, we have described a method for determining the feasible locations of b_1 for each subhole, which is linear in the number of edges present in C and D. We can iterate on this

procedure *PLACING* (h, C, D)

$i \leftarrow j \leftarrow 1$
if $y(l'_1) - y(l_1) \geq h$
  then $E \leftarrow \{(l_1, yes)\}$
  else $E \leftarrow \{(l_1, no)\}$
while $i < m \vee j < p$
  begin
    while $r'_j \leq_x r_i \wedge j < p$
      begin
      $j \leftarrow j + 1$
      $M \leftarrow (x(l'_j), y(l_i))$
      if $y(l'_j) - y(l_i) \geq h$
        then $E \leftarrow E \cup \{(M, yes)\}$
        else $E \leftarrow E \cup \{(M, no)\}$
      end
    while $r_i \leq_x r'_j \wedge i < m$
      begin
      $i \leftarrow i + 1$
      if $y(l'_j) - y(l_i) \geq h$
        then $E \leftarrow E \cup \{(r_{i-1}, yes), (l_i, yes)\}$
        else $E \leftarrow E \cup \{(r_{i-1}, no), (l_i, no)\}$
  end
if $y(r'_p) - y(r_m) \geq h$
  then $E \leftarrow E \cup \{(r_m, yes)\}$
  else $E \leftarrow E \cup \{(r_m, no)\}$

