Covering Rectilinear Polygons by Rectangles

SAN-YUAN WU AND SARTAJ SAHNI, FELLOW, IEEE

Abstract—**Three approximation algorithms to cover a rectilinear polygon that is neither horizontally nor vertically convex by rectangles are developed. All three guarantee covers that have at most twice as many rectangles as in an optimal cover. One takes $O(n \log n)$ time where n is the number of vertices in the rectilinear polygon. The other two take $O(n^2)$ and $O(n^4)$ time, respectively. Experimental results indicate that the algorithms with complexity $O(n^2)$ and $O(n^4)$ often obtain optimal or near optimal covers.**

Keywords and Phrases—**Rectilinear polygon, polygon covers, approximation algorithm, computational geometry.**

I. Introduction

RECTILINEAR polygons arise frequently in VLSI layout and artwork analysis, computer graphics, databases, image processing, etc. [3], [7], [10], [15], [18]. Often, the functions to be performed on a rectilinear polygon or on a set of rectilinear polygons are more easily performed by considering the rectilinear polygon(s) as being composed of several rectangles. A set T of rectangles is said to *cover* the polygon P iff P is the union of the rectangles of T. A cover T is a *partitioning* of P iff the rectangles in T are disjoint. Performing design rules checks and painting and drawing polygons on screens are two examples of functions that are easier to perform using rectangle covers rather than the original polygons.

A *minimal overlapping cover* (MOC) of a rectilinear polygon P is a cover of P that has the fewest number of rectangles. A *minimal nonoverlapping cover* (MNC) of a rectilinear polygon P is a partitioning of P with the smallest number of rectangles. Fig. 1 shows a rectilinear polygon P for which $|\text{MNC}(P)| = 3$ (Fig. 1(a)) and $|\text{MOC}(P)| = 2$ (Fig. 1(b)). In [12], it is shown that finding an MOC of a rectilinear polygon with holes (Fig. 2) is *NP*-complete. An exponential time algorithm to find an MOC is developed in [10]. Chaiken *et al.* [2] established the existence of a polynomial time algorithm to find the MOC of rectilinear polygons that are convex in both the x and y directions. This convexity restriction implies that the polygons are hole free. Franzblau *et al.* [5] have developed an $O(n^2)$ algorithm to find an MOC of rectilinear polygons that are convex in either the x or the y direction (n is the number of vertices in P). This restriction also implies a hole free polygon. For the case of hole free rectilinear polygons that are not convex in either the x or the y direction, the MOC problem is *NP*-hard [4]. The MNC of a rectilinear polygon can, however, be obtained in $O(n \log \log n)$ time [11]. This algorithm is not very practical as it employs finger search trees. The $O(n^{3/2} \log n)$ algorithm of [9] remains the best practical algorithm to find the MNC.

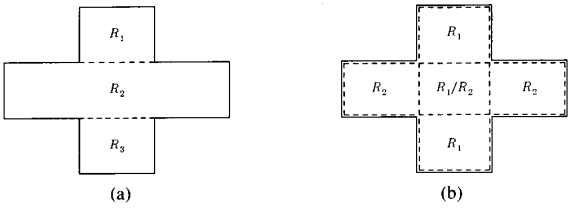


Fig. 1. Rectangular covers of a rectilinear polygon. (a) (a) Nonoverlapping cover (size = 3). (b) Overlapping cover (size = 2).

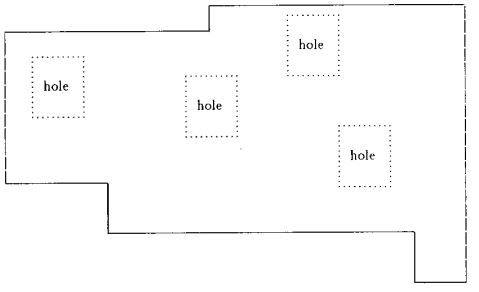


Fig. 2. Rectilinear polygons with holes.

In this paper, we are concerned solely with the problem of finding an MOC of a hole free rectilinear polygon. First, in Section II, we show that $|\text{MNC}| \leq 2|\text{MOC}| - 1$ for every hole free rectilinear polygon. Hence, the known polynomial time MNC algorithms can be used to obtain covers that are within a factor of 2 of optimal. In Section III, we develop three greedy approximation algorithms that find covers whose sizes are also $\leq 2|\text{MOC}| - 1$. Experimental data provided in Section IV indicate that these algorithms often do better than an MNC. In fact, they often find an optimal or near optimal cover. Since the MOC of horizontally convex and vertically convex rectilinear polygons can be found in $O(n^2)$ time [5], our approximation algorithms are recommended when the polygons are convex in neither the x nor the y directions.

II. $|\text{MNC}| \leq 2|\text{MOC}| - 1$

In this section, we show that for every hole free rectilinear polygon P, the size of the MNC is at most two

Manuscript received September 6, 1988; revised June 29, 1989. This work was supported in part by the National Science Foundation under Grant MCS-83-05567. This paper was recommended by Associate Editor R. H. J. M. Otten.

The authors are with the Computer Science Department, University of Minnesota, Minneapolis, MN 55455.

IEEE Log Number 8933512.

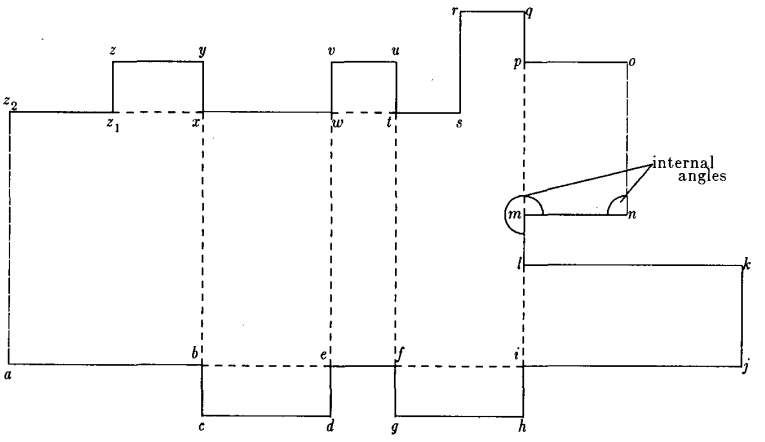


Fig. 3. Vertices = $\{a, b, \cdots, z_2\}$; edges = $\{ab, bc, cd, ef, gh, \cdots, z_2a\}$; |vertices| = |edges| = 28 (even); convex vertices = $\{a, c, d, g, h, j, k, n, o, q, r, u, v, y, z, z_2\}$; concave vertices = vertices − convex vertices = $\{b, e, f, i, l, m, p, s, t, w, x, z_1\}$; extended horizontal edges = $\{abefij, cd, gh, kl, mn, op, qr, stwxz_1z_2, uv, yz\}$; extended vertical edges = $\{z_2a, zz_1, yxbc, vwed, utfg, rs, qpmlih, on, kj\}$.

times that of the MOC minus one. Furthermore, this is a tight bound as there exist hole free rectilinear polygons P for which $|\text{MNC}| = 2|\text{MOC}| - 1$. This result tells us that by using an MOC versus an MNC we can reduce the number of rectangles in the cover by a factor of at most two.

First, we introduce some terminologies. A hole free rectilinear polygon may be described by its boundary vertices and edges. The number of vertices is always even and equal to the number of edges. Two vertices (x_1, y_1) and (x_2, y_2) are *cohorizontal (covertical) iff* $y_1 = y_2$ ($x_1 = x_2$). A vertex is convex (concave) iff the interior angle made by the two edges incident on this vertex is 90° (270°). A *chord* is a line that joins two cohorizontal or two convertical concave vertices without passing through any polygon edge. An *extended edge* is a longest continuous horizontal or vertical segment comprised of edges and chords. Fig. 3 gives an example. Two chords are *independent* iff they do not intersect. In the example of Fig. 3, the chords z_1x, wt, pm, fi, and be are independent. There are several other sets of independent chords.

