AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Optimum Scan-Width Selection Under Containment Constraints

By M. R. GAREY* and R. Y. PINTER*

(Manuscript received December 20, 1983)

We consider the following algorithmic problem, which arises in connection with optimally choosing beam widths and positions for electron exposure of integrated circuit wafers. Let $H > 0$ be a fixed real number, and let c be a fixed, positive-valued, nondecreasing cost function defined on $(0, H]$. An instance of the problem consists of a given range $R = [a, b]$ and n given intervals $I_i = [a_i, b_i]$, $1 \leq i \leq n$, each contained in R and of length not exceeding H. A solution (or feasible solution) for such an instance is a collection of segments, $S_1, S_2, \cdots, S_k$, each of length at most H and contained in R, such that each given interval is contained in at least one segment and the union of all the segments is R. The goal is to find an *optimal solution* with respect to c, i.e., a solution for which the sum of the costs of the individual segment lengths is as small as possible. The segments in a solution describe the beam positions and widths, projected on one side of the wafer, and the given intervals correspond to particularly sensitive regions on the layout mask, each of which must be entirely exposed by a single scan. The cost function gives the time required for a single scan of given width, including alignment overhead. Using dynamic programming techniques, we give efficient algorithms and data structures for solving this problem for several natural classes of cost functions, the most general of which is the class of all concave increasing functions, solved by an algorithm that runs in time $O(n^2)$.

I. INTRODUCTION

Electron lithography systems[1] are used in the fabrication of integrated circuits to expose areas of the manufactured device specified

* AT&T Bell Laboratories.

by layout masks. In most such systems an electron beam repeatedly sweeps across the mask in parallel horizontal movements, exposing one horizontal stripe of the material at a time, so that each point of the mask is included in at least one stripe. The height of the stripe can vary on successive sweeps and is subject to a physically determined upper bound, H, with the time required for a sweep being a nondecreasing function of height. In general, it is desirable to process each mask in the minimum possible amount of time in order to maximize throughput and minimize the risk of failure.

The quality of the etching made by a single sweep is highly reliable, and the generated stripe can be regarded as an atomic piece of the process for purposes of quality control and cost evaluation. However, the machinery cannot be realigned between sweeps with absolute precision, and errors can arise from the resulting imperfect positioning of the stripes. Usually, conservative design rules prevent such alignment errors from being harmful, but some parts of a circuit mask may be too sensitive to tolerate these (otherwise acceptable) errors. These more sensitive regions ("islands") cannot tolerate either small unexposed gaps between stripes or multiple exposures created by overlapping stripes, although an island can be shielded during some sweeps to prevent exposure during those sweeps. Thus, the islands impose containment constraints on the scanning process, in that each island must be completely contained within a single stripe. This paper focuses on how to minimize the cost (time) for exposing a two-dimensional region in the presence of such containment constraints.

Notice that the upper bound, H, on the height of any stripe implies that no island can have height greater than H. It also implies that, in general, some parts of the region may have to be scanned more than once. For example, if there are two islands of height $3H/4$ arranged as in Fig. 1, the stripe exposing the first must overlap and be distinct from the stripe exposing the second. Hence, each of the islands would need to be shielded during the sweep that exposes the other island.

The total cost of the scanning process is the sum of the costs for the individual sweeps, where the alignment overhead is incorporated into the cost for the sweep. Various cost functions can be used to approximate the actual cost of scanning a stripe of given height. We will examine several classes of such functions and present efficient algorithms that minimize the covering cost for them. All cost functions considered will be monotonically nondecreasing in the stripe height.

Section II gives the mathematical formulation of the scan-width selection problem and makes some preliminary observations. Section III describes our optimization algorithms for the various cost functions. Section IV concludes the paper by mentioning several open problems.

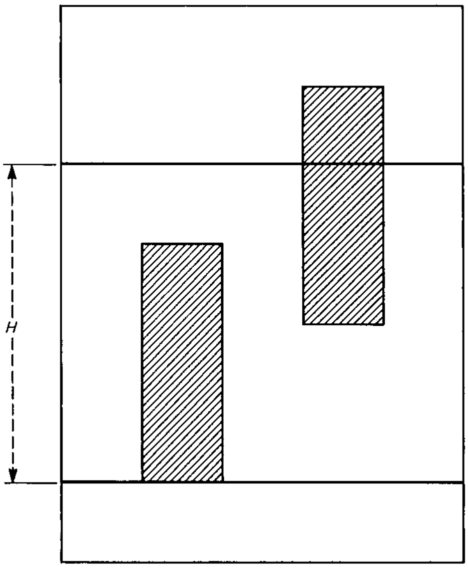

Fig. 1—Layout mask showing that two islands of height $3H/4$ require overlapping stripes.

II. PROBLEM FORMULATION

We first observe that the problem we have described is essentially one-dimensional. We can project the region to be exposed and the islands contained in the region onto a vertical line, transforming the region into a *range* $[a, b]$ and each island into an *interval* $[a_i, b_i]$ contained in this range. (See Fig. 2.) A stripe of height $h \leq H$ then becomes a *segment* of length $h \leq H$, and containment of an island within a stripe corresponds to containment of the associated interval within the appropriate segment. We will deal exclusively with this one-dimensional formulation in the remainder of the paper, rotating it 90 degrees for convenience of description.

Problem definition: Let $H > 0$ be a fixed real number, and let c be a fixed, positive-valued, nondecreasing cost function defined on $(0, H]$. An instance of the problem consists of a given range $R = [a, b]$ and n given intervals $I_i = [a_i, b_i]$, $1 \leq i \leq n$, each of length at most H and contained in R. A solution (or feasible solution) for such an instance is a collection of segments, $S_1, S_2, \cdots, S_k$, each of length at most H and contained in R, such that each given interval is contained in at least one segment and the union of all the segments is R. Our goal is to find an *optimal solution*, i.e., a solution for which the sum of the costs of the individual segment lengths is as small as possible.