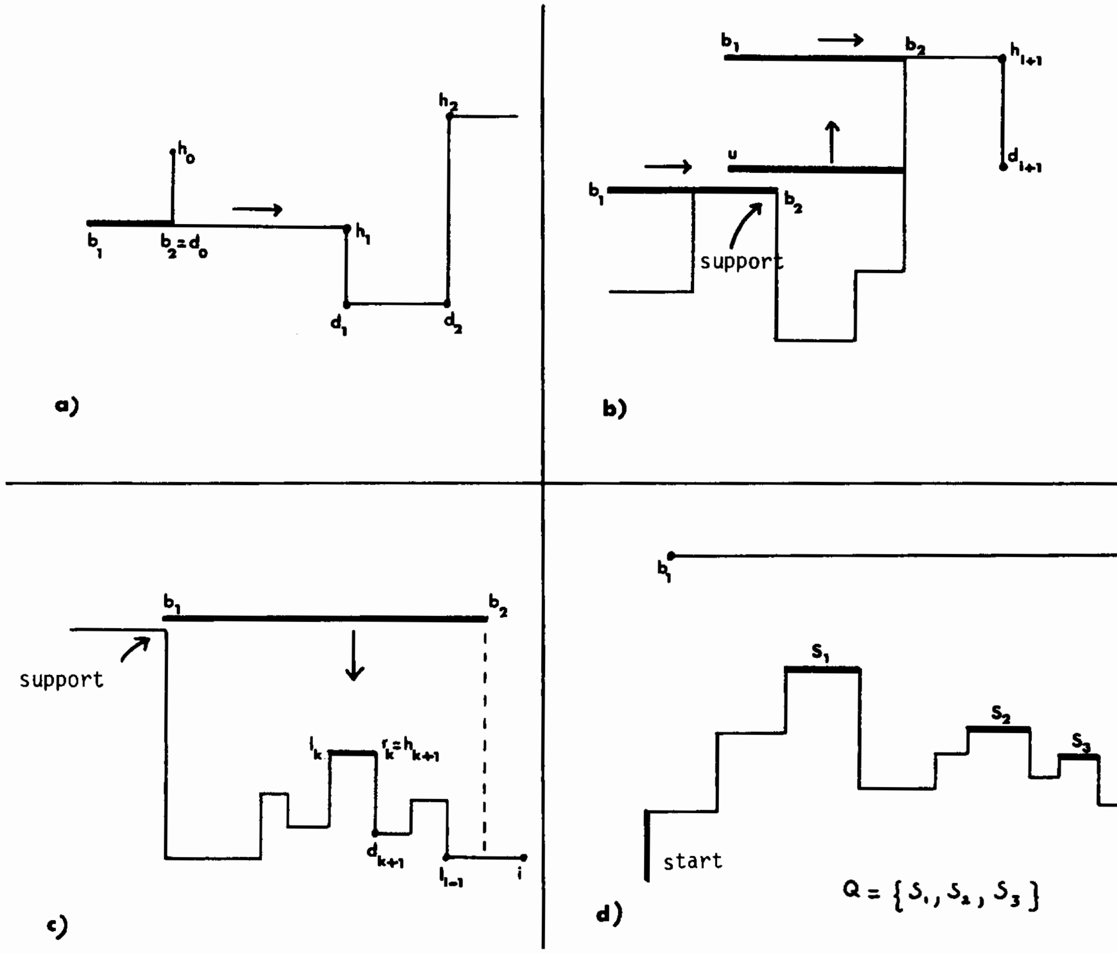


Fig. 11. Generating all feasible positions.

procedure for each subhole in every hole in the bin, and thus report all the feasible locations of b_1 in the bin. It is easy to tailor the function *PLACING* to the BL-heuristic. We simply have to keep the lowest feasible position for b_1 only, and the leftmost one in case of ties. If we disregard the updating of the data structure due to the insertion of the new rectangle, it appears that the time complexity of the entire algorithm is proportional to the added number of edges in the lists C, D for all the subholes. The motivation for partitioning each hole into subholes is now evident, since it allows us to carry out the previous procedure, starting at each leftmost edge of the hole, without duplicating any computation. Thus, in order to have a linear algorithm for inserting a rectangle, it remains to show that

1) C and D can be computed from $FB(L_i)$ and $FT(L_i)$ in time proportional to the size of these lists.

2) Updating the data structure F, after packing a rectangle, can be done in time linear in the size of F.

3) The size of F is, at any time, at most proportional to the number of rectangles present in the bin.

2) Computing C and D: We begin with a description of the method for C, from which the method for D can be easily derived. As mentioned earlier, we will make use of a priority queue Q, implemented as a doubly ended queue [5], whose elements are edges of $FB(L_i)$. TOP1(Q) (resp. TOP2(Q)) gives the value of its first (respectively last) element, while POP1(Q) [respectively POP2(Q)] removes this element from the queue. $s \cup Q$ (respectively $Q \cup s$) appends s to the top (respectively bottom) of Q. For example, if $Q = \{s_1, s_2, s_3\}$, TOP1(Q) $= s_1$, $s \cup Q = \{s, s_1, s_2, s_3\}$, and POP1($Q$) produces the assignment $Q \leftarrow \{s_2, s_3\}$. Whereas sliding b_1b_2 to the right is straightforward as long as the only vertical motion is caused by b_2 hitting an obstacle, the absence of such obstacles will cause b_1b_2 to *fall* upon another supporting edge, which makes matters somewhat more complicated. Note that it is precisely to keep track of these supporting edges that we introduce the queue Q. Let B denote the list $FB(L_i)$ which, for convenience, we assume to be given as a list of vertical edges ordered from left to right $\{(h_0, d_0) \cdots (h_m, d_m)\}$, with $y(h_i) \geq y(d_i)$ for all i; $0 \leq i \leq m$ (Fig. 11). Note that we can also represent $FB(L_i)$ by an ordered list of its horizontal edges $\{(l_k, r_k)\}$, with the correspondence $r_k = h_{k+1}$ [Fig. 11(c)]. If the subhole associated with L_i has a QW_i vertex, we assume that d_m is precisely that vertex, with the understanding that once C has been computed, all the points M such that $Q_i \leq_x M$ should be removed from C. Similarly, for simplicity, the computation of C starts out with irrelevant vertices, which are to be removed later on. C is computed by calling the function BOTTOM(B), which we proceed to describe next. Note that the function BOTTOM calls on three subroutines, SLIDE, SETUP, and MERGE, which are described afterwards The variable *support* is global for all the functions.

```
                    procedure BOTTOM (B)
      (b1, b2) ← (d0 -x l, d0)
      C ← b1
      Q ← ∅
      support ← h1d1 ∩ Line (b1b2)
      SLIDE (1)
      Remove from C all points M; M ≤x d0 or Qi ≤x M.
                    procedure SLIDE (start)
while start ≤ m
begin
   i ← start
   while hidi ≤x support +x l
   begin "slide on support"
        if y(hi) > y(b2)
              then   "hit hidi [Fig. 11(b)]"
                     (b1, b2) ← (hi -x l, hi)
                     u ← b1 -y [y(hi) - y (support)]
                     C ← C ∪ {u, b1}
                     Q ← ∅
                     support ← hi+1di+1 ∩ Line (b1b2)
                     SLIDE (i + 1)
                     stop
        else if i = m
              then stop
              else "keep sliding"
                       i ← i + 1
   end
   "ready to fall [Fig. 11(c)]"
   b1 ← support
   b2 ← b1 +x l
   C ← C ∪ {b1}
   Q' ← SETUP (start, i)
   MERGE (Q, Q')
   a ← POP1(Q)
   "Let lk (respectively rk) be the left (respectively right) endpoint of the segment a."
   (b1, b2) ← (b1, b2) -y [y(b1) - y(lk)]
   C ← C ∪ {b1}
   start ← i
   support ← rk
end
                    procedure SETUP (start, end)
"By convention, if Q = ∅, then
TOP1(Q) ≤y M for any point M in the
structure. Q is here a local variable."
Q ← ∅
i ← end - 1
while i ≥ start
   begin
        if TOP1(Q) ≤y liri
              then Q ← liri ∪ Q
   end
   return (Q)
                    procedure MERGE (Q, Q')
   if Q ≠ ∅ ∧ Q' ≠ ∅
      then
           s ← TOP1(Q')
           while TOP2(Q) ≤y s begin POP2(Q) end
   Q ← Q ∪ Q'
```