Theorem 1: (a) For every hole free rectilinear polygon P, $|\text{MNC}| \leq 2|\text{MOC}| - 1$. (b) There exist hole free rectilinear polygons for which $|\text{MNC}| = 2|\text{MOC}| - 1$.

Proof: (a) Let P be an arbitrary hole free rectilinear polygon. Let n be the number of vertices (and, hence, edges) in P. Let δ be the number of chords. In [17] and [15], it is shown that $|\text{MNC}| = n/2 - \gamma - 1$ where γ is the size of the maximum independent chord set. Since all horizontal chords form an independent chord set, all vertical chords form an independent chord set, and all chords are either horizontal or vertical, it follows that $\gamma \geq \delta/2$. So, $|\text{MNC}| \leq (n - \delta)/2 - 1$.

Let R be any rectangle in any overlapping cover of P. The edge segments of P (if any) that lie on any one edge of R must all be from the same extended edge of P. Hence, the edges of R can include edge segments of P from at most four different extended edges. Since every overlapping cover of P must include all of the edges of P, it follows that $|\text{MOC}| \geq (n - \delta)/4 \geq (1/2)[(n - \delta)/2 - 1] + (1/2) \geq (1/2)|\text{MNC}| + (1/2)$. Hence, $|\text{MNC}| \leq 2|\text{MOC}| - 1$.

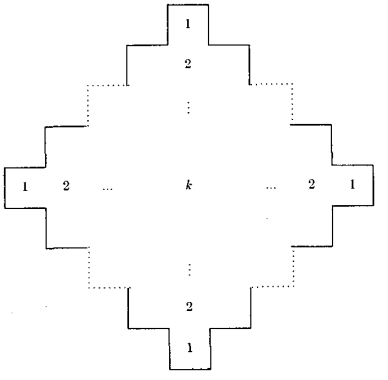


Fig. 4. $|\text{MNC}| = 2k - 1$, $|\text{MOC}| = k$.

(b) Fig. 4 shows a hole free rectilinear polygon for which $|\text{MOC}| = k$ and $|\text{MNC}| = 2k - 1$. □

III. Approximation Algorithms

3.1. Preliminaries

A *horizontal (vertical) cut* of a hole free rectilinear polygon is a horizontal (vertical) line that begins at a concave vertex, passes through the polygon interior and ends at the first point at which it meets a polygon edge. Fig. 5(a) shows an example polygon with all its horizontal and vertical cuts drawn as broken lines. Note also that the cuts partition the polygon into rectangular regions called *cells*. The *cells* of Fig. 5(a) have been labeled $C_1, C_2, \cdots, C_{11}$.

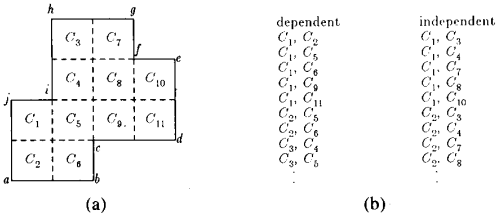


Fig. 5. Relationship between two cells. (a) Cells in P. (b) Pairs.

The cuts also divide the edges of the polygon into segments called *edge segments*. A *boundary cell* is a cell that has at least one side that is an edge (or edge segment) of the polygon. The boundary cells of Fig. 5(a) are C_1, C_2, C_3, C_4, C_6, C_7, C_9, C_{10}, and C_{11}. A cell that is not a boundary cell is called an *interior cell*. C_5 and C_8 are the only interior cells of Fig. 5(a).

Theorem 2: Let P be a hole free rectilinear polygon with n vertices.

i) The number β of concave vertices is $n/2 - 2$.
ii) The number of cells is at most $(n/2 - 1)^2$.
iii) The number of edge segments is at most $2n - 4$.
iv) The number of boundary cells is at most $3n/2 - 5$.

Proof: (i) May be shown by induction on n. ii) Follows from the observation that the number of horizontal and vertical cuts is at most β each. Hence, the number of cells cannot exceed $(\beta + 1)^2 = (n/2 - 1)^2$. For iii), we note that before any cuts are drawn, there are exactly n edge segments (each segment being a full edge). Each cut increases the number of segments by at most 1 (as it begins at a concave vertex and ends in an edge or at a vertex). Since the total number of cuts is at most $2\beta = n - 4$, the total number of edge segments can be at most $n + n - 4 = 2n - 4$. For iv), observe that for each boundary cell, at least one of the four lines that define its boundary is an edge segment. Further, each edge segment is on the boundary of exactly one cell. Hence, the number of boundary cells is at most the number of edge segments. Since there are $n/2 + 2$ convex corners and at each such corner, two edge segments bound the same cell, the number of boundary cells cannot exceed $2n - 4 - (n/2 + 2) + 1 = 3n/2 - 5$. □

R is said to be a rectangle of the rectilinear polygon P (abbreviated rectangle of P) iff R is a rectangle and R is composed solely of cells in P. R is a *maximal* rectangle of P iff R is a rectangle of P and R is contained in no other rectangle of P. Note that each cell of P is a rectangle of P. For the example of Fig. 5(a), $\{C_2, C_6\}$ and $\{C_1, C_5\}$ are two of its many rectangles. Neither is maximal as both are contained in the rectangle $\{C_1, C_2, C_5, C_6\}$. The maximal rectangles of the example polygon are $\{C_1, C_2, C_5, C_6\}$, $\{C_3, C_4, C_5, C_6\}$, $\{C_1, C_5, C_9, C_{11}\}$, $\{C_3, C_4, C_5, C_7, C_8, C_9\}$, and $\{C_4, C_5, C_8, C_9, C_{10}, C_{11}\}$. It is easy to see that every MOC can be extended to an MOC that consists solely of maximal rectangles.

Two cells of P are *dependent* iff there is a rectangle R of P that contains both the cells. Two cells are *independent* iff they are not dependent. Fig. 5(b) lists the dependent and independent cell pairs of the polygon of Fig. 5(a).

Theorem 3: Let P be a hole free rectilinear polygon, S be a set of pairwise independent cells of P, and Y be an MOC of P. $|Y| \geq |S|$.

Proof: Since the cells in S are pairwise independent, no two of them can be covered by the same rectangle in Y. This together with the observation that every cell of P (and hence, of S) must be covered by at least one rectangle of Y implies that $|Y| \geq |S|$. □

$S = \{C_1, C_3, C_{10}\}$ is a set of independent cells of the polygon of Fig. 5(a). So, any MOC of this polygon has at least 3 rectangles in it. It is easily seen that the MOC actually has exactly 3 rectangles. Fig. 6 shows a polygon P for which the maximum set of independent cells has size 7. However, the MOC has 8 rectangles.

Theorem 3 suggests the following approach to obtain an overlapping cover of a polygon P.

Step 1: Find a maximal independent cell set of P.

Step 2: Expand each cell in this set into a maximal rectangle of P.

Step 3: Cover the cells of P not covered by step 2 with maximal rectangles of P chosen in some greedy manner.

Since we know of no algorithm to do step 1 in polynomial time, we abandon this approach.

A set $T = \{R_1, R_2, \cdots\}$ of rectangles of P *covers* a cell C_i iff C_i is in at least one of the rectangles in T. *Covered*(T) denotes the set of cells covered by T. *Uncovered*(T) denotes the remaining cells of P. Let the *neighborhood* of cell X, *neb*(X), be the set of cells $\{Y \mid X = Y$ *or* X *and* Y *are dependent*$\}$. A cell X is said to be *nonchoice* with respect to a set T, of rectangles of P iff:

a) $X \in$ *uncovered*(T);
b) *neb*$(X) \cap$ *uncovered*(T) is contained in some maximal rectangle of P.

A cell that is not nonchoice is *choice*.