We will assume that the specified intervals are given in a particular

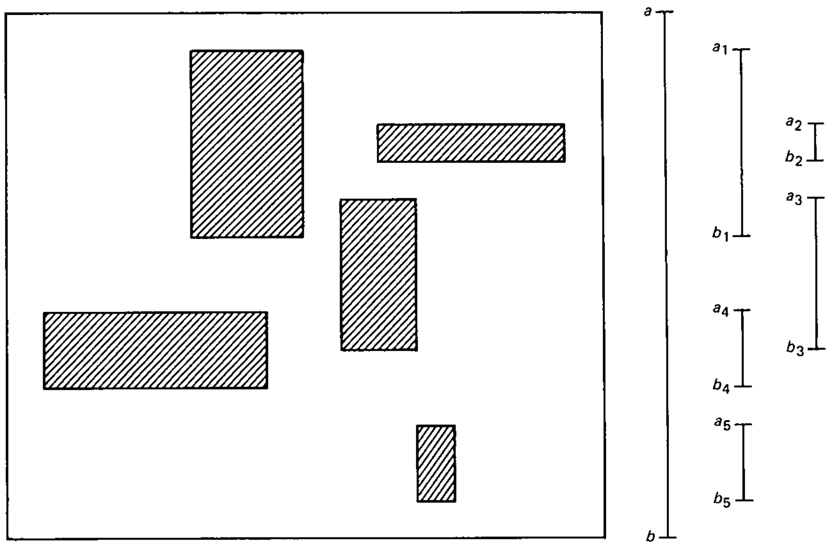

Fig. 2—A covering problem and its one-dimensional interpretation.

sorted and reduced format. First, we assume that the intervals are sorted by their left end points, so that $a_i \leqslant a_{i+1}$ for $1 \leqslant i < n$. Second, we assume that no given interval is contained in another given interval; any segment that contains the larger one must also contain the smaller, so the smaller can be ignored without loss of generality. Thus, the intervals will also be sorted by their right endpoints, i.e., $b_i \leqslant b_{i+1}$ for $1 \leqslant i < n$. We shall not account for the complexity of the preprocessing needed to meet these assumptions, which is $O(n \log n)$ if sorting is necessary or $O(n)$ if the intervals are presented in sorted order.

We will consider the following types of cost functions (all are assumed to be positive and nondecreasing on $(0, H]$):

1. $c(h)$ is constant, i.e., $c(h) = \beta$.
2. $c(h)$ is proportional to h, i.e., $c(h) = \alpha h$.
3. $c(h)$ is linear, i.e., $c(h) = \alpha h + \beta$.
4. $c(h)$ is concave, i.e., if $h_1 < h_2 < H$ and $0 < \epsilon \leqslant \min\{h_1, H - h_2\}$, then $c(h_1) + c(h_2) \geqslant c(h_1 - \epsilon) + c(h_2 + \epsilon)$.

We will also consider the variant on the constant cost function in which all segments must have length exactly H. This is not a nondecreasing function, since segments shorter than H have essentially infinite cost, but we shall see that the solution for this case follows directly from the solution for the case of constant cost functions.

In the general case of a concave cost function, we will assume that the function is given simply by a "black box" subroutine for computing $c(h)$. We will also assume that the running time for the subroutine is bounded by some constant, γ; if the calculation is more complex and a more detailed timing analysis is needed, this should be easily obtainable from our analysis.

We conclude this section with a general normalization lemma for optimal solutions. Let us define a *gap* in a given problem instance to be a maximal subinterval of $R = [a, b]$ that is disjoint from all the given intervals. We regard a gap as an open subinterval, i.e., the two endpoints of a gap do not belong to the gap. A maximal subcollection of the given intervals such that no two are separated by a gap will be called a *block*; by our assumption that the intervals are sorted, each block consists of intervals whose indices are consecutive. Thus, the given problem instance is partitioned into an alternating sequence of gaps and blocks, with the indices of all intervals in each block being less than the indices of all intervals in blocks to its right. If no interval starts at the left end, a, of the range, we will regard a as both the left end of the corresponding gap and as the right end of an empty block. Similarly, if no interval ends at b, we will regard b as both the right end of the corresponding gap and as the left end of an empty block. Our "normalization lemma", which will be used for all but the case of constant cost functions (and its fixed length variant), follows.

Lemma 1: For any nondecreasing cost function, c, and any given problem instance, there always exists an optimal solution in which:

(1.1) No segment is contained in another segment.

(1.2) All segments start at left ends, a_i, of intervals, right ends of blocks, or in gaps; and all segments end at right ends, b_i, of intervals, left ends of blocks, or in gaps.

(1.3) No point in a gap is in the interior of more than one segment.

Proof: For (1.1) we simply observe that any segment that is contained in another can be deleted with no increase in cost and without destroying the solution. Thus, any optimal solution that also has a minimum number of segments (among all optimal solutions) must satisfy (1.1).

For (1.2) consider any optimal solution with a minimum number of segments [hence, satisfying (1.1)]. If the left end of any segment is not of the specified form, we can shorten the segment to start at the leftmost point of that form that it contains, without changing the set of intervals contained in the segment and without leaving uncovered any portion of a gap. Thus, the new solution is still a solution, and since the cost cannot have increased, it remains an optimal solution. Similarly, we can shorten the right end of any segment whose right endpoint fails to have the specified form. Repeating these operations to the given optimal solution results in a new optimal solution satisfying (1.2). Furthermore, it must still satisfy (1.1), since the number of segments in the solution has not been increased (if the shortening of any segment were to make it contained in another segment, we could delete the smaller, which would be a contradiction of our assumption that the number of segments was originally minimal—hence, this cannot happen).

For (1.3) consider any optimal solution obtained as in the preceding paragraph, and suppose some point in a gap is in the interior of two segments. Let the segments be $[x, y]$, $[z, w]$, with $x < z < y < w$. Since some point, u, in the range (z, y) is in a gap, we can replace these two intervals by the two intervals $[x, u]$, $[u, w]$ without destroying the solution and without increasing the solution cost, since both intervals have been shortened. Moreover, this operation clearly preserves (1.1), because the number of segments remains minimum, and preserves (1.2), because the only new point to start or end a segment is u, which is in a gap. Thus, repeating this operation will eventually result in an optimal solution satisfying (1.1), (1.2), and (1.3). □

We will restrict attention to finding optimal solutions of the form given in Lemma 1. Notice that (1.1) of the lemma allows us to order the segments in a solution from left to right in the same way intervals were ordered; if the left end of a segment is less than the left end of another segment, then its right end must also be less than the right end of the other segment. Thus, there will be no confusion when we say a segment is to the left of another segment.

III. OPTIMIZATION ALGORITHMS

We now consider each of the four types of cost functions, in order of increasing difficulty.

