Fast Algorithms To Partition Simple Rectilinear Polygons*

San-Yuan Wu[1] and Sartaj Sahni[2]

ABSTRACT

Two algorithms to partition hole-free rectilinear polygons are developed. One has complexity ~ O(kn) and the other O($n$log$k$) where n is the number of vertices in the polygon and k is the smaller of the number of vertical and horizontal inversions of the polygon. k is a measure of the simplicity of a polygon. Since k is small for most practical polygons, our algorithms are fast in practice. Experimental results comparing our algorithms with that of Imai and Asano [1] are also presented.

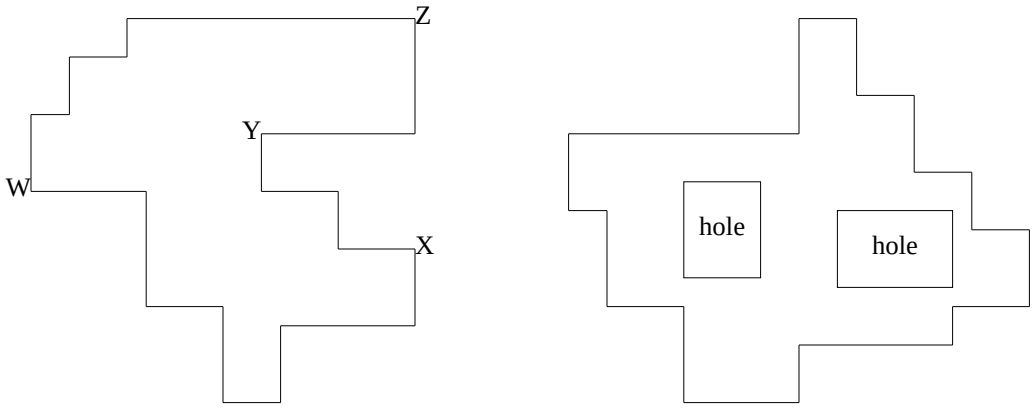
Key words and phrases

Rectilinear hole-free polygons, partition, complexity, computational geometry

*This research was supported, in part, by the National Science Foundation under grants MIP 86-17374 and MIP-9103379.
1. Cadence Design Systems, Inc., 555 Riveroaks Parkway, Bldg. 2,, San Jose, CA 95134.
2. Department of CIS, CSE 301, University of Florida, Gainesville, FL 32611.

1 Introduction

Rectilinear polygons (Figure 1) arise frequently in VLSI layout and artwork analysis, computer graphics, databases, image processing, etc. ([2], [3], [4], [5], [6]). The functions to be performed on rectilinear polygons are often more easily performed using either a rectangle partition or a rectangle cover. In either case the rectilinear polygon is decomposed into a set of rectangles whose union is the original rectilinear polygon. If this set of rectangles is disjoint it is a *partition*. In a *cover* the rectangles need not be disjoint.



(a) hole-free (b) with holes

Figure 1: Rectilinear polygons.

A *minimal nonoverlapping cover* (*MNC*) of rectilinear polygon P is a rectangle partition of P that contains the fewest possible number of rectangles. Ohtsuki [7] has developed an $O(n^{5/2})$ algorithm to find the *MNC* of a rectilinear polygon with n vertices. Imai and Asano [1] have developed an $O(n^{3/2} \log n)$ algorithm to find the *MNC* and Liou, Tan, and Lee [8] have an $O(n \log\log n)$ algorithm for hole-free rectilinear polygons. The algorithm of [8] is not expected to perform better than that of [1] as it uses finger search trees that are known to be impractical (‘‘Finger search trees are sufficiently complicated that one would probably not want to use them in an actual implementation’’, pg. 165 of [9]).

Nahar and Sahni [10] introduced a complexity measure for hole-free rectilinear polygons. They defined the number of *horizontal inversions*, k_H, to be the minimum number of changes in