For the example polygon of Fig. 5(a), *neb*$(C_1) = \{C_1, C_2, C_5, C_6, C_9, C_{11}\}$. C_1 is not a nonchoice cell (i.e., it is a choice cell) with respect to $T = \varnothing$ as no single maximal rectangle of P covers all of these cells. However, if $T = \{\{C_4, C_5, C_8, C_9, C_{10}, C_{11}\}\}$, then *neb*$(C_1) \cap$ *uncovered*$(T) = \{C_1, C_2, C_6\}$ which is contained in the maximal rectangle $\{C_1, C_2, C_5, C_6\}$. So, C_1 is a nonchoice cell with respect to this T.

The notion of a nonchoice cell suggests the greedy heuristic of Fig. 7. Interestingly, one can prove that the cover generated by this heuristic is optimal if the algorithm never enters the **else** clause of the **if** statement. This follows from Theorem 4.

Theorem 4: Let P be a hole free rectilinear polygon. Let Y be an MOC of P and let Z be the cover generated by procedure Greedy (Fig. 7). Let *nonchoice#* be the

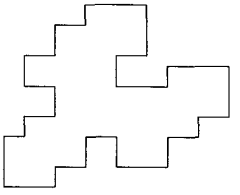


Fig. 6. |Maximum independent cell set| = 7, but |MOC| = 8.

```
Procedure Greedy;
begin
    Cover := ∅;
    while uncovered(Cover) ≠ ∅ do
        if there is a nonchoice cell X then
            add the rectangle that contains neb(X) ∩ uncovered(Cover) to Cover
        else
            pick any uncovered cell X and add to Cover any maximal rectangle that in-
            cludes X;
end; {of Greedy}
```

Fig. 7. Greedy heuristic.

number of rectangles in Z introduced by the **then** clause of the algorithm. $|Y| \geq Nonchoice\#$.

Proof: The number of nonchoice cells found by procedure Greedy is also $Nonchoice\#$. All of these cells must be independent as each is covered by a rectangle that includes $neb(X) \cap uncovered(Cover)$. Hence, there are at least $Nonchoice\#$ cells in a maximum independent cell set of P. From Theorem 3, it follows that $|Y| \geq Nonchoice\#$. □

Note 1: If $|Z| = Nonchoice\#$, then Z is an MOC.

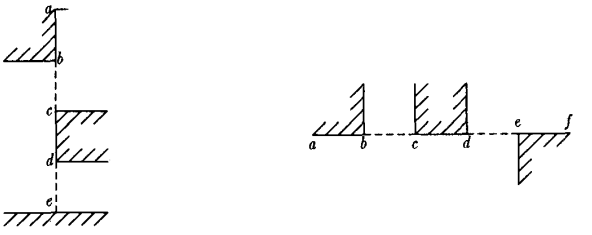
Note 2: If the **else** clause is entered p times, then Z has at most p rectangles more than in an MOC.

Note 2 above may be used to obtain an upper bound on the size difference between an MOC and a cover obtained by any heuristic that is structured as in Fig. 7. Two of the approximation algorithms we develop later are refinements of Fig. 7.

Another lower bound on |MOC| is obtained by classifying the extended edges of a hole free polygon into the categories *vertical* and *horizontal* (Fig. 8). If we delete the right boundary edges from a vertical extended edge, we get *left extended segments* (LES). For example, deletion of the right boundary edges from the left extended edge *abcde* of Fig. 8(a) gives the LES's *abc* and *de*. A *left extended segment with edges* (LESWE) is an LES that contains at least one left boundary edge. So, the LES *abc* is an LESWE while the LES *de* is not. *Right extended segments* (RES), *top extended segments (TES), bottom extended segments* (BES), *RESWE*, *TESWE*, and *BESWE* are similarly defined.

Theorem 5: For every hole free rectilinear polygon P, |MOC| ≥ max {#LESWE, #RESWE, #TESWE, #BESWE}.

Proof: Consider the set L of LESWE. Each LESWE, l_i, of L has at least one left boundary edge, le_i, on it. Let x_i be any cell adjacent to le_i, $1 \leq i \leq$ #LESWE. It is easy to see that $\{x_i \mid 1 \leq i \leq \#LESWE\}$ is an independent



(a) (b)

Fig. 8. (a) A vertical extended edge. (b) A horizontal extended edge. Shaded area is polygon exterior.

cell set. Hence, |MOC| ≥ #LESWE. Similarly, we may show that |MOC| ≥ $\#X$, $X \in$ {RESWE, TESWE, BESWE}. Hence, |MOC| ≥ max {#LESWE, #RESWE, #TESWE, #BESWE}. □

3.2. Approximation Algorithm 1 (Approx1)

This algorithm, Approx1 (Fig. 9), is based on Theorem 5. It uses the concept of a *support edge*. Such an edge has the property that both of the vertices at its ends are convex. In Fig. 5(a), edges *ab*, *de*, *gh*, and *ja* are support edges. The remaining edges are not. Fig. 10(a) shows an example polygon. It has two left support edges, *ab*, and *uv*. *abgh* is the maximal rectangle one of whose sides is *ab* and *uvjt'* is the maximal rectangle one of whose sides is *uv*. If the support edge *uv* is picked the rectangle *uvjt'* is added to the cover T being constructed. Next, the polygon is contracted. This contraction is done in such a way that:

i) all support edges of the contracted polygon have at least one adjacent uncovered cell;

ii) the contracted polygon requires the same number of rectangles to cover its uncovered cells as does the original partially covered polygon.

The resulting contracted polygon is shown in Fig. 10(b). The shaded area represents the area that is already covered. The polygon of Fig. 10(b) has the single left support edge *ab* and the rectangle *abgh* is added to the cover T. Following the contraction we got the polygon of Fig. 10(c). *sd* is its only left support edge. The maximal rectangle, one of whose sides is *sd* is *sder'*. This is added to T and the polygon contracted. Fig. 10(d) shows the results. Next, *qi'iq'* is added to T and the polygon contracted to get Fig. 10(e). The rectangles *nopq'* and *jklm* are added to T in the next two iterations.

Since every rectilinear polygon (or component) has at least one left support edge, Approx1 guarantees to find a cover. It is less evident that it guarantees to find a cover T such that $|T| < 2\,|MOC|$.

Theorem 6: Let P be any hole free rectilinear polygon. #LES < 2 |MOC|.

Proof: We shall show that #LES < #LESWE + #RESWE. The theorem then follows from Theorem 5. Consider a variant of Approx1 in which the **while** loop is replaced by the following:

```
Procedure Approx1 (P, T);
{Find an overlapping rectangle cover T of polygon P}
begin
    T := ∅;
    while P is not null do
    begin
        Let e be a left support edge;
        expand e to a maximal rectangle R one of whose sides is e;
        T := T ∪ {R};
        contract P;
    end;
end; {of Approx1}
```

Fig. 9. Approx1.