3.1 Constant cost function

In this case we are simply trying to minimize the number of segments, so we may, without loss of generality, restrict all segments to the maximum possible length H. (We will not be using Lemma 1 here.) The solution then follows immediately by observing that we can always start the first segment at the left endpoint, a, of the range $[a, b]$. This leaves a remaining problem that includes exactly those intervals not contained in the first segment, i.e., those intervals $[a_i, b_i]$ for which $b_i > a + H$; and the range for the new problem can be taken to be $[a', b]$, where a' is the minimum among $a + H$ and the left endpoints, a_i, of the remaining intervals. Because we can always start the first segment at the left end of the range, we can repeat this "greedy" approach until we reach the end of the range $[a, b]$. However, the last segment must be started at the point $b - H$ to keep it within the original range. The following algorithm implements this approach and runs in time linear in the larger of the number of given intervals and the number of segments in the optimal solution. The variable s_j

denotes the left endpoint of the jth segment, and the final value of j is the number of segments in the optimal solution.

Algorithm 1:

```
initialize $i := 1$; $j := 0$; $newa := a$;
while $newa \leqslant b - H$ do
  $j := j + 1$; $s_j := newa$;
  while ($b_i \leqslant newa + H$) do $i := i + 1$;
  $newa := \min(a_i, newa + H)$;
if ($newa < b$){$j := j + 1$; $s_j := b - H$};
```

3.2 Proportional cost function

For the sake of simplicity, we shall assume that the constant of proportionality α has been normalized to 1, so the cost of a solution is just the sum of the lengths of its segments. We first observe that the gaps can be covered independently of the blocks, with each gap being covered simply by a sequence of adjacent segments whose lengths sum to the length of the gap. The lengths of these segments can be chosen arbitrarily, but all must be H or less. Thus, we need only show how to solve problem instances that consist of a single block, starting at the left end of the range and running all the way to the right end.

So, suppose the given problem consists of just a single block. Since there are no gaps, Lemma 1 tells us that we can restrict attention to solutions in which all segments start at left ends of intervals and run to right ends of intervals. This sets the stage for the use of a dynamic programming approach. Let $C(i)$, $1 \leqslant i \leqslant n$, be the cost of an optimal covering for the range $R(i) = [a_i, b]$ and the intervals $[a_j, b_j]$, $i \leqslant j \leqslant n$, contained in that range. We want to find $C(1)$. Then we can write

$$C(i) = \min_{\substack{j \geqslant i \\ b_j - a_i \leqslant H}} \{b_j - a_i + C(j + 1)\}, \qquad (1)$$

where we artificially set $C(n + 1) = 0$. The solution corresponding to a particular value of j consists of the segment $[a_i, b_j]$ followed by an optimal solution for the range $[a_{j+1}, b]$. (Notice that $a_{j+1} \leqslant b_j$ since there are no gaps). By using (1) to solve for the $C(i)$ in order of decreasing i, we can then find $C(1)$. The segments that realize the solution can be recorded in a one-dimensional trace vector T, where $T(i)$ is set to that value of j for which $C(i)$ is minimized. From this information we can easily reconstruct the optimal solution.

The obvious algorithm based on this approach can require time proportional to n^2. We will show how to reduce this to $O(n)$, by using some simple algebraic transformations and a carefully chosen data structure. First, we can rewrite (1) as

$$C(i) = -a_i + \min_{\substack{j \geq i \\ b_j - a_i \leq H}} \{b_j + C(j + 1)\}, \tag{2}$$

which leaves the minimization independent of i, except for defining the relevant j values. Let us define $C^*(i) = b_{i-1} + C(i)$. Then, from (2) we have

$$C^*(i) = b_{i-1} - a_i + \min_{\substack{j \geq i \\ b_j - a_i \leq H}} \{C^*(j + 1)\}, \tag{3}$$

and by defining $b_0 = 0$, we have $C^*(1) = C(1)$. We also extend the definition of C^* so that $C^*(n + 1) = b_n + C(n + 1) = b_n$. We will use (3) as the basis for our improved dynamic programming algorithm.

The $C^*(i)$'s will be computed in order of decreasing i. The key idea is to keep accessible only those $C^*(i)$ values that can be useful for subsequent minimizations and, at the same time, to make it easy to find the new minimum for each $C^*(i)$. At each stage (new value of i), we need to delete those previous $C^*(j)$ values that are too large to be of further use or that are "too far away" to be used because $b_j - a_i > H$. We do this by storing the so-far-computed C^* values in a *deque* (double-ended queue), a data structure that allows lookup, insertion, and deletion at both ends, but allows no direct access to interior elements. The elements stored in the deque are pairs $(C^*(i), i)$ with i monotonically increasing from left to right, although the i values need not be consecutive. The deque operations are (since there is only one deque, we omit explicit reference to it):

push-left (x, i), *push-right* (x, i)	to insert element (x, i) at designated end
pop-left ( ), *pop-right* ( )	to delete element at specified end
C-of-left ( ), *C-of-right* ( )	to return C^*-value on specified end without changing deque
i-of-left ( ), *i-of-right* ( )	to return i-value on specified end without changing deque.

These are easily implemented in standard ways.[2]

Now we can present the linear-time algorithm for computing $C(1) = C^*(1)$ and the trace vector, T. The input for the algorithm is the sequence $a_1, a_2, \cdots, a_n$ of interval left endpoints and the sequence $b_1, b_2, \cdots, b_n$ of interval right endpoints, both in increasing order.

Algorithm 2:

```
initialize deque := $\langle (b_n, n + 1) \rangle$; $b_0 := 0$;
for $i := n$ to 1 by $-1$ do
    while ($b_{i\text{-of-right}( )-1} - a_i > H$) do pop-right ( );
```

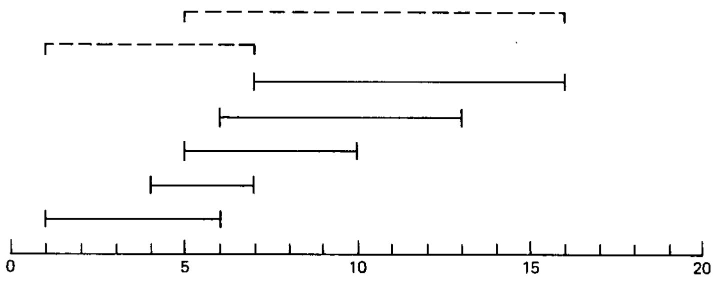

Fig. 3—Intervals (solid lines) and segments (dashed lines). There are two segments in the solution and their total length is 17.

```
$C^*(i) := b_{i-1} - a_i +$ C-of-right ( );
$T(i) :=$ i-of-right ( ) $- 1$;
while (C-of-left ( ) $\geq C^*(i)$) do pop-left ( );
push-left $(C^*(i), i)$;
```

The preceding discussion and the following easily verified observations directly imply the correctness of the algorithm and the fact that it runs in time $O(n)$.

Observation 1. Algorithm 2 maintains the deque in such a way that if (x, i) is to the left of (y, j), then $x > y$ and $i < j$.