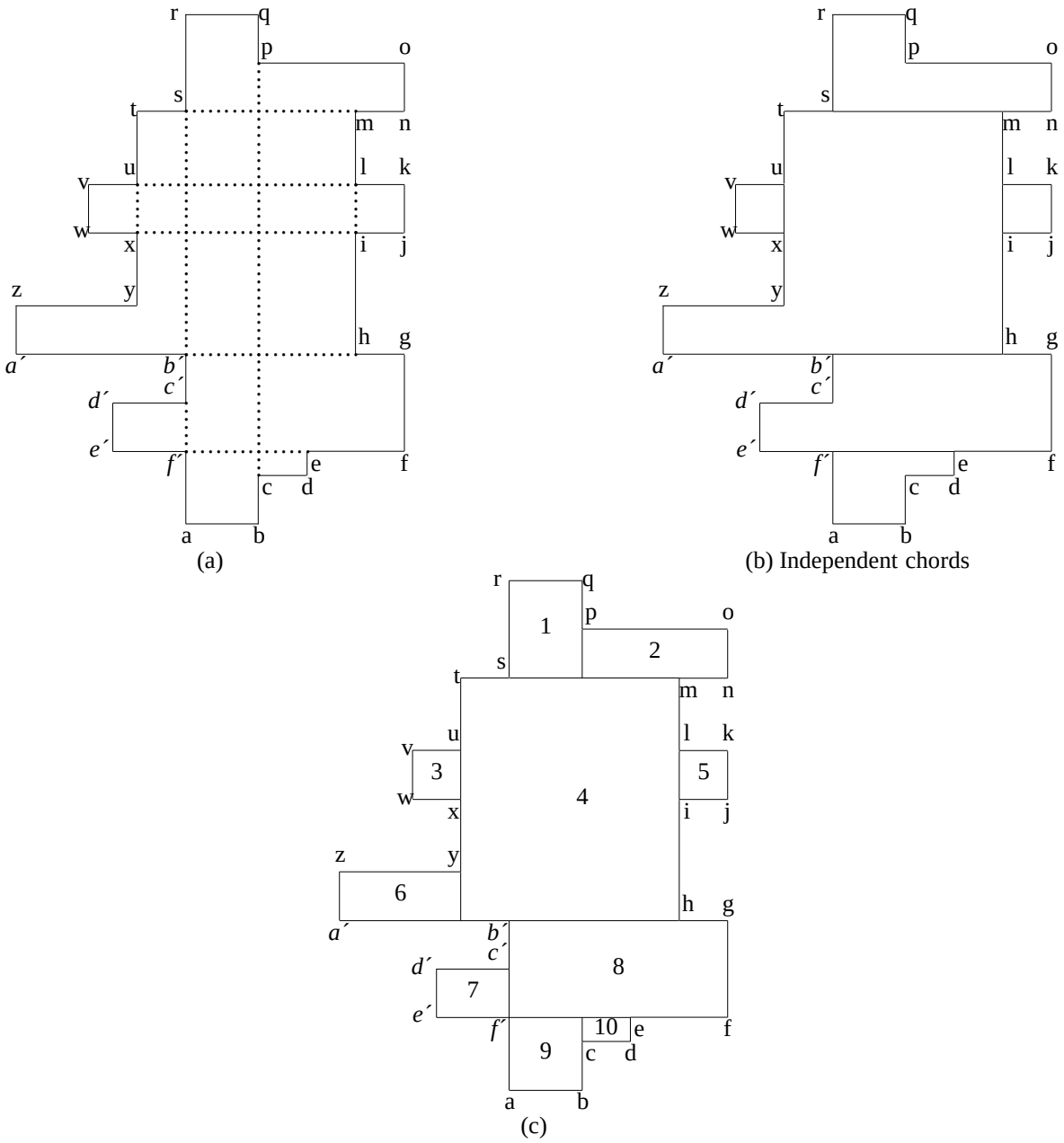


Figure 2: MNC of a rectilinear polygon

horizontal direction during a walk around the polygon divided by 2. For example if we walk around the polygon of Figure 1(a) beginning at vertex *W* such that the interior is to our left, then we first walk rightwards up to vertex *X* then the direction is leftwards up to vertex *Y* and then it it rightwards up to *Z* and then leftwards up to *W*. The direction of horizontal motion changes at *W*, *X*, *Y*, *Z*. So, $k_H = 2$. For the polygon of Figure 2(a) $k_H = 5$. The number of *vertical inversions*, k_V, is

defined analagously. $k_V = 1$ for the polygons of Figures 1(a) and 2(a). The *inversion number*, k, of a hole-free rectilinear polygon is defined to be $k = \min \{ k_H, k_V \}$. The polygon of Figure 1(a) has inversions number 1 and that of Figure 2(a) has inversion number 1 too. A polygon is simple if it is hole-free and has a small inversion number. Nahar and Sahni [10] examined 2869 polygons from VLSI mask data provided by UNISYS and found that 85% of these have $k = 1$ and 95% have $k \leq 2$ (Figure 3). They took advantage of this observation and developed a fast algorithm to partition hole-free rectilinear polygons under the restriction that only horizontal cuts are permitted. Their algorithm outperforms the scan line method on simple (and hence practical) polygons.

In this paper we extend the work of Nahar and Sahni [10] and obtain an ~O(kn) and an O($n \log k$) (recall that $k = \min\{k_H, k_V\}$) algorithm to find the *MNC* of a hole-free rectilinear polygon. Thus our algorithms permit both horizontal and vertical cuts while that of [10] permits only horizontal cuts. Experiments conducted with randomly generated polygons with low k show that our algorithms outperform that of [1]. We do not compare our algorithms with that of [10] as the algorithm of [10] does not obtain MNCs. Rather, it obtains a minimum partitioning under the assumption that only horizontal (rather than both horizontal and vertical) cuts are permitted.

As noted earlier, most of the polygons that arise in VLSI layout and artwork analysis have a small k. Hence our algorithms will reduce the run time of current VLSI software that uses rectilinear partitioning. Note that our algorithms are not recommended for rectilinear polygons with high k. In the worst case, k can be O(n) and the complexity of our algorithms becomes ~O(n^2) and O($n \log n$), respectively. While the latter is still better than the asymptotic complexity of the algorithm of [1], it should be noted that O($n^{3/2} \log n$) is the only the worst case run time of the algorithm of Imai and Asano [1] and that there algorithm takes much less time on many instances. Hence, the algorithm of Imai and Asano can outperform our second algorithm on rectilinear polygons for which k is not small.

Section 2 provides background material necessary for the development of our algorithms. This section also defines much of the terminology we shall use. In Section 3, we develop the ~O(kn) MNC algorithm. The O($n \log k$) algorithm is developed in Section 4 and experimental

results are provided in Section 5.

Number of vertices	Number of polygons
$0 < n \leq 10$	1739
$10 < n \leq 20$	926
$20 < n \leq 30$	84
$30 < n \leq 40$	25
$40 < n \leq 90$	91

Figure 3: Distribution of polygons by number of vertices

2 Background

2.1 MNC Algorithm of [7]

We assume that the rectilinear polygon is oriented so that its vertical edges are parallel to the y-axis and its horizontal edges are parallel to the x-axis. 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 at this vertex is 90^0 (270^0). A *chord* is a line segment that lies wholly within the polygon and joins two cohorizontal or two covertical concave vertices. A set of chords is *independent* iff no two chords of the set intersect. Chords are used to generate an MNC of a rectilinear polygon. Since the MNC is a rectilinear partition, we may restrict ourselves to horizontal and vertical chords.

Example 1: Consider the hole-free polygon of Figure 2(a). Some of the cohorizontal vertex sets are (*r*, *q*), (*p*,*o*), (*t*, *s*, *m*, *n*), and (*v*, *u*, *l*, *k*). Some of the covertical vertex sets are (*z*, *a´*), (*w*, *v*), and (*t*, *u*, *x*, *y*). Some of the convex verteices are *a*, *b*, *d*, *f*, and *g*. And some of the concave vertices are *c*, *e*, *h*, and *i*. The complete set of horizontal chords is {*sm*, *ul*, *ix*, *hb´*, *ef´*}, and the complete set of vertical chords is {*ux*, *sb´*, *c´f´* , *pc*, *il*}. Notice that horizontal chords are always independent. So also are vertical chords. The chords *sm* and *sb´* are not independent as they intersect at *s*. □

The *MNC* algorithm of [7] uses the following three steps:

Step 1: Find a maximum independent set of chords (i.e., a maximum cardinality set of independent chords).