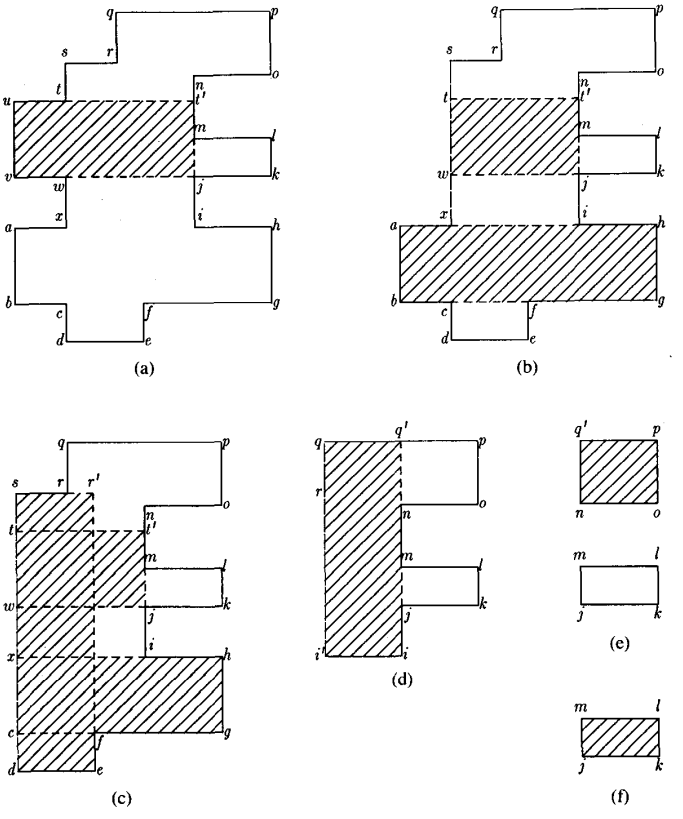


Fig. 10. Example for Approx1.

while P is not null **do**
begin
    let e be a left support edge;
    move e right to the next vertical extended edge discarding the column of cells passed over;
end;

The number of iterations of the **while** loop equals the number of left support edges encountered. This is equal to #LES. To count the number of left support edges (note that movement of e may create new support edges), we shall balance each left support edge against an extended edge of the original polygon. At times, this balancing will be postponed to a later iteration of the **while** loop. Let U denote the number of left support edges not yet accounted for and let C denote the number of components in the polygon we are currently dealing with. Initially, $C = 1$ and $U = 0$. On termination of the **while** loop, $C = 0$ and $U = -1$.

The movement of the left support edge e performed at the end of each iteration of the **while** loop affects C and U in one of the following ways.

a) C is unchanged. In this case, when e is moved to the next vertical extended segment with edges, a new left support edge may or may not be created (Fig. 11(a) and (b)). If it is, it is balanced against the extended segment with edge to which e is moved.

b) C is decreased by 1. The component of P that contained e must have been a rectangle. Its right boundary is a RESWE. This RESWE is used to decrease U by 1.

c) C increases by k, $k \geq 1$. In this case the extended edge to which e is moved is a RESWE (Fig. 11(c)). k +

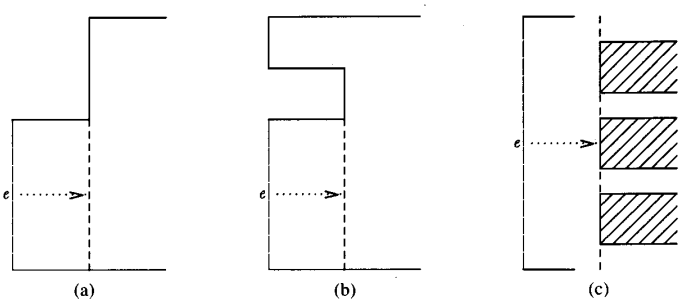


Fig. 11. (a) New left support edge. (b) No left support edge. (c)

1 new left support edges are created. One of these is balanced against the RESWE to which e is moved. The remaining k edges result in increasing U by k.

It is easy to see that no LESWE or RESWE can be balanced against more than once and that LESWE's that are originally left support edges are not balanced against in a), b), or c). Further, since the increase in U in (c) equals the increase in C and since C is zero on termination, U must decrease to -1 by termination. So all newly created left support edges not balanced in c) must get balanced in b). Consequently, the total number of left support edges (both original and created) is less than #LESWE + #RESWE. So, #LES $<$ #LESWE + #RESWE $\leq 2\,|\,\mathrm{MOC}\,|$. □

Theorem 7: Let P be any hole free rectilinear polygon. Let T be the cover generated by Approx1. $|\,T\,| < 2\,|\,\mathrm{MOC}\,|$.

Proof: $|\,T\,|$ equals the number of left support edges. Only an LES of P can become a left support edge. So, the number of left support edges encountered by Approx1 is $\leq$#LES $< 2\,|\,\mathrm{MOC}\,|$ (Theorem 6). Note that the contraction of Approx1 can cause e to move beyond some LES and so it is possible for the generated cover to contain fewer rectangles than #LES. □

3.3 Implementation and Complexity

Our implementation of Approx1 relies on the fact that each of the maximal rectangles in the constructed cover contains a segment of one or more right edges of the original polygon. The implementation of Approx1 has three phases. In phase 1 the vertices of the polygon are input and the vertical edges that are on its boundary are constructed. In addition right boundary edges that are incident to a horizontal support edge are identified. These tasks are easily performed in $O(n)$ time by traversing the vertices in the input counterclockwise order. In phase 2 the vertical edges are decomposed into segments and pairs of segments that are visible to each other are identified. This is done in $O(n \log n)$ time by using a vertical sweep line to scan the vertical edges from left to right. The active left segments are maintained in a balanced binary search tree. In phase 3 left support edges are identified; their maximal rectangles constructed; and the polygon contracted. This is also done by using a left to right vertical sweep line. At each position of the sweep line we have a list of active left support edges and a list of disjoint vertical intervals (DVI). For each vertical interval in DVI we maintain a list of all right edge segments that are visible from it. DVI is used to decide when and how to form a left support edge.

We illustrate the implementation by an example. Consider the polygon of Fig. 12(a). In phase 1 the vertical edges *ab*, *cd*, *ef*, *gh*, *ij*, *kl*, *mn*, *op*, *qr*, *st*, *uv*, and *wx* are constructed and the right edges *ef* and *op* are identified as being incident to horizontal support edges. In phase 2 the vertical edge *mn* is split into the segments *nt'* and *t'm*. The edges *op*, *st*, and *uv* are also split. Also, the visible segment pairs (*cd*, *ef*), (*ab*, *gh*), (*wx*, *ij*), (*vm'*, *kl*) (*m'u*, *mt'*), (*tn'*, *t'n*), (*n's*, *oo'*), and (*qr*, *o'p*) are formed.

For phase 3 the vertical sweep begins at edges *ab* and *uv*. Since both edges are left support edges, they are added to LSE. The vertical intervals at this scan line position are also *ab* and *uv*. *ab* together with its visible right segment *gh* and *uv* together its visible right segments *kl* and *t'm* are added to DVI. The maximal rectangles *abgh* and *uvjt'* are added to T. The scan line then moves to the vertical edges *cd*, *wx*, and *st*. Processing *cd* first, the vertical interval *ab*, in DVI, is extended downward to get the interval *ad* and the visible right segment *ef* added to its list of visible segments. Processing *wx* next, intervals *ad* and *uv* are combined to get the single interval *du* and the right segment *ij* is added to the visibility list. Next, *st* is processed and the single vertical interval *du* of DVI becomes *ds* and the right segments *oo'* and *t'n* are added to the visibility list. At this time the left support edge *ds* is formed. The maximal rectangle *sder'* is formed by determining the minimum x-coordinate of the visibility segments of *ds*. This rectangle is added to T. Whenever a rectangle is added to T we attempt to shrink the vertical span of the polygon. The example polygon (Fig. 12(b)) is shrunk by first removing the right visible segment *ef* from the vertical interval *ds* in DVI. Next *gh* is removed and *ds* is changed to *xs*. The scan line is then moved to *rq* and the left support edge *i'q* formed (Fig. 12(c)). The interval *xs* becomes *i'q* and *o'p* is added to its list of right segments. The maximal rectangle *qi'iq'* is added to T and the vertical span of the polygon reduced by removing *ij* from the visibility list of *i'q*. The interval *i'q* is split to