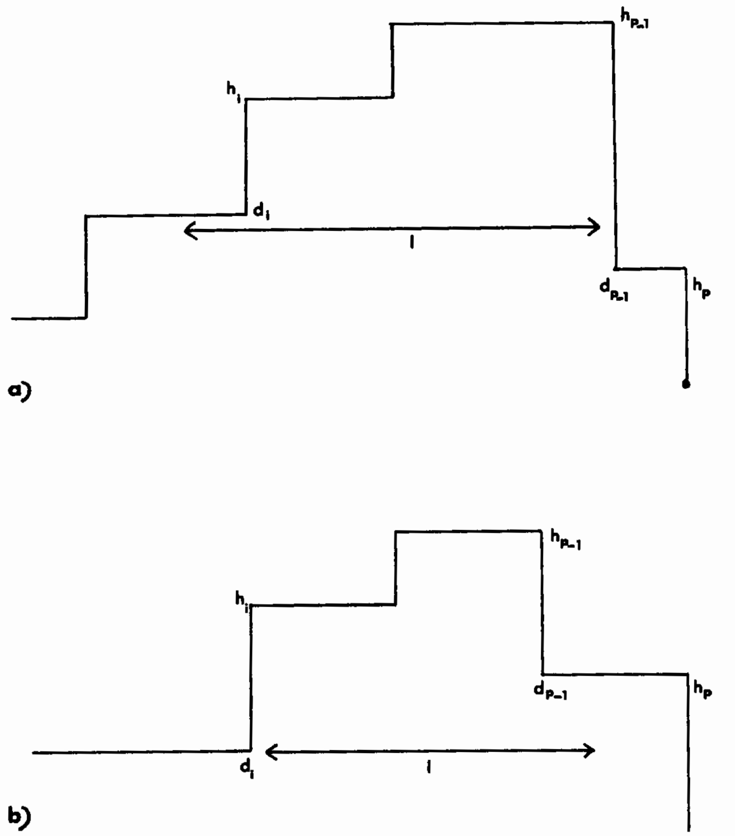


Fig. 12. The function TOP.