Observation 2. For each i, $1 \leq i \leq n$, $(C^*(i), i)$ is inserted exactly once, and in each iteration of the loop the number of elements that are examined but not deleted is exactly two.

Theorem 1: Algorithm 2 correctly computes $C(1) = C^*(1)$ and the trace vector, T, in $O(n)$ steps.

Example. Consider the set of intervals [1, 6], [4, 7], [5, 10], [6, 13], [7, 16], with $H = 11$. Then the deque will assume the following values at the end of each iteration:

init:	(16, 6)	
$i = 5$:	(22, 5), (16, 6)	$T(5) = 5$
$i = 4$:	(20, 4), (16, 6)	$T(4) = 5$
$i = 3$:	(18, 3), (16, 6)	$T(3) = 5$
$i = 2$:	(20, 2), (18, 3)	$T(2) = 2$
$i = 1$:	(17, 1)	$T(1) = 2$.

The intervals and the optimal covering are shown in Fig. 3.

3.3 Linear cost function

In this case, gaps become significant and can no longer be treated separately. However, we shall see that we can restrict our attention to a limited set of potential segment starting points in the gaps. This will let us use a slightly more complicated, but still very efficient, dynamic programming approach.

Lemma 2: For any linear cost function, c, and any given problem instance, there exists an optimal solution of the form given by Lemma 1 such that, for any segment [x, y] in the solution:

(2.1) If $x + H$ lies in a gap, then $y = x + H$, i.e., the segment has length H.

(2.2) If $x + H$ does not lie in a gap, then y does not lie in a gap.

Proof: Consider any optimal solution of the form given in Lemma 1 and suppose also that it has a minimum number of segments among all such solutions. Then we know that no two adjacent or overlapping segments in the solution have lengths summing to H or less, since by the linear cost function they could then be combined into a single segment at no increase in cost.

Consider the leftmost segment $[x, y]$ that violates either (2.1) or (2.2). If no such segment exists, we are done. Otherwise, let $[z, w]$, with $x < z \leqslant y < x + H < w$, denote the next segment in the ordering. [The inequalities on x, y, z, and w follow from Lemma 1 (1.1) and our observation at the beginning of the proof]. If $x + H$ is in a gap but $y \neq x + H$, we can replace the two segments $[x, y]$ and $[z, w]$ by the two segments $[x, x + H]$ and $[x + H, w]$ without destroying the solution, since the new segments cover the same span, and any interval contained in one of the original two is contained in $[x, x + H]$ if it was to the left of $x + H$ and is contained in $[x + H, w]$ otherwise. If $x + H$ is not in a gap but y is in a gap, let u be the right endpoint of the gap containing y. Then we can replace $[x, y]$ and $[z, w]$ (where in fact $z = y$) with $[x, u]$ and $[u, w]$ without destroying the solution. In neither case have we increased the cost of the solution, because the sum of the lengths of the two new segments is no greater than the sum of the lengths of the original two. Moreover, the new segment starting at x retains the position of $[x, y]$ in the left-to-right ordering of segments in the solution and no longer violates (2.1) or (2.2). In addition, it is easy to verify that all segments starting to the left of x continue to satisfy (2.1) and (2.2) and no violations of the conditions of Lemma 1 have been introduced. Thus, we can repeatedly apply the appropriate one of these two transformations to the leftmost violator of (2.1) or (2.2), and we will eventually obtain an optimal solution of the stated form. □

We will restrict attention to optimal solutions of the form given by Lemma 2.

Consider what this tells us about solving any subproblem consisting of a range $[x, b]$ and all intervals contained in that range. If $x + H$ is not in a gap, then, by Lemma 2 (2.2) and Lemma 1 (1.2), we need only consider solutions in which the leftmost segment runs from x to the right endpoint of some interval or to the left end of some block. The remaining subproblem in each such case will have a range starting at the left endpoint of the leftmost uncovered interval from the original subproblem, as in the proportional case, so we do not need to know about any subproblems that start with points in gaps. If $x + H$ does lie in a gap, then, by Lemma 2 (2.1), we know that there is an optimal solution in which the leftmost segment has length exactly H, and we can, without loss of generality, choose to start with such a segment. However, to compute the cost of that solution, we need to know the optimal solution cost for the subproblem that then remains, and the range for that remaining subproblem begins at the point $x + H$, which does lie in a gap. This suggests that we will indeed need to solve certain subproblems whose ranges begin with points that lie in gaps. The key lies in finding a small number of such points that will suffice.

For any point x, define $fwd(x)$ to be the rightmost point $y \geqslant x$ such that y is congruent to x modulo H and such that all points in the sequence $x + H, x + 2H, \cdots, y$ lie in gaps. If $x + H$ does not lie in a gap, we let $fwd(x) = x$. Now, again consider the above situation of solving a subproblem with range $[x, b]$, where $x + H$ belongs to a gap. Then, by repeated application of Lemma 2 (2.1), we know that there is an optimal solution for this subproblem that begins with a sequence of adjacent segments of length exactly H ending at the points $x + H$, $x + 2H, \cdots, fwd(x)$, and we can, without loss of generality, choose to start this way. The remaining subproblem, whose solution cost we need for computing the optimal cost for the initial subproblem, still begins with a point in a gap (namely, the point $fwd(x)$), but it is a point of very special form. In particular, by the definition of $fwd(x)$, we have that $fwd(fwd(x)) = fwd(x)$.

It follows directly from the preceding discussion that we need only solve subproblems for ranges of the form $[x, b]$, where x is the left end of some interval, the right end of some block, or $fwd(z)$ for z one of those two types of points. We can describe the computation as follows: Let $\{z_1, z_2, \cdots, z_m\}$ be the set of all such starting points, sorted so that $a = z_1 < z_2 < \cdots < z_m = b$. For each z_i, let $b(z_i)$ be defined as b_j if $z_i = a_j$ for some interval $[a_j, b_j]$ (there can only be one such j by our assumption that no interval is contained in another) and as z_{i+1} otherwise. Notice that $b(z_1) < b(z_2) < \cdots < b(z_m)$ and that $b(z_i) \geqslant z_{i+1}$ for all i, $1 \leqslant i < m$. Let $C(z_i)$ denote the cost of an optimal solution for the subproblem with range $[z_i, b]$ consisting of all intervals contained in that range. Then, if $z_i \neq fwd(z_i)$, we have

$$C(z_i) = \frac{fwd(z_i) - z_i}{H}(\alpha H + \beta) + C(fwd(z_i)). \tag{4}$$

The solution in this case consists of a sequence of intervals of length H starting at the points $z_i, z_i + H, \cdots, fwd(z_i) - H$, followed by an optimal solution for the range $[fwd(z_i), b]$. If $z_i = fwd(z_i)$, we have

$$C(z_i) = \min_{\substack{j \geq i \\ b(z_j) - z_i \leq H}} \{\alpha(b(z_j) - z_i) + \beta + C(z_{j+1})\}. \tag{5}$$

The solution corresponding to a particular choice of j here consists of the segment $[z_i, b(z_j)]$ followed by an optimal solution for the range $[z_{j+1}, b]$. It is not difficult to see that the sequence $b(z_i) < b(z_{i+1}) < \cdots < b(z_k)$, where k is the greatest index such that $b(z_k) - z_i \leq H$, includes all possible right endpoints that need be considered for a segment starting at z_i, although it may also include some points in gaps that could have been ignored.