Step 2: Draw the chords in this maximum independent set. This partitions the polygon into smaller rectilinear polygons.

Step 3: From each of the concave vertices from which a chord was not drawn in step 2 draw a maximum length vertical line that is wholly within the smaller rectilinear polygon created in step 2 that contains this vertex.

In step 3 we may draw horizontal lines or vertical ones. The rectangles formed by the polygon edges, chords of step 2 and lines of step 3 form an *MNC* [7].

Example 2: The chord set {*ux*, *li*, *sm*, *hb´*, *ef´*} is a maximum independent chord set for the polygon of Figure 2(a). Figure 2(b) shows the polygon after the chords in this set are drawn. Figure 2(c) shows the polygon after vertical lines are drawn as in Step 3. The rectangles in the *MNC* are numbered 1 through 10. □

The complexity of steps 2 and 3 of the algorithm of [7] is $O(n\log n)$ where n is the number of vertices in the rectilinear polygon. A simple sweep line algorithm can be used for this. We concentrate on step 1. A maximum independent chord set may be found [7] by first setting up a bipartite graph $G = (H \cup V, E)$ where each vertex in H represents a horizontal chord and each vertex in V represents a vertical chord. There is an edge between two vertices iff the chords they represent intersect (i.e., cross or meet). Figure 4 shows the bipartite graph for the polygon of Figure 2(a). The problem of finding a maximum independent chord set now becomes one of finding a maximum independent vertex set (MIS) in a bipartite graph (two vertices are independent iff there is no edge between them).

A maximum independent vertex set of a bipartite graph is related to a maximum matching by the following theorem.

Theorem 1: [11] Let $G = (H \cup V, E)$ be a bipartite graph. Let M be a maximum matching of G and let F be the set of free vertices relative to this matching (i.e., F contains all vertices of G that

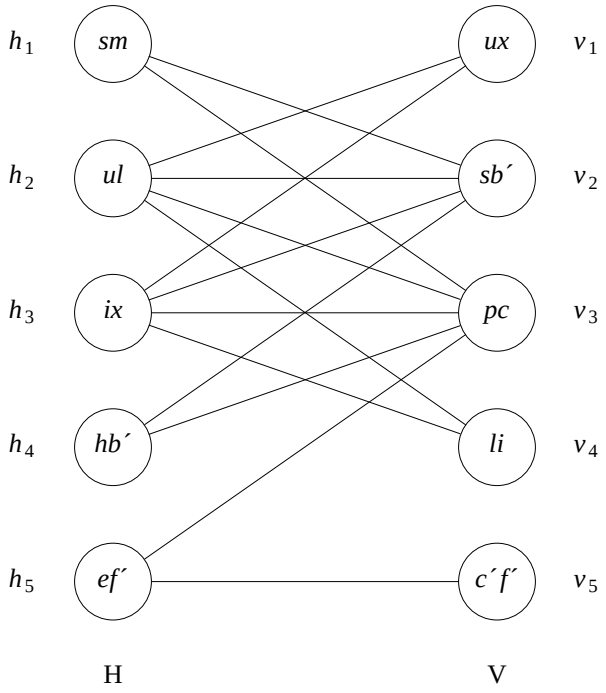


Figure 4: Intersection graph of chords of polygon in Figure 2 (a).

are not matched in M).

(a) Every MIS of G has cardinality $|H| + |V| - |M|$.

(b) There is an MIS S of G such that $F \subseteq S$ and for every edge $(v, h) \in M$ exactly one of v and h is in S. □

Using the above theorem one may obtain the algorithm of Figure 5 to construct an MIS S from a maximum matching M [12].

Example 3: Consider the maximum matching $M = \{(sm, sb´), (ux, ul), (li, ix), (hb´, pc), (c´f´, ef´)\}$ for the bipaprtite graph of Figure 4.

Iteration 1: $F = \varnothing$, $S = \varnothing$, and $M = \{(sm, sb´), (ux, ul), (li, ix), (hb´, pc), (c´f´, ef´)\}$. $(sm, sb´)$ is picked from M; sm is added to S and $hb´$ is added to F.

Iteration 2: $F = \{hb´\}$, $S = \{sm\}$, and $M = \{(ux, ul), (li, ix), (c´f´, ef´)\}$. $hb´$ is added to S and processed.

```
Procedure MaxInd(G,M,S);
{Given a bipartite graph G = ( H∪V, E) and a maximum matching M ⊆ E. Find an MIS S such that
S ⊆ H∪V}
begin
    S := ∅;
    F := {u | u ∈ H∪V and (u, x) ∉ M for any x}; {Free vertices}
    while (F ≠ ∅ ) or (M ≠ ∅) do
    begin
        if F ≠ ∅ then
        begin {add a free vertex to S}
            Let u ∈ F; F := F - {u}; S := S ∪ {u};
        end
        else {add a vertex in M to S}
        begin
            Let (u, v) ∈ M; M := M - {(u, v)};
            E := E - {(u, v)}; S := S ∪ {u};
        end;
        { Process vertex u }
        for all (u, v) ∈ E do
        begin
            E := E - {(u, v)};
            if there is an h such that (v, h) ∈ M then
            begin
                M := M - {(v, h)}; F := F ∪ {h}; {h is free}
            end;
        end; {of for}
    end;{of while}
end;{of MaxInd}
```

Figure 5

Iteration 3: $F = \emptyset$, $S = \{sm, hb'\}$, and $M = \{(ux, ul), (li, ix), (c'f', ef')\}$. (ux, ul) is considered; ux is added to S and li is added to F.

Iteration 4: $F = \{li\}$, $S = \{sm, hb', ux\}$, and $M = \{(c'f', ef')\}$. li is added to S and processed.

Iteration 5: $F = \emptyset$, $S = \{sm, hb', ux, li\}$, and $M = \{(c'f', ef')\}$. $(c'f', ef')$ is picked. $c'f'$ is added to S.