We next review the algorithm in detail to show its correctness and analyze its complexity. After a phase of initialization [Fig. 11(a)], the function SLIDE is called upon. The variable *support* is the rightmost point common to b_1b_2 and B. Thus b_1b_2 can slide on this point over a distance at most l. The second while statement checks whether any h_id_i in that range can be an obstacle to the sliding of b_1b_2. If one is found, the **if** statement is true and b_2 can be brought up to the position of h_i [Fig. 11(b)]; *support* is then updated to h_{i+1}. On the other hand, if no obstacle is found, b_1b_2 can slide to the right by a distance l, and must then move down to the level of the highest edge below it [Fig. 11(c)]. TOP1(Q) is precisely this edge. Updating b_1b_2 and C is then trivial.

We next turn to the two functions used to maintain Q: SETUP and MERGE. When b_1b_2 leaves the edge TOP1(Q) on which it is sitting, this edge must be deleted from Q and the next highest edge must be made readily available. To do so, we must consider all the horizontal edges of B lying below b_1b_2. Then we define Q as the subsequence of these edges with the Y-coordinate increasing from right to left [Fig. 11(d)]. The queue Q, so defined, is clearly sufficient for our purposes. At the beginning, the function SETUP (*start*, *end*) computes Q from scratch, knowing that b_1b_2 is enclosed between the verticals passing through h_{start} and h_{end}. The algorithm is fairly straightforward and does not necessitate more explanation. Since it is prohibitive to recompute Q from scratch every time b_1b_2 moves down, we compute only the queue Q' corresponding to the part of B newly scanned, and we merge Q' with the former Q. Since the function SLIDE always keeps track of the interval most recently scanned (i.e., [start, i]), Q' can be computed easily. To MERGE Q' with Q, we need only to look at the edges of Q which must be removed from the queue. Note that the MERGE actually consists of removing elements from Q, if necessary, then concatenating Q and Q'. A doubly-linked implementation of the deque is sufficient for performing these operations in constant time, and since removed elements will never reappear later on, the overall queue management takes $O(m)$ time. The final task of BOTTOM is to remove all the points of C which lie on the left of h_0d_0. From the remarks above, it is clear that the function BOTTOM has an $O(m)$ execution time.

Computing the locus D of the left endpoint t_1 of the upper bar can be done in a similar manner. However, since the polygonal line $FT(L_i)$ has a stair-like form, except possibly for the unique falling corner of the hole, the algorithm TOP for computing D can be made much simpler than its counterpart BOTTOM. Let T be the list of vertical edges of $FT(L_i)$: $T = \{(h_0, d_0), \cdots, (h_p, d_p), s\}$. The variable s is a flag set to 1 if T contains a falling corner (d_{p-1}) and 0 otherwise. The function TOP is self-explanatory. The list D consists essentially of the vertices of $FT(L_i)$, except possibly for the last vertices which may not be reachable by the upper bar because of a falling corner. There are two possible cases, as illustrated in Fig. 12, and a specific treatment is required to handle them and compute the last vertices of D. In a post-processing stage, the function trims the list D by keeping only the points whose X-coordinates lie between the minimum and maximum X-coordinates of C. This is due to an idiosyncrasy of the function TOP which, for the sake of simplicity, starts out with a line D placed, on purpose, too far to the left.

```
procedure TOP (T)
if s = 0 then "no falling corner"
        D ← {h_0, d_1, h_1, ···, h_p}
        stop
D ← {h_0 -_x l}
i ← 1
while i < p
    begin
        if d_{p-1} ≤_y d_i ∧ d_{p-1} ≤_x d_i +_x l
            then "Fig. 12(a)"
                u ← (x(d_{p-1}) - l, y(d_i))
                v ← d_{p-1} -_x l
                D ← D ∪ {u, v, h_p -_x l}
                stop
        D ← D ∪ {d_i}
        if d_i ≤_y d_{p-1} ≤_y h_i ∧ d_{p-1} ≤_x d_i +_x l
            then "Fig. 12(b)"
                u ← (x(d_i), y(d_{p-1}))
                D ← D ∪ {d_i, u, h_p -_x l}
                stop
        D ← D ∪ {d_i, h_i}
        i ← i + 1
    end
```

Remove from D all points M; $M \leq_x C$ or $M \geq_x C$.