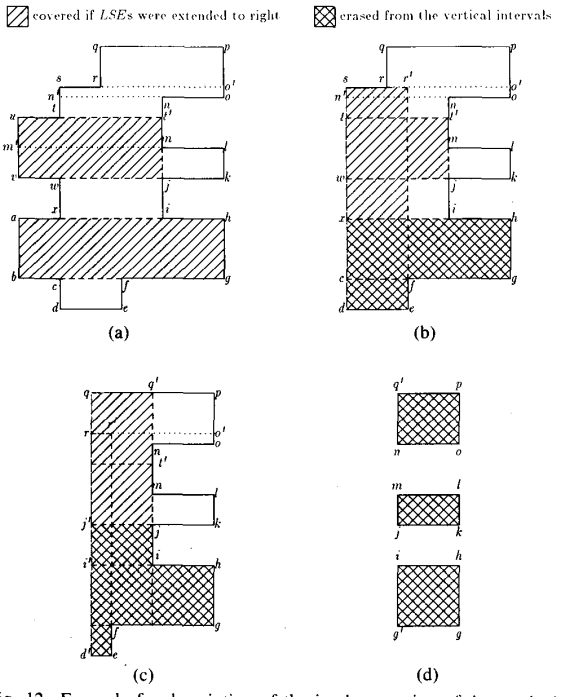


Fig. 12. Example for description of the implementation of Approx1. (a) LSE = {*ba*, *vu*}, DVI = {*ba*, *vu*}, T = {*abgh*, *uvjt'*}. (b) LSE = {*ba*, *vu*, *ds*}, DVI = {*xs*}, $T := T + \{sder'\}$. (c) LSE = {*ba*, *vu*, *ds*, *i'q*}, DVI = {*j'q*}, $T := T + \{qi'iq'\}$. (d) LSE = {*ba*, *jm*, *nq'*}, DVI = $\varnothing$, $T := T + \{nopq', jklm\}$.

get the intervals *nq'* and *jm*. The scan line then moves to the position of these two intervals and the maximum rectangles *mjkl* and *q'nop* are added to T. Following the vertical shrinking at this position, DVI becomes empty and the algorithm terminates.

By using 2–3 trees and either interval trees [14] or segment trees [1] the above strategy can be implemented to have a run time $O(n \log n)$.

3.4. Approximation Algorithm 2 (Approx 2).

This is a refinement of Fig. 7. Our primary concern in this refinement is to obtain a fast algorithm that obtains covers whose sizes are within some factor of the optimal cover size. As we shall see, Approx2 has a complexity $O(n^2)$. This is accomplished by limiting the search for a nonchoice cell to boundary cells. For boundary cells, we develop an easy to test sufficient (but not necessary) condition for the cell to be nonchoice. Our algorithm, therefore, is able to select only those nonchoice cells that are also boundary cells and that satisfy this sufficient condition. This is not as restrictive as it may first appear, as Approx2 moves the boundary as it progresses. So the nonchoice cells are limited to be on the current boundary rather than on the original boundary.

Let E be a support edge and let X be a cell adjacent to it. Fig. 13 shows the case when E is a left boundary edge. Draw a line through the middle of X and perpendicular to E. Move along this line away from E. Let OPP (E, X) be the first edge encountered. Let DIST (E, X) be the distance between the edge E and the edge OPP (E, X).

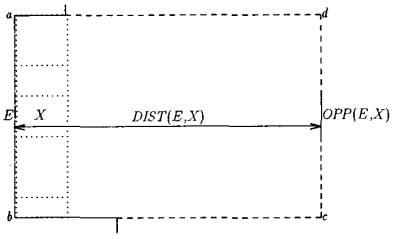


Fig. 13. A support edge E and some notations.

Theorem 8: Let P be a hole free rectilinear polygon that has been partially covered by rectangles. Let X be an uncovered cell adjacent to the support edge E. Assume that DIST $(E, X) \leq$ DIST (E, Y) for every cell Y (covered or uncovered) adjacent to E. X is a nonchoice cell.

Proof: Since DIST $(E, X) \leq$ DIST (E, Y), it follows that the unique rectangle one of whose sides is E and such that OPP (E, X) is a segment of another side (i.e., rectangle *abcd* of Fig. 13) is both a rectangle of the polygon and includes all the cells in $neb(X)$. So, X is nonchoice. □

If support edge E has a nonchoice cell adjacent to it, edge E is a *nonchoice edge*. Otherwise, it is a *choice edge*.

We shall first illustrate the strategy employed by Approx2 by means of an example. Consider the polygon of Fig. 14(a). All cells are uncovered initially. Edges e_1, e_2, e_3, and e_4 are the support edges. Cells C_1, C_2, $\cdots$, C_{18} are the boundary cells. Of these, only cells C_4 and C_{15} satisfy Theorem 8. These are detected as nonchoice cells. So edges e_1, e_2, e_3, and e_4 are all nonchoice edges. Actually, one may show that when no cells are covered, all support edges of a hole free rectilinear polygon are nonchoice. Let NSE denote the set of nonchoice support edges and let CSE denote the set of choice support edges. So, at this time, we have NSE = $\{e_1, e_2, e_3, e_4\}$ and CSE = $\varnothing$. Approx2 selects an arbitrary edge in NSE and constructs the unique rectangle described by Theorem 8. This rectangle is added to the set T of covering rectangles. Assume that edge e_1 is selected. The covering rectangle is *abcd*. In addition to other cells, this covers all cells adjacent to edges e_1 and e_2.

We shall now contract the polygon. In the case of our example, this contraction is accomplished by shifting edge e_1 right and edge e_2 up. The contracted polygon is shown in Fig. 14(b). Since the edge shifting eliminates only covered cells and there is a one-to-one correspondence between rectangles that cover at least one uncovered cell in the polygon before and after contraction, it follows that the edge shifting satisfies properties i) and ii) of a contraction. The shaded cells of Fig. 14(b) are cells that are in $covered(T) = covered(\{abcd\})$. In a more general situation, edge shifting may involve moving an edge across several cut lines or even splitting an edge and then moving the split segments (Fig. 15).

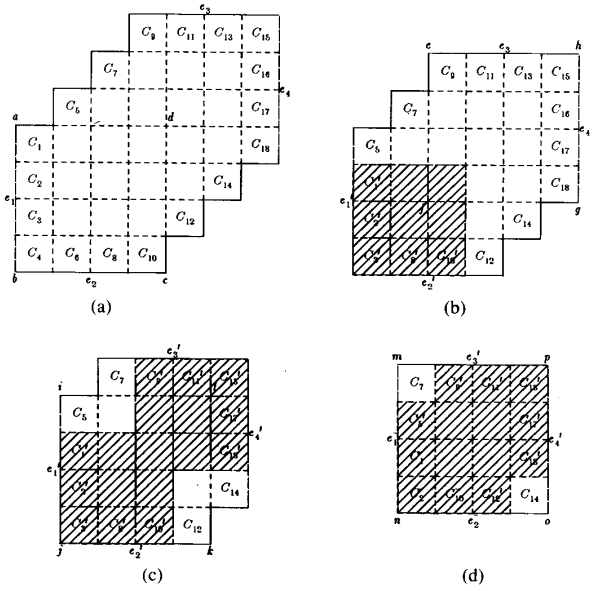


Fig. 14. Illustration of Approx2. (a) NSE = $\{e_1, e_2, e_3, e_4\}$, CSE = $\emptyset$. (b) NSE = $\{e_3, e_4\}$, CSE = $\{e_1', e_2'\}$. (c) NSE = $\emptyset$, CSE = $\{e_1', e_2', e_3', e_4'\}$. (d) NSE = $\{e_1^-, e_2^-, e_3', e_4'\}$, CSE = $\emptyset$.

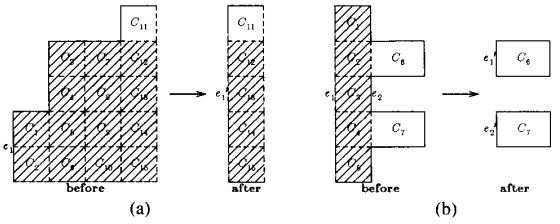


Fig. 15. (a) Shifting edge of e_1. (b) Edge splitting.