We get the MIS $S = \{sm, hb', ux, li, c'f'\}$. □

One may easily verify that procedure MaxInd constructs an independent set of cardinality $|H| + |V| - |M|$. From Theorem 1 it follows that this is an MIS. Step 1 of Ohstuki's algorithm [7] is implemented as:

Step 1a: Find a maximum matching of the bipartite graph that represents the chords.

Step 1b: Find a maximum independent chord set using the maximum matching of Step 1a and procedure MaxInd.

Ohtsuki used the $O(n^{5/2})$ bipartite graph matching algorithm of Hopcroft and Karp [13] for Step 1a. Since the remaining steps take less time than this, Ohtsuki's implementation has complexity $O(n^{5/2})$. Imai and Asano [1] have developed an $O(n^{3/2} \log n)$ algorithm to find a maximum matching of the intersection graph of horizontal and vertical line segments. This results in an $O(n^{3/2} \log n)$ implementation of Ohtsuki's method to find an *MNC*.

2.2 Maximum Matching In Convex Bipartite Graphs

The neighborhood, NEB(x), of a vertex x is defined to be the set of vertices adjacent to vertex x. For the bipartite graph of Figure 4, NEB(h_1) = $\{v_2, v_3\}$ and NEB(v_1) = $\{h_2, h_3\}$. A bipartite graph $G = (H \cup V, E)$ is *convex on V* iff the vertices in V can be ordered such that for every $h \in H$, NEB(h) = [FIRST(h), LAST(h)] (i.e. all vertices in the ordering beginning at vertex FIRST(h) and going on to vertex LAST(h)) or *null*. *Convex on H* can be defined similarily. The bipartite graph of Figure 4 is convex on H as

NEB(v_1) = $\{h_2, h_3\}$ = $[h_2, h_3]$.

NEB(v_2) = $\{h_1, h_2, h_3, h_4\}$ = $[h_1, h_4]$.

NEB(v_3) = $\{h_1, h_2, h_3, h_4, h_5\}$ = $[h_1, h_5]$.

NEB(v_4) = $\{h_2, h_3\}$ = $[h_2, h_3]$.

NEB(v_5) = $\{h_5\}$ = $[h_5, h_5]$.

A vertex x is *matchable* iff there is a vertex $y \in$ NEB(x) such that NEB(y) $\subseteq$ NEB(z) for all $z \in$ NEB(x). Vertex h_1 of Figure 4 is matchable as NEB(h_1) = $\{v_2, v_3\}$ and NEB(v_2) $\subseteq$ NEB(z) for all $z \in$ NEB(h_1). The importance of a matchable vertex x is that we can show that there is a maximum matching M that contains (x, y).

Theorem 2: Let x be a matchable vertex in a graph G. Let y be such that $y \in$ NEB(x) and NEB(y) $\subseteq$ NEB(z) for all $z \in$ NEB(x). There is a maximum matching that contains the edge (x, y).

Proof: Let M' be a maximum matching of G that does not contain the edge (x, y). Since M' is a

maximum matching and (x, y) is an edge of G, both x and y cannot be free relative to M'. If only x is free then $(y, w) \in M'$ and we can construct $M = M' - \{(y, w)\} \cup \{(x, y)\}$. If only y is free then $(x, a) \in M'$ and we construct the maximum matching $M = M' - \{(x, a)\} \cup \{(x, y)\}$. If neither x nor y is free then $(x, a) \in M'$ and $(y, w) \in M'$. Since $a \in \text{NEB}(x)$, $\text{NEB}(y) \subseteq \text{NEB}(a)$. Now since $w \in \text{NEB}(y)$, $w \in \text{NEB}(a)$. So, (w, a) is an edge of G. From M', we may construct the maximum matching $M = M' - \{(x, a), (y, w)\} \cup \{(x, y), (w, a)\}$. □

Theorem 2 leads us to formulate the following strategy to find a maximum matching:

Step 1: If G has no edges then stop.

Step 2: If G has no matchable vertex then fail and stop.

Step 3: Select a matchable vertex x and a vertex y as in Theorem 2. Add (x, y) to M. Delete x, y, and all edges incident to x and y from G.

Step 4: Go to Step 1.

It is evident that a maximum matching is found so long as we do not have a failure termination in Step 2. Graphs which do not cause the preceding strategy to fail are called *matchable* graphs. For a bipartite graph $G = (H \cup V, E)$ that is convex on V and in which $V = \{v_1, v_2, ..., v_{n_V}\}$ is ordered as required by the convex property the above strategy to find a maximum matching takes the form given in procedure MaxMatch (Figure 6). This procedure is due to [12]. Step 2 has been eliminated as for any iteration i of the **for** loop if $\text{NEB}(v_i) \neq \varnothing$ then for every $h \in \text{NEB}(v_i)$, $\text{NEB}(h) = [v_i, \text{LAST}(h)]$. By the choice of h_j, $\text{NEB}(h_j) \subseteq \text{NEB}(h)$ for every $h \in \text{NEB}(v_i)$. So, vertex v_i is matchable and by Theorem 2 (h_j, v_i) may be added to M. As a result of this we may conclude that every bipartite graph that is convex on V is matchable.

Starting with a maximum matching, the MIS of a bipartite graph that is convex on V can be obtained in $O(n_H + n_V)$ time where $n_H = |H|$ and $n_V = |V|$. For this we use a strategy slightly different from that used in procedure MaxInd (Figure 4). We assume that the maximum matching M is obtained by using procedure MaxMatch (Figure 6) and that the vertices in V are ordered as required by the convexity property and those in H are ordered so that $\text{FIRST}(h_i) \leq \text{FIRST}(h_{i+1})$.