In addition, we can still use an algebraic transformation like that of the preceding section to simplify the computation. Accordingly, define $C^*(z_i) = \alpha b(z_{i-1}) + C(z_i)$. Then, rewriting (4), we have, if $z_i \neq fwd(z_i)$,

$$C^*(z_i) = \alpha(b(z_{i-1}) - fwd(z_i)) + \frac{fwd(z_i) - z_i}{H}(\alpha H + \beta) + C^*(fwd(z_i)). \tag{6}$$

Here we used the observation that if $fwd(z_i) = z_k$, $k > i$, then $b(z_{k-1}) = z_k$. Then, rewriting (5), we have, if $z_i = fwd(z_i)$,

$$C^*(z_i) = \alpha(b(z_{i-1}) - z_i) + \beta + \min_{\substack{j \geq i \\ b(z_j) - z_i \leq H}} \{C^*(z_{j+1})\}. \tag{7}$$

We also have $C^*(z_m) = b(z_{m-1})$, and we define $b(z_0) = 0$. These equations will provide the basis for our algorithm, which again uses the deque data structure of the previous subsection.

We assume that the algorithm is given as input the integer m, the sequence $z_0 = z_1, z_2, \cdots, z_m$, and, for $0 \leq i \leq m$, the corresponding values for $b(z_i)$ and $fwd(z_i)$. The algorithm computes $C(z_1) = C^*(z_1)$ and the trace vector T, as before. However, the value $T(i)$ of the trace vector T at z_i will be undefined whenever $z_i \neq fwd(z_i)$, since we already know that the solution for the subproblem starting at z_i begins with a sequence of length H segments from z_i to $fwd(z_i)$.

Algorithm 3:

```
initialize deque := $(b(z_{m-1}), m)$;
for $i := m - 1$ to 1 by $-1$ do
   while $(b(z_{i\text{-}of\text{-}right()-1}) - z_i > H)$ do pop-right ( );
   if $z_i \neq fwd(z_i)$
      then $C^*(z_i) := \alpha(b(z_{i-1}) - fwd(z_i))$
      $+ ((fwd(z_i) - z_i)/H)\ (\alpha H + \beta) + C^*(fwd(z_i))$
      else {
         $C^*(z_i) := \alpha(b(z_{i-1}) - z_i) + \beta +$ C-of-right ( );
         $T(i) :=$ i-of-right ( ) $- 1$};
```

```
while ($C$-of-left ( ) $\geq C^*(z_i)$) do pop-left ( );
push-left ($C^*(z_i)$, $i$);
```

The correctness of this algorithm and the fact that it runs in time $O(m) = O(n)$ follow from the preceding discussion and observations analogous to those made in the previous subsection.

Theorem 3: Algorithm 3 correctly computes $C(z_1) = C^*(z_1)$ and the trace vector T in $O(n)$ steps.

It remains for us to show how to compute the input for Algorithm 3. The computation of the $b(z_i)$ values is straightforward and can be accomplished easily in linear time. However, computing $fwd(z)$ for z, the left end of an interval or the right end of a block, from which the sequence $z_1, z_2, \cdots, z_m$ is determined, requires some care. It is easy to do this in linear time for each such z, simply by comparing z to each of the given intervals in left to right order, searching for the first interval to the right of z that contains a point congruent to z modulo H, and then setting $fwd(z)$ equal to that point minus H. Unfortunately, this would require $O(n^2)$ time overall, substantially worse than Algorithm 3 itself. We now describe a method for computing the values of $fwd(z)$ for all such z in time $O(n \log n)$, still worse than linear but comparable to the preprocessing time for originally sorting the intervals.

The basic idea of the method follows: Suppose the range $[a, b]$, and all the given intervals, are cut at all points that are exact multiples of H and the resulting pieces are arranged into "shelves", as shown in Fig. 4, so that the multiples of H increase as we go down the shelves. For any point x, consider a vertical cutline through the shelves that passes through x, i.e., a line at distance x mod H in from the left ends of the shelves. Then it is easy to see that $fwd(x)$ is the furthest point not in any interval that can be seen from x by looking downward along this cutline, where the presence of an interval along the cutline blocks the view of shelves below it. To prevent "looking" beyond the last shelf, we include a dummy interval $[b, b + H]$ at the end of the original range.

Let us define the H-value of a shelf, or any point on that shelf, to be the integer t such that tH is the left endpoint of the shelf, and let us call a shelf *empty* for a particular cutline if the point at which the cutline passes through the shelf does not belong to any interval on that shelf. Now suppose we start with a cutline through the left ends of the shelves and gradually move it to the right, always keeping track of the H-values for the shelves that are nonempty at the current cutline position. This set of H-values changes whenever the cutline encounters the left or right endpoint of some interval, and we maintain this information in a data structure that can be updated easily when-

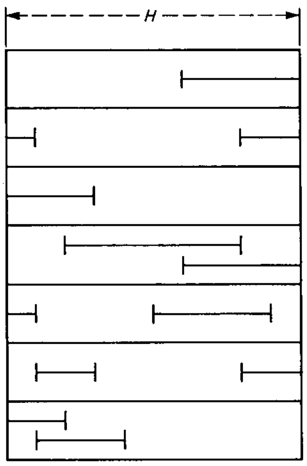

Fig. 4—Intervals arranged in shelves. Each shelf has width H.

ever such an endpoint is encountered. In addition, we choose our data structure so that it is easy to find, for any given integer t, the least H-value in the data structure that is larger than t. This is used for computing the value of $fwd(x)$ each time the moving cutline hits a point x that is the left endpoint of an interval or the right end of a block (it may hit more than one such point simultaneously). To compute $fwd(x)$ at this point, we simply find the least H-value q in the data structure that is larger than the H-value for x and set $fwd(x) = (q - 1)\,H + x \bmod H$. Thus, the values of $fwd(x)$ for all such points x will have been computed by the time the cutline has moved all the way to the right end of the shelves.