Continuing with our example, we see from Fig. 14(b) that NSE = $\{e_3, e_4\}$, and CSE = $\{e_1', e_2'\}$. If the nonchoice edge e_3 is selected, the covering rectangle *efgh* is added to T. This is the unique rectangle described in the proof of Theorem 7. The next step is to contract the polygon. This is accomplished by shifting the edges e_3 and e_4. Once again, both edges need to be shifted by one cut line. The contracted polygon is shown in Fig. 14(c). The new support edges are $\{e_1', e_2', e_3', e_4'\}$. None is a nonchoice edge. At this time e_1' is chosen. The maximal rectangle with e_1' as one side is *ijkl*. Contracting requires edges e_1' and e_2' to be shifted. The contracted polygon is shown in Fig. 14(d). All edges are nonchoice. A single rectangle *mnop* covers all remaining cells. The contracted polygon is empty and we are done. The rectangle cover obtained is $T = \{abcd, efgh, ijkl, mnop\}$.

From Theorem 4, it follows that since T contains three nonchoice rectangles, $|\text{MOC}| \geq 3$. Actually, an examination of Fig. 14(a) reveals that $|\text{MOC}| = 4$ and so T is optimal. Note that some of the rectangles in T may not be maximal with respect to the original polygon. These may be expanded to maximal polygons if desired.

```
Procedure Approx2 (P, T);
{Find an overlapping rectangle cover T of polygon P}
begin
    T := ∅;
    while P is not null do
    begin
        if there is a nonchoice support edge
        then let e be one such edge
        else let e be any left support edge;
        expand e to a maximal rectangle R one of whose sides is e; {Note: regard-
        less of whether e is choice or nonchoice, R is unique }
        T := T ∪ {R};
        contract P;
    end;
end; {of Approx2 }
```

Fig. 16. Approx2.

A high level description of Approx2 is provided in Fig. 16.

To facilitate the search for a nonchoice support edge, the construction of a maximal rectangle R with a given side e, and the contraction of a polygon P, we maintain the following data structures.

Edge data structure: We assume the edges are ordered circularly by traversing the boundary of the polygon counterclockwise. Since polygon contraction can split a connected polygon into one with several connected component polygons (Fig. 15(b)), the edge ordering is done for each component polygon independently. The following information is kept with each edge.

E1. Orientation—is this on the top, bottom, left, or right boundary of the (component) polygon?
E2. A linked list of all cells adjacent to the edge.
E3. The number of uncovered cells in the list of *E2*.
E4. The preceding edge in the edge ordering.
E5. The succeeding edge in the edge ordering.
E6. Edge status—support edge/nonchoice.
E7. Coordinates of the edge.

Note that through *E4* and *E5*, we maintain one doubly linked list of edges for each component polygon of P.

3.5. Cell Data Structure

For each cell, X, we maintain four sets of information. These sets are associated with the top, bottom, left, and right sides of the cell. We describe the information only for the top side. The remaining three information sets are similar.

ctop 1. The y coordinate of the top side of the cell.
ctop 2. Status—is the top side a segment of a boundary edge?
ctop 3. **If** status is true **then**
edge = boundary edge identifier,
opp = OPP (X, *edge*),
and CoveredDistance = distance such that all cells including and below this one upto this distance are covered cells
else next = next boundary cell above this cell;

The data structure for a cell is illustrated in Fig. 17.

Using these structures, each iteration of the **if** statement of Fig. 16 takes $O(n)$ time and the expansion of e to R can be done in $O(n)$ time. This comes from the following observations.

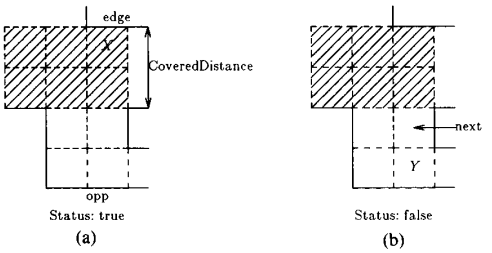


Fig. 17. Data structure for a cell. (a) **ctop** of cell X. (b) **ctop** of cell Y.

1) A contraction does not increase the number of edges.

2) The number of boundary cells is at most $2e - 4$ where e is the number of edges. (Theorem 2 and the observation that the number of edges and vertices in a polygon is the same.)

The total number of iterations of the **while** loop is $O(n)$ as each contraction either reduces the number of edges by at least 2 or results in an edge split. An edge split does not change the number of edges but replaces at least one concave vertex by a convex vertex. The number of concave vertices is $O(n)$.

The total time spent on the **if** statement can be reduced by maintaining two doubly linked lists: one of NSE edges and another of CSE edges. These lists are easily updated during a contraction. The time to contract P is proportional to the number of cells eliminated from P. So over the entire execution of the algorithm, the contraction time is $O(\text{number of cells}) = O(n^2)$. The time needed to initialize the data structures and to update them is also $O(n^2)$. Hence, the overall complexity of Approx2 is $O(n^2)$.

While experimental results reported in this paper indicate that Approx2 often obtains MOC's or at least covers very close to optimal, the example of Fig. 18 shows that it is possible for Approx2 to obtain a cover that has $2\,|\text{MOC}| - 4$ rectangles. Theorem 9 shows that the number of rectangles in the cover generated by Approx2 never exceeds $2\,|\text{MOC}| - 1$.

Theorem 9: Let P be any hole free rectilinear polygon. Let T be the cover generated by Approx2. $|T| < 2\,|\text{MOC}|$.

Proof: Let U denote the number of LES of P that have at least one uncovered cell adjacent to them. Initially, $U = \#\text{LES}$. Each time Approx2 adds a rectangle R to T, U decreases by at least one. To see this, observe that the edge e expanded to obtain the maximal rectangle R is either a left support edge or has a nonchoice cell adjacent to it (or both). In the former case, it is clear that following the addition of R to T, U decreases. So, consider the case when e is a NSE that is not a left support edge. Let C be a nonchoice cell adjacent to e. Since the sides of all cells are segments of extended edges, the left side of C is part of some LES of P. Since C is a nonchoice cell, all cells adjacent to this LES must be covered following the inclusion of R into T. Hence, U must decrease by at least one following the addition of R to T.

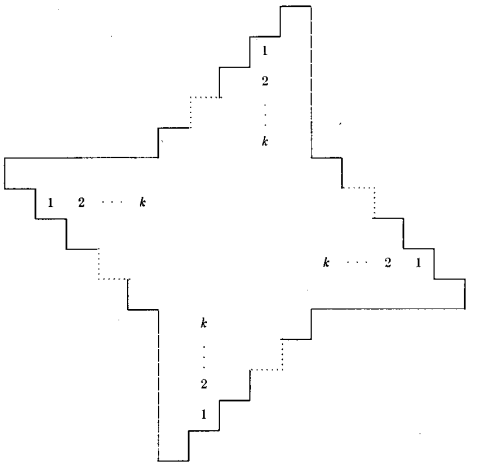


Fig. 18. A bad example for Approx2. Approx2 may generate a cover with $4k$ rectangles, while $|\text{MOC}| = 2k + 2$.

Since U decreases by at least one each time a rectangle is added to T and since $U = \#\text{LES} < 2\,|\text{MOC}|$ (Theorem 6), $|T| < 2\,|\text{MOC}|$. Note that when U becomes zero, all cells of P have been covered. □

3.6. *Approximation Algorithm 3 (Approx3)*

Approx3 initially behaves like Approx2. It repeatedly selects nonchoice edges using Theorem 8 and contracts the polygon. It deviates from Approx2 on the first iteration when no nonchoice edge can be found. At this time, it examines all uncovered cells to see if any is nonchoice (i.e., $neb(X) \cap uncovered(T)$ is contained in some polygon of P). If so, the covering rectangle for $neb(X) \cap uncovered(T)$ is added to the rectangle set T. Otherwise, an uncovered cell that is adjacent to a left support edge is selected arbitrarily and covered by a maximal rectangle R. In both cases, the polygon is contracted. This selection of cells and polygon contraction is repeated until all cells are covered. When selecting a nonchoice cell, preference is given to one adjacent to a support edge.

Theorem 10: Let P be any hole free rectilinear polygon. Let Z be a cover generated by Approx3. $|Z| \leq 2\,|\text{MOC}| - 1$.