The strategy in the new independent set algorithm is to begin by placing all vertices in V

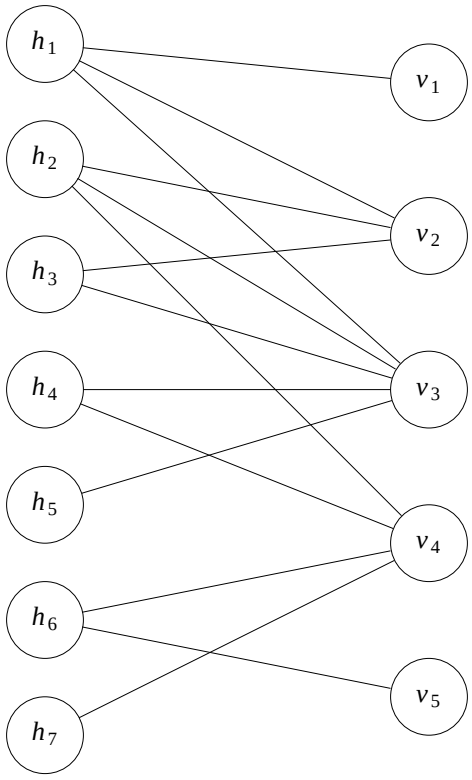
```
Procedure MaxMatch(G,M);
{G = ( H∪V, E) is a bipartite graph that is convex on V.  V = {v_1, ..., v_{n_V}} is ordered as required by
the convex property.  A maximum matching M is computed.}
begin
    M := ∅;
    for i := 1 to n_V do
        if NEB(v_i) ≠ ∅ then
        begin
            Let h_j ∈ H be such that LAST(h_j) = min_{h ∈ NEB(v_i)} {LAST(h)}
            M := M ∪ {(h_j, v_i)};
            Delete h_j, v_i, and incident edges from G;
        end
    end;{of if & for}
end;{of MaxMatch}
```

Figure 6 Maximum matching algorithm of [12].

into the MIS S that is being constructed. From Theorem 1 we know that there is an MIS that contains all free vertices and exactly one vertex from each pair of matched vertices in M. To construct such an indenpent set, we examine the free vertices that are also in H (all free vertices in V are already in S). They are examined in order (i.e. if h_i and h_j are free and $i < j$ then h_i is examined before h_j). When a free vertex h_i is examined, it is added to S. To ensure that the vertices in S are still independent we need to examine the V vertices in the range [FIRST(h_i), LAST(h_i)]. These must be eliminated from S. Further each vertex v_k in this range must be in the matching M as h_i is free and M is maximal. If v_k is matched to h_q in M then h_q is to be added to S and the V vertices in the range [FIRST(h_q), LAST(h_q)] eliminated from S. Note that LAST(h_q) ≤ LAST(h_i) as otherwise MaxMatch will match v_k to h_i rather than h_q.

Figure 7 gives an example bipartite graph that is convex on V. Consider the matching M = {(v_1, h_1), (v_2, h_3), (v_3, h_5), (v_4, h_2), (v_5, h_6)}. We start with S = {v_1, v_2, v_3, v_4, v_5} and Stack contains the single interval [-∞, -∞]. The following describes the progress of the above strategy on this example. A stack is used to keep track of processed intervals of V.

h_i	Stack	Actions on S	*low*	*high*	*l*	*r*



NEB(h_1) = $[v_1, v_3]$

NEB(h_2) = $[v_2, v_4]$

NEB(h_3) = $[v_2, v_3]$

NEB(h_4) = $[v_3, v_4]$

NEB(h_5) = $[v_3, v_3]$

NEB(h_6) = $[v_4, v_5]$

NEB(h_7) = $[v_4, v_4]$

$M = \{(v_1, h_1), (v_2, h_3), (v_3, h_5), (v_4, h_2), (v_5, h_6)\}$

$S = \{v_1, v_5, h_2, h_3, h_4, h_5, h_7\}$

Figure 7: Apply MaxInd1 to a convex (on V) bipartite graph.

h_4	$[-\infty, -\infty]$	Add h_4	v_3	v_4	$-\infty$	$-\infty$
		Delete v_4	v_3	v_4		
		Add h_2	v_2	v_4		
		Delete v_3	v_2	v_3		
		Add h_5	v_2	v_3		
		Delete v_2	v_2	v_2		
		Add h_3	v_2	v_2		
	$[-\infty, -\infty], [v_2, v_4]$		v_2	v_1		
h_7	$[-\infty, -\infty], [v_2, v_4]$	Add h_7	v_4	v_4	v_2	v_4

$[-\infty, -\infty]$ v_4 v_1 v_2 v_4

$[-\infty, -\infty]$, $[v_2, v_4]$

The strategy described above is implemented in procedure MaxInd1 (Figure 8). This procedure is due to [12]. It uses a stack to keep track of intervals $[l, r]$ of V that have already been processed. These intervals are in decreasing order as you go down the stack and are disjoint. The complexity of the algorithm is readily seen to be $O(n_H + n_V)$.

An examination of procedures MaxMatch and MaxInd1 reveals that an MIS of a convex bipartite graph $G = (H \cup V, E)$ can be found in $O(n_H + n_V)$ time if we can do the following in this much time:

a) Compute FIRST(h_i) and LAST(h_i), $1 \le i \le n_H$.

b) For each $v_i \in V$ with NEB(v_i) $\neq \varnothing$ compute $h_j \in NEB(v_i)$ such that LAST(h_j) = $\min_{h \in NEB(v_i)}$ {LAST(h)} (Procedure MaxMatch).

3 ~ O(kn) MNC Algorithm

3.1 Preliminaries

If the vertices of a polygon edge are both convex (concave) the edge is called a *support edge* (*reflex edge*). A polygon edge is a left, right, top, or bottom edge if the polygon interior is to its right, left, bottom, or top, respectively. The edges vw, za', and $d'e'$ of the polygon of Figure 2(a) are left support edges and edges xy and $b'c'$ are left reflex edges.

Theorem 4: Let P be a rectilinear hole-free polygon with horizontal and vertical inversion numbers k_H and k_V respectively. The following are true.

(a) k_H = number of right support edges plus number of left reflex edges.
= number of left support edges plus number of right reflex edges.

(b) k_V = number of top support edges plus number of bottom reflex edges.
= number of bottom support edges plus number of top reflex edges.