By using a balanced binary tree for storing the H-values, we can add or delete an individual H-value, or find the least H-value larger than a given integer, in time $O(\log n)$. Since at most $O(n)$ H-values will be added to or deleted from the data structure, regardless of the number of shelves, and since at most $O(n)$ requests for the least H-value larger than a given integer will occur, the entire procedure will thus require only $O(n \log n)$ time. The input $z_1, z_2, \cdots, z_m$ for Algorithm 3 is then obtained by combining the set of points x for which fwd was computed with the corresponding set of points $fwd(x)$ computed for them, and sorting the resulting collection of points (also in time $O(n \log n)$).

To clearly show how this scanning process can be implemented, we will describe it in more detail. The balanced tree data structure is quite standard and we will not go into detail here (see Ref. 3, for example). For our purposes, the entries in the tree can be regarded as pairs consisting of an H-value and an integer *multiplicity* for that H-value. The insertion of an H-value into the tree requires following the appropriate path from the root to find the node for that H-value, increasing its multiplicity by one if it is in the tree, and otherwise creating and inserting a new node for it with multiplicity 1. The deletion of an H-value from the tree requires following the appropriate path from the root to the node for that H-value, decreasing its multiplicity by one, and deleting the node if the multiplicity becomes zero. Finding in the tree the least H-value larger than a given integer t also involves a simple traversal from the root (the details depend on the exact implementation of the tree structure) and can be facilitated by storing in each interior node the smallest H-value occurring in its right subtree. Notice that the tree must be rebalanced only when a new node is inserted or deleted. Standard methods for implementing these operations ensure that each requires time at most logarithmic in the number of H-values in the tree and hence time $O(\log n)$.

The first step of the procedure is to cut the given intervals into shelves. It is convenient to begin by adding two dummy intervals, $[a - H, a]$ and $[b, b + H]$, at the beginning and end of the range, so that the points a and b do not have to be treated exceptionally. We then cut each interval that contains a point congruent to 0 modulo H in its interior (not as an endpoint) into two adjacent intervals at that point, i.e., replacing $[a_i, b_i]$ by $[a_i, tH]$ and $[tH, b_i]$, where $a_i < tH < b_i$ (there can be at most one such t). Next we combine all the endpoints of the resulting set of intervals into a single set (retaining repetitions), associating with each point the type of point it was in the interval it came from, with the choices being from among left end of a block (*lb*), right end of a block (*rb*), left end of an interval (but not of a block) (*li*), and right end of an interval (but not of a block) (*ri*). We also remember which points were introduced as dummies to split intervals. Finally, we replace each of these points x by the ordered pair $(\lfloor x/H \rfloor, x \bmod H)$, where $\lfloor w \rfloor$ denotes the largest integer not greater than w, and we sort them into nondecreasing order according to their second components. Notice that the first component for a point is its H-value (designating the shelf it is on), the second component is its position on that shelf, and the ordering corresponds to the sequence in which the points will be encountered as the cutline is moved from left to right.

We initialize the binary tree data structure to contain the H-values for all points with shelf position (second component) 0 and type either *rb* or *ri*. We then repeat the following steps until all points in the list have been processed (initially none have been processed):

1. Let δ be the shelf position (second component) for the next

unprocessed point on the list. (Notice that all points on the list with shelf position δ occur consecutively on the list.) Add to the tree the H-values of all points with shelf position δ and type either *lb* or *li*.

2. For each point x with shelf position δ and type *lb*, *li*, or *rb*, except for those that were added as dummies, compute $fwd(x)$ by finding in the tree the least H-value q larger than the H-value for x and setting $fwd(x) = (q - 1)\,H + \delta$.

3. Remove from the tree the H-values of all points with shelf position δ and type either *rb* or *ri*. All points with shelf position δ have now been processed.

It is easy to check that this procedure correctly computes $fwd(x)$ in each instance and that, except for the points $a - H$ and $b + H$, the points x for which $fwd(x)$ has been computed are exactly those that are left endpoints of intervals or right ends of blocks for the original set of intervals, as required. Thus, we simply need to delete those two inappropriate points, combine the remaining points x for which $fwd(x)$ was computed with the corresponding set of points obtained as values of $fwd(x)$ for them, and sort the resulting collection to form the sequence $z_1, z_2, \cdots, z_m$.

Combining Algorithm 3 with this method for computing the sequence $z_1, z_2, \cdots, z_m$, we then have Theorem 4.

Theorem 4: The proportional cost function case can be solved in time $O(n \log n)$.

3.4 Concave cost function

In this case we will again show that we can restrict attention to a limited set of potential segment starting points in gaps. The key lemma is the following:

Lemma 3: For any concave cost function c and any given problem instance, there exists an optimal solution of the form given by Lemma 1 such that, for any segment [x, y] in the solution:

(3.1) if $x + H$ lies in a gap, then $y = x + H$, i.e., the segment has length H, and

(3.2) if y lies in a gap and $y \neq x + H$, then for some integer $k \geq 1$ the points $y + H, y + 2H, \cdots, y + (k - 1)H$ all lie in gaps, the point $y + kH$ does not lie in a gap (and, hence, by Lemma 1, $y + kH$ is either the right endpoint of some interval or the left end of some block), and the segments $[y, y + H], [y + H, y \pm 2H], \cdots, [y + (k - 1)H, y + kH]$ all belong to the solution.

Proof: Consider any optimal solution of the form given in Lemma 1, and suppose also that it has a minimum number of segments among all such solutions. Then, as in the proof of Lemma 2, we know that no two adjacent or overlapping segments in the solution have lengths summing to H or less, since by the concave cost function they could then be combined into a single segment at no increase in cost.

We first deal with violations of (3.1). Consider the leftmost segment $[x, y]$ that violates (3.1). If no such segment exists, we are done with this portion of the proof. Otherwise, let $[z, w]$, with $x < z \leqslant y < x + H < w$, denote the next segment in the ordering. (The inequalities on x, y, z, and w follow from Lemma 1 (1.1) and our observation at the beginning of the proof.) If $x + H$ is in a gap but $y \neq x + H$, we can replace the two segments $[x, y]$ and $[z, w]$ by the two segments $[x, x + H]$ and $[x + H, w]$ without destroying the solution, since the new segments cover the same span, and any interval contained in one of the original two is contained in $[x, x + H]$ if it was to the left of $x + H$ and is contained in $[x + H, w]$ otherwise. Moreover, since the sum of the lengths of the two new segments is no greater than the sum of the lengths of the original two, and since the length of the longer of the two is now as large as possible, we cannot have increased the cost of the solution. It is easy to see that this transformation can cause no violation of (3.1) to the left of $[x, y]$ and that the properties of Lemma 1 are preserved, so repeated application to the leftmost violator of (3.1) will eventually remove all such violations.