This completes the computation of C and D. We should observe that the algorithm is valid regardless of the shape of the new rectangle to be packed. If, however, we wish to use it to implement the BL-packing heuristic with decreasing widths,[2] we may take advantage of this geometric feature, and simplify the procedure SLIDE. Indeed, it is easy to see that with the decreasing width requirement, before packing the rectangle of width l, no lower-notch in the bin may have a width under l, nor can the length of BC be smaller than l, if A, B, C, D are consecutive vertices of a hole, given in clockwise order, with AB, BC, and CD going respectively down, left, and down. From this simple observation, it follows that when in SLIDE the lower bar is ready to fall down, the lower boundary within horizontal distance l of the point SUPPORT must have the shape of a descending staircase, except possibly for a final rising step (see the analogy with procedure TOP and falling corners). Thus, it can be decided at once whether b_1 falls to A or to $(x(A), y(C))$, depending on the relative heights of AB and CD (Fig. 13). As a result, SLIDE can proceed straight from left to right, and does not need the deque Q.

C. Updating the Structure

The effect of packing a rectangle may be either simply to reduce the sixe of one hole, or to subdivide a hole into smaller holes. To translate this effect into the structure F, the first task to accomplish is to determine whether the upper-right corner of the rectangle is a vertex of the kind Q_i, and if yes, compute the points QN_i and QW_i associated with it. This procedure requires time $O(|F|)$, since it can simply proceed by testing all the edges of F against the lines supported by the sides of the rectangle. Next, for each subhole in turn, we traverse its boundary in, say, clockwise order, testing every edge visited against the edges of the rectangle. When an intersection is encountered, we know that the current subhole must be subdivided and new holes or subholes must be created. To ensure the appropriate updating, it suffices to continue traversing the boundary of the new subholes, which can be ensured by the simple policy of *never crossing* any edge in the structure, i.e., always keeping the interior of the subholes on the right.

To go from one new subhole to another new subhole, we must cross an edge of the new rectangle, from where we can repeat the procedure sketched above. The detection of the new edges L_i and of the new subholes will simply follow through, as well as the updating of all the new *special* links. We omit the details since the procedure involves only straightforward graph manipulation which lies outside of our purpose here. Indeed a complete description of all possible situations would entail a simple, yet rather lengthy case analysis. The procedure which we have just described requires time linear in the size of F, so our next task must be to find an upper bound on that size.

D. The Size of the Data Structure

As we have seen earlier, both of the procedures that operate on F, i.e., compute the location of the next rectangle and update the data structure, run in time proportional to the size of F. Therefore, to achieve an overall performance of $O(N)$ time per rectangle-insertion, we must show that the size of F is at most proportional to the number of rectangles packed in the bin.

[2] Recall that, without the decreasing width requirement, the performance of the heuristic can be arbitrarily bad compared to optimal [2].

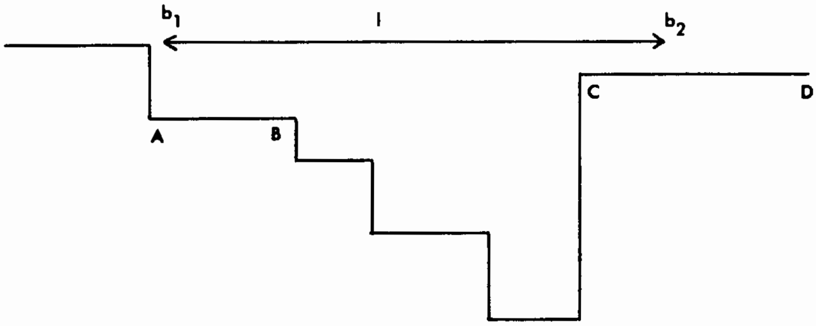


Fig. 13. Introducing the decreasing width requirement into the algorithm.

Lemma 2: At any step of the packing heuristic, the size of F is $O(p)$, where p is the number of rectangles in the bin at this step.