(c) For each of the four sides left, right, top, and bottom, the number of reflex edges is one

```
Procedure MaxInd1(G,M,S);
{G = ( H∪V, E) is a bipartite graph that is convex on V. M is a matching produced by procedure
MaxMatch. This procedure constructs an MIS S of G. It is assumed that the vertices in V are
ordered as required by the convexity property and that FIRST(h_i) ≤ FIRST(h_{i+1}), 1 ≤ i < n_H }
begin
    Initialize a stack so that it contains the single interval [l, r] = [-∞, -∞];
    S := V; {all vertices of V initially in S}
    for each free vertex h_i in order do
    begin
        Add h_i to S;
        if NEB(h_i) ≠ ∅ then
        begin
            low := FIRST(h_i); high := LAST(h_i);
            [l, r] := interval at top of stack;
            repeat
                if l ≤ high ≤ r then
                begin
                    delete [l, r] from top of stack;
                    high := l – 1; {bypass already processed interval}
                    low := min {l, low};
                    set [l, r] to new top of stack interval;
                end
                else
                begin
                    delete v_high from V;
                    if v_high is a matched vertex in M then
                    begin
                        Let h_q be the vertex it is matched to;
                        add h_q to S;
                        low := min {low, FIRST(h_q)};
                        {Note: high is not to be updated as LAST(h_q) ≤ LAST(h_i)}
                    end; {if matched}
                    high := high - 1;
                end; {disjoint from the interval at top of stack}
            until (high < low);
            Add [low, LAST(h_i)] to top of stack;
        end; {of if NEB(h_i) ≠ ∅}
    end; {of for each free vertex h_i}}
end;{of MaxInd1}
```

Figure 8 Maximum independent set algorithm of [12]

less than the number of support edges.

(d) k_H = number of vertical support edges minus one.

(e) k_V = number of horizontal support edges minus one.

Proof: (a) A walk around the polygon changes from right to left only at a right support edge and

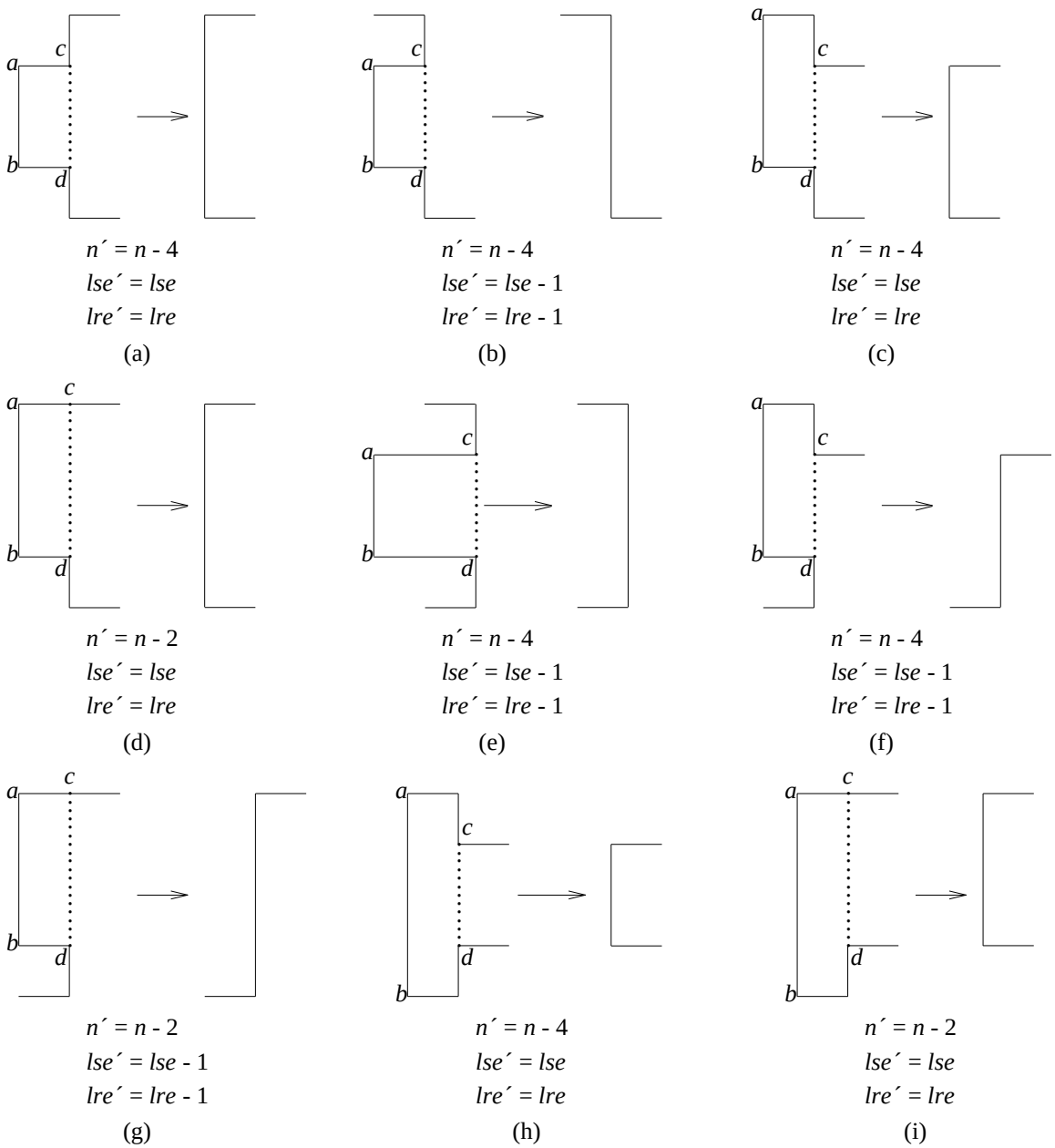
at a left reflex edge. Since the number of right to left changes equals the number of left to right changes, the total horizontal direction changes on a complete walk around the polygon is 2 * (number of right support edges plus number of left reflex edges). So k_H = number of right support edges plus number of left reflex edges. The equality k_H = number of left support edges plus number of right reflex edges is obtained in a similar way. (b) can be proved analogously.

We shall prove (c) by induction on the number n of vertices in P. Let *lse* and *lre* denote the number of left support edges and left reflex edges. *rse*, *bse*, *tse*, *rre*, *bre*, and *tre* are defined analogously. We explicitly show *lse* - *lre* = 1. The proof for the other three sides is similar.