So suppose we now have a solution satisfying Lemma 1 in which there are no violations of (3.1). Consider the leftmost segment $[x, y]$ that violates (3.2). If none exists, we have a solution of the form given by the lemma. Otherwise, since $[x, y]$ violates (3.2), $x + H$ is not in a gap and y is in a gap. Letting k be the least positive integer such that $y + kH$ does not lie in a gap, all the segments $[y, y + H]$, $[y + H, y + 2H]$, $\cdots$, $[y + (k - 2)H, y + (k - 1)H]$ belong to the solution, since (3.1) is not violated, but $[y + (k - 1)H, y + kH]$ does not belong to the solution. Let $[y + (k - 1)H, z]$ denote the segment in the solution that does start at $y + (k - 1)H$. We propose to shift the entire sequence of length H intervals starting at y to either the left or the right in a way that lengthens the longer of $[x, y]$ and $[y + (k - 1)H, z]$, correspondingly shortening the shorter of the two, until either one of the points $y + iH$ (for the new value of y), $0 \leqslant i < k$, no longer belongs to a gap or one of the two extreme segments achieves length H. By the concave cost function, this cannot increase the solution cost; it also introduces no violations to (3.2) to the left of $[x, y]$ and no violations to the conditions of Lemma 1. If the shifting terminates because one of the points $y + iH$, $0 \leqslant i < k$, ceases to belong to a gap, then the segment $[x, y]$ no longer violates (3.2). However, in this case it is possible that the new segment $[y + (k - 1)H, z]$ now violates (3.1), but we can then reapply the transformation described in the preceding paragraph until there are no such violations of (3.1) without affecting any intervals to the left of the point $y + (k - 1)H$. If the shifting terminates with all points $y + iH$, $0 \leqslant i < k$, still in gaps, then it must have terminated with the new segment $[y + (k - 1)H, z]$ having length H, since the point $x + H$ does not belong to a gap. In this case we have not introduced any violations to (3.1), and if z is not in a gap, $[x, y]$ no longer violates (3.2). On the other hand, if z is in a gap, we now have a new value of k for our new value of y and a longer sequence of length H segments starting at y. Hence, we can continue as above, and since the length of such a sequence of length H segments is finite, eventually we must obtain a solution in which $[x, y]$ no longer violates (3.2). Therefore, continued application of such transformations to repeatedly eliminate the leftmost segment violating (3.2) will eventually produce a solution satisfying the lemma. □

We will restrict attention to optimal solutions of the form given by Lemma 3.

Consider what this tells us about solving any subproblem consisting of a range $[x, b]$ and all intervals contained in that range. If $x + H$ lies in a gap, then, by Lemma 3, we know that there is an optimal solution that begins with a sequence of segments of length exactly H ending at the points $x + H, x + 2H, \cdots, fwd(x)$, where $fwd(x)$ is defined exactly as in the preceding subsection. If $x + H$ does not lie in a gap, then, by Lemma 1 (1.2), we need only consider solutions in which the leftmost segment runs from x to the right endpoint of some interval or the left end of some block, or from x to some point y in a gap. Furthermore, in the latter case, by Lemma 3 (3.2), we can restrict attention to points y such that, for some integer $k \geqslant 1$, $y + kH$ is the right endpoint of some interval or the left end of some block, and all the points $y + H$, $y + 2H, \cdots, y + (k - 1)H$ lie in gaps.

As in the preceding subsection, let $a = z_1 < z_2 < \cdots < z_m = b$ be the sorted collection of all interval left endpoints, block right endpoints, and points of the form $fwd(x)$ for x one of those two types of points; for $1 \leqslant i \leqslant m$, let $b(z_i)$ be defined in the same way as before. In addition, for $1 \leqslant i \leqslant m$, define $bkwd(z_i)$ to be $b(z_i)$ if $b(z_i)$ is in a gap or $b(z_i) - H$ is not in a gap; otherwise, define $bkwd(z_i)$ to be the leftmost point $y \leqslant b(z_i)$ congruent to $b(z_i)$ modulo H such that all points $b(z_i) - H, b(z_i) - 2H, \cdots, y$ lie in gaps. Notice that, for $b(z_i)$ not in a gap, $bkwd(z_i)$ is defined in the same way as $fwd(z_i)$ except that the jumps of length H are made to the left from $b(z_i)$ instead of to the right from z_i, i.e., we reverse our sense of direction and use the other end of the interval or block. Thus, $bkwd(z_i)$ for all i can be computed in time $O(n \log n)$ using a method analogous to that of the previous subsection for computing $fwd(z_i)$. Also as before, let $C(z_i)$ denote the cost of an optimal solution for the subproblem with range $[z_i, b]$ consisting of all intervals contained in that range.

Now, once again consider the consequences of Lemma 3. If

$z_i \neq fwd(z_i)$ (or, equivalently, $z_i + H$ lies in a gap), we have

$$C(z_i) = \frac{fwd(z_i) - z_i}{H} c(H) + C(fwd(z_i)). \tag{8}$$

The corresponding solution consists of a sequence of length H intervals ending at the points $z_i + H$, $z_i + 2H$, $\cdots$, $fwd(z_i)$, followed by an optimal solution for the range $[fwd(z_i), b]$. On the other hand, if $z_i = fwd(z_i)$, then, from the fact that every right endpoint of an interval or left end of a block in this subproblem is included among $\{b(z_j): j \geq i\}$, $C(z_i)$ will be the smaller of

$$\min_{\substack{j \geq i \\ b(z_j) \leq z_i + H}} \{c(b(z_j) - z_i) + C(z_{j+1})\} \tag{9}$$

and

$$\min_{\substack{j \geq i \\ b(z_j) > z_i + H \\ bkwd(z_j) \leq z_i + H}} \left\{ \left\lfloor \frac{b(z_j) - z_i}{H} \right\rfloor c(H) + c((b(z_j) - z_i) \bmod H) + C(z_{j+1}) \right\}. \tag{10}$$