Proof: Since any vertex Q_i has only two "special" links, i.e., $Q_i \leftrightarrow QN_i$ and $Q_i \leftrightarrow QW_i$, it suffices to show that the total number of edges for all the holes is $O(p)$. We can easily show that this number never exceeds $4p$. To see that, we simply observe that a rectangle which is being placed can contribute to the total number of edges in two ways. It may add edges of its own or it can split existing edges. An edge of F, already present in the structure before the insertion of the rectangle, may be split into two or three parts by an edge of the rectangle. In the first case, one part will be deleted, so the edge count will not change. In the second situation, one more edge will be introduced as a result, but to achieve this configuration, the rectangle must sacrifice one of its edges, the effect of which is to restore the edge count to its former value, set initially to 4 + its value before insertion. Similarly, we observe that if an edge of the rectangle is split into k parts, at most $\lceil k/2 \rceil$ of them are visible, hence part of F. Also between any pair of consecutive visible edges must lie an edge present in F before insertion and bound to disappear afterwards. As a result, the increase of one in the edge count related to the edge of the rectangle is sufficient to account for this situation. It follows that, in all cases, the total increase in the edge count is certainly bounded by 4. This crude analysis could actually be improved to lower the factor, but this is not necessary for our purposes. □

Combining this result with the fact that computing the lists C and D for each subhole, determining the feasible locations for a new rectangle to pack, and updating the data structure can all be done in time linear in the size of F, we conclude that it is possible to determine all the BL-stable positions for the pth rectangle to pack in the bin in time $O(p)$. This proves our main result:

Theorem 3: It is possible to pack N rectangles in a bin with the bottom-left heuristic, in time $O(N^2)$, using $O(N)$ space.

E. Optimality Considerations

Using a simple output argument, we can show that the method described above is optimal for any scheme requiring an explicit enumeration of all the possible BL-stable positions for the next rectangle. It suffices to prove that with any packing heuristic preserving BL-stability, once p rectangles have been placed in the bin, the set of possible positions for the next rectangle may involve $\Omega(p)$ vertices. We define the list L of N rectangles as a list of squares with the following characteristics:

$$L = \{(w/2, w/2), (w/4, w/4), \cdots, (w/2^N, w/2^N)\}.$$

We will show that when the first p rectangles have been placed, the unbounded hole (which is unique) has $\Omega(p)$ edges, all of which are part of E. Recall that E is the list of all feasible locations of the lower-left corner of the rectangle. It is easy to prove by induction that before the $(p + 1)$st rectangle is inserted, no pair of parallel edges in the bin can be found at a distance strictly less than $w/2^p$ from each other. As a result, the next rectangle cannot cover an exposed edge entirely, so it cannot decrease the total number of edges already exposed, and it itself will have exactly two of its edges exposed. This implies that before inserting this rectangle, E contains at least $2p$ edges, which completes our proof.

III. Conclusions

We have now achieved our main goal, which was to present an $O(N^2)$-time, $O(N)$-space bottom-left packing algorithm. The quadratic term corresponds to the worst-case performance of the algorithm, so we can expect the inclusion of simple heuristics to better the average time performance significantly. For example, instead of performing the search procedure on each subhole separately, we may perform partial searches on each of them in order to ensure a Y-coordinate progression. As a result, the first feasible location found is guaranteed to be the lowest as well, and all subsequent searches will thus be avoided. It is also conceivable to keep a small number of parameters for each subhole, in order to avoid searching them when the rectangle to insert is blatantly non-candidate. For instance, a pair (*WidthMax*, *HeightMax*) can be associated with each subhole to signify the maximum width and height of any rectangle which can fit into the subhole.

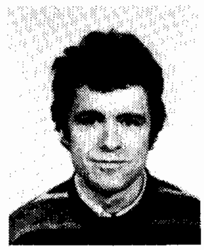
We feel that beyond its practical interest and its relevance to bin-packing, the method described in this paper gives new techniques for computational geometry. In particular, we define a tree-representation of a rectilinear polygon, and we use it to facilitate a type of searching strongly reminiscent of pattern matching. Similar techniques may also be applied to other problems in pattern recognition, where typically a geometric figure is to be tested for containment against a more complex rectangular subdivision of the plane.