For the induction base, we have $n = 4$. When $n = 4$, P has one left support edge and no left reflex edge. So, *lse* - *lre* = 1. Assume *lse* - *lre* = 1 for all hole-free rectilinear polygons with $n \leq m$ vertices where m is an arbitrary natural number ≥ 4. Let P be any rectilinear hole-free polygon with $n = m + 1$ vertices. P has at least one left support edge. Pick any one of P's left support edges and move this rightwards until this edge aligns with the first left or right edge encountered. Figure 9 gives the cases that can arise. For each case the primed quantities represent values for the polygon that remains following the elimination of the portion of the polygon crossed during the movement of the left support edge. In cases (j) through (t) *cmp´* gives the number of resulting hole-free polygons. This number is 1 for cases (a) through (i). In case (a) the number of vertices decreases by 4 as vertices *a*, *b*, *c*, and *d* are no longer polygon vertices following the movement of the support edge *ab* to the position *cd*. So, $n´ = n - 4$, *lse´* = *lse*, *lre´* = *lre*. From the induction hypothesis, *lse´* - *lre´* = 1. So, *lse* - *lre* = 1. As another example, consider case (j). The resulting $h + 1$ polygons have a total of n–4 vertices. Applying the induction hypothesis to each and summing, we get *lse´* – *lre´* = $h + 1$. So, *lse* – *lre* = *lse´* – h – *lre´* = 1. Using the induction hypothesis, we can show that for each of the remaing cases of Figure 9, *lse* - *lre* = 1.

For (d) we see from (a) and (c) that

$$\begin{aligned} k_H &= (rse + lre + lse + rre) / 2 \\ &= (2rse + 2lse - 2) / 2 \\ &= (rse + lse) - 1 \end{aligned}$$



ab = left support edge.

Figure 9: *lse* - *lre* = 1 (continued).

= number of vertical support edges minus one.

(e) is obtained in a similar way. □

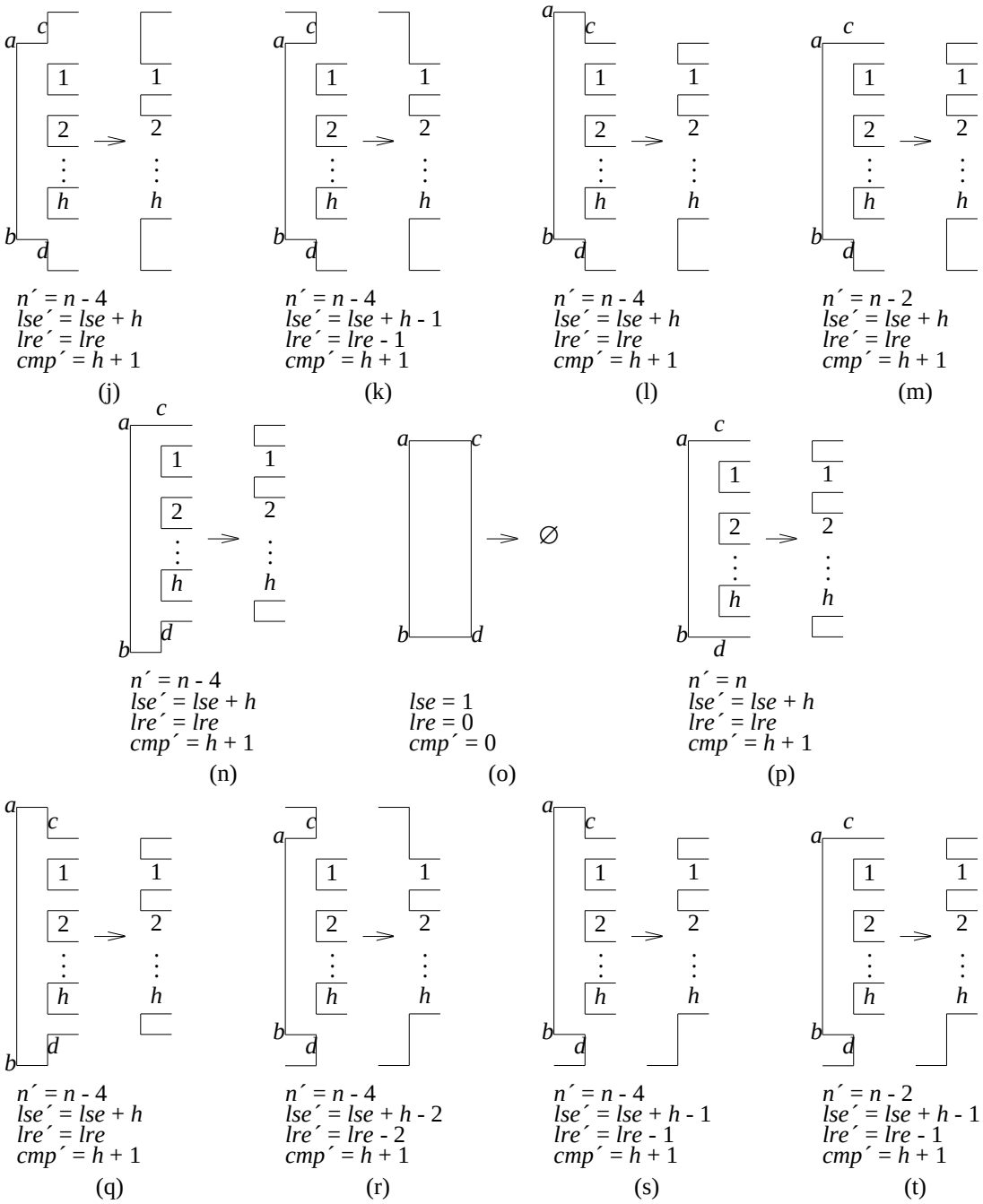


Figure 9: *lse* - *lre* = *cmp*.