Formula (9) covers all the possibilities in which the leftmost segment does not end in a gap (but includes some potential ending points in gaps that could have been ignored), and (10) covers all the possibilities in which the leftmost segment does end in a gap and is followed by a sequence of segments of length H that terminates at the right endpoint of some interval or left end of some block (the corresponding $b(z_j)$). The solution corresponding to a particular choice of j is, in (9), the segment $[z_i, b(z_j)]$ followed by an optimal solution for the subproblem with range $[z_{j+1}, b]$ and, in (10), is the segment $[z_i, z_i + (b(z_j) - z_i) \bmod H]$, followed by a sequence of length H intervals starting at the points $z_i + (b(z_j) - z_i) \bmod H$, $\cdots$, $b(z_j) - 2H$, $b(z_j) - H$, followed by an optimal solution for the subproblem with range $[z_{j+1}, b]$. Notice that the condition that $bkwd(z_j) \leq z_i + H$ will be satisfied if and only if the left endpoints of all those length H segments belong to gaps, as required. Since $bkwd(z_j) \leq b(z_j)$, we can combine (9) and (10) to obtain, for the case of $fwd(z_i) = z_i$,

$$C(z_i) = \min_{\substack{j \geq i \\ bkwd(z_j) \leq z_i + H}} \left\{ \left\lfloor \frac{b(z_j) - z_i}{H} \right\rfloor c(H) + c((b(z_j) - z_i) \bmod H) + C(z_{j+1}) \right\}. \tag{11}$$

Notice that the cost for the interval starting at z_i is accounted for in the first term of the minimization if that interval has length H and otherwise is accounted for in the second term of the minimization.

Equations (8) and (11) will serve as the basis for our algorithm. (The further simplifications used in the previous cases do not apply here, since they hold only for cost functions that are linear). We assume that the algorithm is given as input the integer m, the sequence $z_1, z_2, \cdots, z_m$, and, for $1 \leq i \leq m$, the corresponding values for $b(z_i)$, $fwd(z_i)$, and $bkwd(z_i)$. These can be computed using the methods of the previous subsection in time $O(n \log n)$. The algorithm computes $C(z_1)$ and the trace vector T. The value of $T(i)$ will be that value for j for which the cost expression for $C(z_i)$ is minimized and from which the segments in the optimal solution can be reconstructed. $T(i)$ will be undefined whenever $z_i \neq fwd(z_i)$, since then we already know that the solution for that subproblem begins with a sequence of adjacent length H segments running from z_i to $fwd(z_i)$.

Algorithm 4:

```
initialize $C(z_m) := 0$;
for $i := m - 1$ to 1 by $-1$ do
   if $z_i \neq fwd(z_i)$
      then $C(z_i) := ((fwd(z_i) - z_i)/H)c(H) + C(fwd(z_i))$
      else {
         initialize $C(z_i) := \infty$; $T(i) := \infty$;
         for $j := m - 1$ to $i$ by $-1$ do
            if $bkwd(z_j) \leq z_i + H$
               then {
                  $hmult := \left\lfloor \frac{b(z_j) - z_i}{H} \right\rfloor$;
                  $resid := (b(z_j) - z_i) \bmod H$;
                  $newC := hmult \cdot c(H) + c(resid) + C(z_{j+1})$;
                  if $newC < C(z_i)$
                     then {$C(z_i) := newC$; $T(i) := j$};
                  };
      };
```

It is straightforward to verify that the algorithm runs in time $O(m^2) = O(n^2)$. The correctness of the algorithm follows from the preceding discussion and Lemma 3.

Theorem 5: Algorithm 4 correctly computes $C(z_1)$ and the trace vector T in $O(n^2)$ steps.

IV. OPEN PROBLEMS

The obvious open problems are to ask for improvements on and simplifications of the algorithms derived in this paper. Given the apparent need for sorting, it appears unlikely that the speed of our first three algorithms can be improved in any major (i.e., asymptotic) way. However, the algorithm for the concave cost function case offers some room for improvement here, even though it intuitively seems that the arbitrary nature of such functions can force one to evaluate the cost function for $O(n^2)$ different segment lengths in general. Further simplification of our algorithms, on the other hand, seems feasible and certainly would be useful.

There is also a generalization of our problem that is of great potential interest. Suppose that the problem definition is augmented to include an additional integer K as input, with the additional restriction on solutions that no more than K segments can ever overlap at a single point. This overlap constraint arises in situations where extensive multiple exposures are likely to arise and cannot be tolerated anywhere on the layout. It is easy to see that all of our normalization lemmas continue to hold under such a constraint, i.e., the transformations done in their proofs do not increase the maximum number of overlapping segments in the solution. However, it is no longer the case that in an optimal solution the solutions to subproblems (of the types considered in this paper) need be optimal solutions for those subproblems, since they may need to satisfy certain derived (and possibly complicated) additional constraints on the maximum overlap in various subregions. At present we do not see an appropriate, more restrictive, definition of subproblem that is both sufficient to allow these problems to be solved via dynamic programming and, at the same time, leads to a small enough class of subproblems for any given problem instance that all can be solved in a reasonable amount of time. We would be very interested in either efficient algorithms for problems of this sort or convincing demonstrations that these problems are inherently intractable.

V. ACKNOWLEDGMENT

We would like to thank Dave Alles for suggesting the problem and for helpful discussions.

REFERENCES

1. D. R. Herriott et al., "EBES: A Practical Electron Lithographic System," IEEE Trans. Electron Dev., *ED-22*, No. 7 (July 1975), pp. 385–92.
2. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Reading, MA: Addison-Wesley, 1968, Chap. 2.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Reading, MA: Addison-Wesley, 1983, Chap. 5.

AUTHORS

Michael R. Garey, B.S. (Mathematics), 1967, M.S. (Computer Science),

1969, and Ph.D. (Computer Science), 1970, University of Wisconsin; AT&T Bell Laboratories, 1970—. Mr. Garey has been Head of the Mathematical Foundations of Computing Department since 1981. He has done research in various areas of mathematics and computer science, including combinatorics, graph theory, design and analysis of algorithms, and computational complexity. He was awarded the 1979 Lanchester Prize of the Operations Research Society of America, and from 1979 through 1982 he served as editor-in-chief of the Journal of the Association for Computing Machinery. Member, ACM, SIAM, ORSA.

Ron Y. Pinter, B.S. (Computer Science), 1975, Technion—Israel Institute of Technology; S.M. and Ph.D. (Computer Science), The Massachusetts Institute of Technology, 1980 and 1982, respectively; AT&T Bell Laboratories, 1982–1983; IBM Israel Scientific Center, 1983—. Mr. Pinter was a member of the Principles of Computing Research Department, where he had been studying layout algorithms for integrated circuits, computational geometry, and the design of programming languages. Recently, he returned to Israel after spending 5 years in the United States as a Fulbright-Hayes grantee. Member, ACM, IEEE Computer Society.