ACKNOWLEDGMENT

I wish to thank J. Bentley, who suggested this problem to me and sparked my interest in it. My gratitude also goes to one of the referees for pointing out the simplification in the procedure SLIDE, possible when rectangles are packed with decreasing widths.

REFERENCES

[1] B. S. Baker, D. J. Brown, and H. P. Katseff, "A 5/4 algorithm for two-dimensional packing," *J. Alg.*, vol. 2, pp. 338–368, 1981.
[2] B. S. Baker, E. G. Coffman, and R. L. Rivest, "Orthogonal packings in two dimensions," *SIAM J. Comput.*, vol. 9, pp. 846–855, 1980.
[3] B. S. Baker and J. S. Schwarz, "Shelf Algorithms for two-dimensional packing problems," in *Proc. 1979 Conf. Inform. Sci. Syst.*, Baltimore, 1979.
[4] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan, "Performance bounds for level-oriented two-dimensional packing algorithms," *SIAM J. Comput.*, vol. 9, pp. 808–826, 1980.
[5] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1968.
[6] J. D. Ullman, "Complexity of sequencing problems," in *Computer and Job-Shop Scheduling Theory*, E. G. Coffman, Ed. New York: Wiley 1975.



Bernard Chazelle received the Diplôme d'ingénieur from the Ecole Nationale Supérieure des Mines de Paris, France, in 1977, and the M.S. and Ph.D. degrees in computer science from Yale University, New Haven, CT, in 1978 and 1980, respectively.

From 1980 to 1982 he was a Research Associate in the Department of Computer Science at Carnegie-Mellon University, Pittsburgh, PA. In September 1982, he joined the Department of Computer Science at Brown University, Providence, RI, where he is currently an Assistant Professor. His research interests include analysis of algorithms, complexity theory, computational geometry, VLSI, and graphics.

Dr. Chazelle is a member of the Association for Computing Machinery.

Reduction of Connections for Multibus Organization

TOMÁS LANG, MATEO VALERO, AND MIGUEL A. FIOL

Abstract—**The multibus interconnection network is an attractive solution for connecting processors and memory modules in a multiprocessor with shared memory. It provides a throughput which is intermediate between the single bus and the crossbar, with a corresponding intermediate cost.**

The standard connection scheme for the multibus connects all processors and all memory modules to all buses. This connection scheme is redundant and expensive for a relatively large number of buses.

Reduced connection schemes that produce the same throughput as the standard connection are presented. The schemes are optimal with respect to the number of connections, are easy to arbitrate, reliable when a bus fails, and expandable. The reduction is specially significant when the number of buses is relatively large, being of 25 percent when this number is half the number of memory modules.

***Index Terms*—Arbitration, connection reduction, interconnection network, multiple buses, multiprocessors.**

Manuscript received November 11, 1981; revised December 17, 1982.

T. Lang was with the Facultat d'Informatica, Universitat Politecnica de Barcelona, Barcelona, Spain. He is now with the Department of Computer Science, University of California, Los Angeles, CA 90024.

M. Valero is with the Facultat d'Informatica, Universitat Politecnica de Barcelona, Barcelona, Spain.

M. A. Fiol is with the School of Telecommunication Engineering, Universitat Politecnica de Barcelona, Barcelona, Spain.

I. INTRODUCTION

ONE of the many important aspects to consider in the design of multiprocessor systems is the structure of the network connecting the processors to the shared memory modules. Many parameters have a bearing on this choice. Among them: reliability, cost, modularity, bandwidth, number of processors, and expandability.

Several interconnection networks have been proposed, such as the crossbar [1], single bus [2], multibus [3], [4], and other special interconnection networks [5]. There are several analytic models to assess the performance of the various topologies under different processor demand patterns [3], [6], [7].

The multibus interconnection is an attractive solution for connecting processors and memory modules in a multiprocessor with shared memory. It provides a throughput which is intermediate between the single bus and the crossbar, with a corresponding intermediate cost. Moreover, if the processor requests are independent and uniformly distributed among the memory modules, the amount of memory conflicts makes the throughput obtained with the crossbar roughly the same as that obtained with the multibus with a number of buses slightly larger than half the number of processors [4].

0018-9340/83/0800-0707$01.00