Proof: Same as that for Theorem 9. □

In each iteration of Approx3, one rectangle is added to the cover. The total number of rectangles in the constructed cover is $O(n)$. Each time a rectangle is added, $O(n^2)$ cells have to be checked for the nonchoice property. For each cell, we can check the nonchoice property and also construct a maximal rectangle in $O(n)$ time. It is easy to see how this may be done if each cell keeps the following information in addition to that given in Section III-3.4.

a) Let a, b, c, and d be the four edges closest to the cell and above, below, to the right, and to the left of the cell, respectively. The cell keeps a pointer to each of these four edges.

b) Each cell has a pointer to the next uncovered cell in each of the four directions.

c) Each cell has a Boolean field that identifies it as having been covered or not.

The overall complexity of Approx3 is, therefore, $O(n^4)$.

IV. Experimental Results

The three heuristics Approx1, Approx2, and Approx3 were programmed in Pascal. In the case of Approx3, our program was a relaxed version of the algorithm described in Section III-3.6. In this relaxation, when no nonchoice cell is available, any support edge (rather than necessarily a left support edge) may be selected. We also programmed variants of Approx1 and Approx2. These are called Approx1′ and Approx2′, respectively. These, respectively, run Approx1 and Approx2 four times. The input for these four runs is: original polygon P, P rotated by 90°, P rotated by 180°, and P rotated by 270°. The smallest cover is selected. In the case of Approx1′ the rotations are equivalent to modifying Approx1 to select e as a left, bottom, right, and top support edge, respectively. The rotations have a similar effect on Approx2 with respect to the selection of e when there is no nonchoice support edge.

To determine the effectiveness of our heuristics in finding near optimal covers, we ran several tests using hole free rectilinear polygons that were convex in either the x or y directions. By limiting our tests to these polygons, we were able to use the algorithm of [5] to obtain optimal covers against which the covers obtained by our heuristics could be compared. Three test sets, each containing 100 polygons, were used. In the first, each polygon had approximately 100 vertices; in the second approximately 160 vertices; and in the third approximately 280 vertices.

Table I gives the number of polygons (out of 100) in each test set for which an optimal cover was generated. For comparison purposes we have also included the results of using the MNC as a cover (recall that $|\text{MNC}| \leq 2\,|\text{MOC}| - 1$. So using the MNC guarantees the same worst-case bound as using any of our heuristics). All of our heuristics performed significantly better than MNC. Approx3 clearly outperformed the others and obtained an optimal cover in 288 of the 300 cases run.

In no case was the solution produced by Approx2 or Approx3 inferior to the MNC. However, Approx1 produced six solutions inferior to the MNC. Further, the Approx3 solution was always at least as good as the Approx2 solution which in turn was always at least as good as the Approx1 solution. Approx1′ and Approx2′ both obtained a smaller cover than obtained by Approx3 in only 1 instance. Approx1′ obtained better solutions than Approx2 for 42 of the 300 instances in our test set. However, Approx1′ never did better than Approx2′.

Let MNC (P), Approx1 (P), Approx1′ (P) Approx2 (P), Approx2′(P), and Approx3 (P), respectively, denote the size of the cover generated by the associated approximation algorithm with input polygon P. Let B be a polygon set. The average percentage deviation from the optimal cover size, MOC (P), by algorithm X is

$$\text{AVG}\,(B, X) = \frac{1}{|B|} \sum_{P \in B} \frac{X(P) - \text{MOC}\,(P)}{\text{MOC}\,(P)} * 100$$

where $X \in \{\text{MNC, Approx1, Approx1′, Approx2, Approx2′, Approx3}\}$.

TABLE I
Number of Optimal Covers (Out of 100 Polygons)

size	*MNC*	*Approx 1*	*Approx 1′*	*Approx 2*	*Approx 2′*	*Approx 3*
≃100	4	42	62	71	78	99
≃160	2	41	57	58	69	96
≃280	1	32	47	48	58	93

The value of AVG (B, X) for our three test sets is given in Table II. Once again, we see that Approx3 is significantly superior to the other heuristics. Table III gives the range of the deviation from optimal. For this, only polygons for which the cover generated is suboptimal are considered.

The smaller value in each range is

$$\text{small}\,(B, X) = \min\left\{\frac{X(P) - \text{MOC}\,(P)}{\text{MOC}\,(P)} * 100 \;\middle|\; P \in B,\ X(P) > \text{MOC}\,(P)\right\}.$$

The larger value in each range is

$$\text{large}\,(B, X) = \max\left\{\frac{X(P) - \text{MOC}\,(P)}{\text{MOC}\,(P)} * 100 \;\middle|\; P \in B\right\}.$$

Table IV gives the runtime for our three test sets on an Apollo DN3500 workstation. We see that MNC and Approx1 have comparable run times. Approx3 takes significantly more time than the others.

Additional experiments using concave polygons were performed. Let $k_H(k_V)$ be the number of horizontal (vertical) support edges minus 1. Let $k = \min(k_V, k_H)$. The polygons in the new test set were classified by their k value. The number, n, of the polygon vertices was set to be 20, 30, 40, $\cdots$, 100, 200, or 300. For each combination of k and n, 50 polygons were generated. Some were generated randomly and others by combining together random polygons with a small k to get one with a bigger k. For each value of k we had a total of 500 polygons in our test set. The performance of our heuristics is measured relative to that of MNC.

The average percentage deviation from MNC by algorithm X is

$$\text{AVG}\,(B, X) = \frac{1}{|B|} \sum_{P \in B} \frac{\text{MNC}\,(P) - X(P)}{X(P)} * 100$$

where $X \in \{\text{Approx1, Approx1′, Approx2, Approx2′, Approx3}\}$.

TABLE II
AVG (B, X)

size	*MNC*	*Approx 1*	*Approx 1'*	*Approx 2*	*Approx 2'*	*Approx 3*
≃100	26.31	5.91	3.02	2.73	1.32	0.03
≃160	24.98	5.50	3.18	2.96	1.35	0.13
≃280	31.18	6.51	4.38	3.74	1.99	0.18

TABLE III
SMALL (B, X) ~ LARGE (B, X)

size	*MNC*	*Approx 1*	*Approx 1'*	*Approx 2*	*Approx 2'*	*Approx 3*
≃100	2.78~60.70	2.86~28.57	2.50~20.83	2.70~25.00	2.70~14.29	2.86~2.86
≃160	1.85~68.42	1.85~24.32	2.00~20.59	1.92~20.59	1.92~14.71	1.96~5.26
≃280	1.22~64.47	1.09~22.86	1.09~20.51	0.97~20.83	0.97~10.26	1.02~5.21

TABLE IV
RUNTIME (MILLISECONDS)

size	*MNC*	*Approx 1*	*Approx 1'*	*Approx 2*	*Approx 2'*	*Approx 3*
≃100	46.44	80.25	316.36	81.70	324.88	2737.15
≃160	66.39	121.94	490.36	149.31	595.92	7961.18
≃280	179.21	281.87	1136.32	556.70	2201.84	169385.80

TABLE V
AVG (B, X)

k	*Approx 1*	*Approx 1'*	*Approx 2*	*Approx 2'*	*Approx 3*
1	12.03	17.28	16.94	17.79	19.06
2	3.69	9.02	9.60	10.26	10.90
3	4.07	8.36	9.85	10.47	11.24
4	2.70	6.18	7.05	7.47	7.85
5	2.21	5.50	6.32	6.56	6.78
6	1.22	4.32	5.03	5.04	5.06
7	1.13	3.81	4.56	4.58	4.59
8	1.50	4.23	5.11	5.13	5.14
9	2.36	5.11	6.42	6.79	6.89
10	2.29	5.42	6.18	6.77	6.86

The value of AVG (B, X) for test sets with different k is given in Table V. Approx3 is superior to Approx2' which is superior to Approx2 which, in turn, is superior to Approx1' and Approx1. Approx1' is superior to Approx1. Table VI gives the range of the deviation from MNC. The smaller value in each range is

$$\text{small}\ (B, X) = \min\left\{\frac{\text{MNC}\ (P) - X(P)}{X(P)} * 100 \mid P \in B,\ X(P) \neq \text{MNC}\ (P)\right\}.$$

The larger value in each range is

$$\text{large}\ (B, X) = \max\left\{\frac{\text{MNC}\ (P) - X(P)}{X(P)} * 100 \mid P \in B\right\}.$$

Table VII gives the runtime for our test sets on an Apollo DN3500 workstation. We see that Approx1 has a larger overhead than MNC and Approx2. Approx3 takes more time than the others. The run time of Approx3 decreases as the value of k increases because as k increases the ratio (number of nonchoice cells)/(number of NSE) decreases. So, Approx3 spends relatively less time searching for nonchoice cells. This search is time consuming.

TABLE VI
SMALL (B, X) ~ LARGE (B, X)

k	*Approx 1*	*Approx 1'*	*Approx 2*	*Approx 2'*	*Approx 3*
1	-16.67~76.00	0.68~76.00	-4.17~76.00	0.83~76.00	0.83~76.00
2	-15.38~44.12	-4.17~58.06	-1.52~58.06	2.38~58.06	2.38~61.96
3	-13.79~31.43	-5.00~34.29	2.94~45.95	2.94~45.95	2.94~54.17
4	-16.67~42.47	-4.00~42.47	2.38~50.72	2.38~50.72	2.38~63.93
5	-15.38~35.96	-5.88~35.96	2.38~43.52	2.38~46.23	2.38~56.25
6	-17.65~23.17	-3.85~29.49	2.04~29.49	2.04~31.17	2.04~31.17
7	-11.76~23.46	-3.57~25.32	2.08~32.00	2.08~33.78	2.08~33.78
8	-12.00~22.09	-3.45~29.87	1.35~29.17	1.35~30.99	1.35~32.86
9	-13.04~22.90	-3.57~30.97	1.37~30.95	1.37~35.78	1.37~39.62
10	-9.76~25.93	-3.33~34.21	2.27~31.48	2.27~36.00	2.27~37.81

TABLE VII
RUNTIME (MILLISECONDS)

k	*MNC*	*Approx 1*	*Approx 1'*	*Approx 2*	*Approx 2'*	*Approx 3*
1	19.29	90.80	363.26	140.64	551.13	15143.27
2	18.75	85.93	326.36	95.99	375.65	3226.73
3	18.13	83.58	324.45	80.58	317.45	1709.86
4	17.87	81.37	309.07	68.99	274.70	899.86
5	17.57	79.30	294.60	59.22	234.92	605.21
6	18.85	86.85	323.21	64.72	257.02	663.46
7	19.15	87.77	326.58	65.49	259.52	668.32
8	21.63	94.88	348.52	65.55	258.81	465.67
9	23.34	104.87	384.41	71.25	281.59	517.08
10	21.53	97.67	355.01	63.78	251.18	367.03

Our experiments show that while $2\,|\text{MOC}| - 1$ is a bound for the tested heuristics, all often perform better than this bound (AVG (B, X) $\simeq$ 100 percent when the bound is achieved). If runtime is not a consideration, Approx3 should be used. When it is, Approx2 or Approx2' are recommended.

REFERENCES

[1] J. L. Bentley and M. I. Shamos, "A problem in multivariate statistics: Algorithms, data structures, and applications," in *Proc. 15th Ann. Allerton Conf. on Communication, Control, and Computing*, pp. 193-201, 1977.

[2] S. Chaiken, D. J. Kleitman, M. Saks, and J. Shearer, "Covering regions by rectangles," *SIAM J. Algebraic Descrete methods*, vol. 23, no. 4, pp. 394-410, Dec. 1981.

[3] J. P. Cohoon, "Fast channel graph construction," Dep. Comput. Sci., Univ. of Virginia, Tech. Report, 1987.

[4] J. C. Culberson and R. A. Reckhow, "Covering polygons is hard," in *Proc. 29th Ann. Symp. on Foundations of Computer Science*, pp. 601-611, 1988.

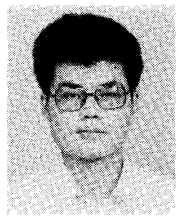
[5] D. S. Franzblau and D. J. Kleitman, "An algorithm for constructing regions with rectangles: independence and minimum generating sets for collections of intervals," in *Proc. 16th Ann. ACM Symp. on Theory of Computing*, Washington, DC, pp. 167-174, 1984.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability*. San Francisco, CA: Freeman, 1979.

[7] K. D. Gourley and D. M. Green, "A polygon-to-rectangle conversion algorithm," *IEEE Comput. Graph.*, vol. 3, pp. 31-36, Jan./Feb. 1983.

[8] E. Horowitz and S. Sahni, *Foundamentals of Data Structures in Pascal*, 2nd ed. MD: Computer Science Press,Inc., 1986.

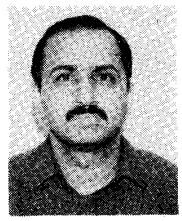
[9] H. Imai and T. Asano, "Efficient algorithms for geometric graph search problems," *SIAM J. Comput.*, vol. 15, no. 2, pp. 478-494, May 1986.
[10] E. Lodi, F. Luccio, C. Mugnai, and L. Pagli, "On two-dimensional data organization I," *Fundamenta Informaticae*, vol. 2, pp. 211-226, 1979.
[11] W. T. Liou, J. J. M. Tan, and R. C. T. Lee, "Minimum partitioning simple rectilinear polygons in $O(n \log \log n)$ time," in *Proc. Fifth Ann. Symp. on Computational Geometry*, pp. 344-353, 1989.
[12] W. J. Masek, "Some *NP*-complete set covering problems," unpublished manuscript, MIT, 1978.
[13] E. M. McCreight, "Efficient algorithms for enumerating intersecting intervals and rectangles," Rep. CSL-80-9, Xerox Palo Alto Res. Ctr., June 1980.
[14] ——, "Priority search trees," *SIAM J. Comput.*, vol. 14, no. 2, pp. 257-276, May 1985.
[15] S. Nahar and S. Sahni, "A time and space efficient net extractor," in *Proc. 23rd Design Automation Conf.*, pp. 411-417, 1986.
[16] S. Nahar and S. Sahni, "A fast algorithm for polygon decomposition," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 478-483, Apr. 1988.
[17] T. Ohtsuki, "Minimum dissection of rectilinear regions," in *Proc. 1982 Int. Symp. on Circuits and Systems* (ISCAS), pp. 1210-1213, 1982.
[18] J. K. Outerhout, "Corner stitching: A data-structuring technique for VLSI layout tools," *IEEE Trans. Computer-Aided Design*, vol. CAD-3, pp. 87-100, Jan. 1984.



San-Yuan Wu received the B.S. degree in mathematics from the National Taiwan University in 1979 and the M.S. degree in mathematics from the University of Minnesota in 1987. He received the Ph.D. degree in computer science from the University of Minnesota in 1989.

He joined CADENCE System Inc. in 1989. His research interests include design and analysis of algorithms, computer-aided design of integrated circuits, computational geometry, and parallel computation.

*



Sartaj Sahni (M'00-SM'00-F'00) received the B.Tech. degree in electrical engineering from the Institute of Technology, Kanpur, India, and the M.S. and Ph.D. degrees in computer science from Cornell University.

He is the co-author and author of *Fundamentals of Data Structures*, *Fundamentals of Data Structures in Pascal*, *Fundamentals of Computer Algorithms*, *Concepts in Discrete Mathematics*, and *Software Development in Pascal*. He is an editor for the *Journal of Parallel and Distributed Computing* and is a member of the editorial boards of *Information and Software Technology* and *Computer Systems: Science and Engineering*. He is currently a Professor of computer science at the University of Minnesota.