3.2 $k = 1$

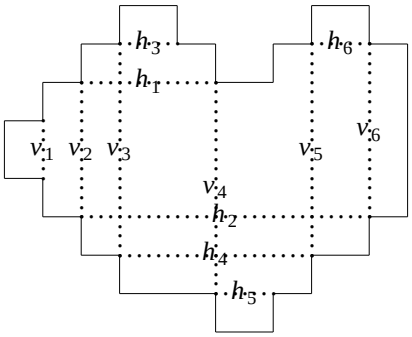
In this section we develop an O(n) algorithm to find the MNC of a rectilinear hole-free polygon with inversion number $k = 1$. We explicitly consider only the case $k = k_H = 1$. The case $k = k_V = 1$ is handled in an analogous manner.

Since every rectilinear polygon has at least one left support edge and one right support edge, it follows from Theorem 4(a) that when $k_H = 1$ the number of vertical reflex edges is zero. As a result the polygon cannot have two vertical chords with the same x-coordinate. Consequently there is a unique left to right ordering of the vertical chords. Let the vertical chords, in this ordering, be $v_1, v_2, ..., v_{n_V}$. This ordering may be obtained in O(n) time by starting at the two vertices of the single left support edge (Thoerem 4) of the polygon and moving rightwards. During this walk we match top vertices with bottom ones to see if a vertical chord can be drawn here. Since the polygon contains no holes and no vertical reflex edges, NEB(h_i) for every horizontal chord h_i is an interval of the vertical chords. The horizontal chords may be constructed and their FIRST and LAST values computed by making a left to right pass over the top and bottom boundaries of the polygon. During this pass the horizontal chords may also be ordered by their FIRST-value. The total time needed for this is O(n).

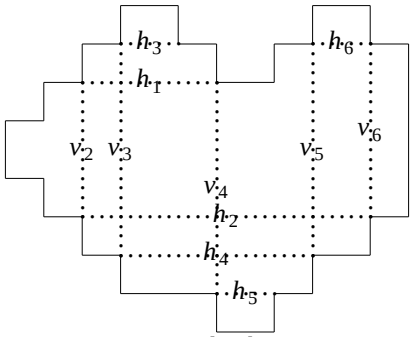
Procedure MaxMatch examines the vertical chords left to right. Suppose that the neighborhood, NEB, of a vertical chord is maintained as a doubly linked list sorted on the y coordinate. Since NEB(v_i) cannot contain two horizontal chords with the same y coordinate, this ordering is well defined. The doubly linked list can be initialized in O(1) time. As we go from one v_i to the next additions and deletions of horizontal chords is localized in the doubly linked list and can be done in time proportional to the number of affected chords. h_j is the horizontal chord in the doubly linked list that has the least right end point. Since the polygon has no right reflex edge, this must either be the horizontal chord with lowest or highest y coordinate. I.e., it is one of the chords at the two ends of the doubly linked list that represents NEB. So, h_j can be found in O(1) time and deleted from the list. Figure 10 illustrates this technique. The MIS is now found using procedure MaxInd1. From the above discussion it follows that the MNC of a hole-free rectilinear polygon with $k = 1$ can be found in O(n) time.

3.3 $k = 2$

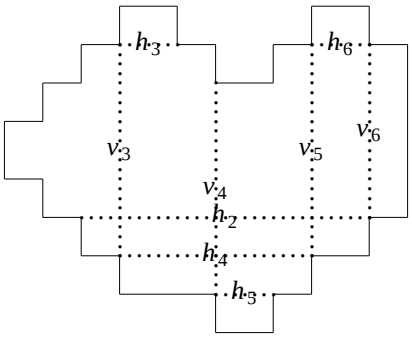
We explicitly consider only the case $k = k_H = 2$. The case $k = k_V = 2$ is similar. From Theorem 4(d) it follows that a polygon with $k_H = 2$ satisfies one of the following:



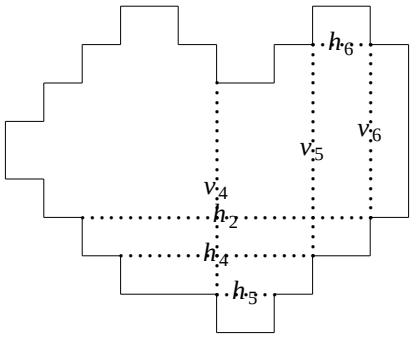
(a) NEB(v_1) = ∅; Unmatched;
M := ∅; F := {v_1}.



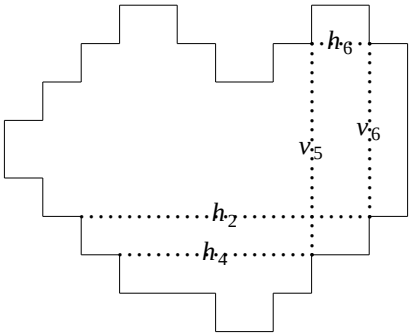
(b) NEB(v_2) = {h_1, h_2};
NEB(h_1) ⊆ NEB(h_2);
So, M := M + {(v_2, h_1)}.



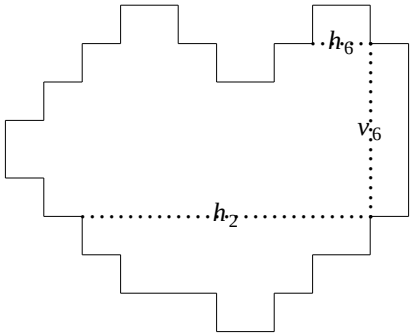
(c) NEB(v_3) = {h_2, h_3, h_4};
NEB(h_3) ⊆ NEB(h_2);
NEB(h_3) ⊆ NEB(h_4);
So, M := M + {(v_3, h_3)}.



(d) NEB(v_4) = {h_2, h_4, h_5};
NEB(h_5) ⊆ NEB(h_2);
NEB(h_5) ⊆ NEB(h_4);
So, M := M + {(v_4, h_5)}.



(e) NEB(v_5) = {h_2, h_4, h_6};
NEB(h_4) ⊆ NEB(h_2);
NEB(h_4) ⊆ NEB(h_6);
So, M := M + {(v_5, h_4)}.



(f) NEB(v_6) = {h_2, h_6};
NEB(h_2) ⊆ NEB(h_6);
So, M := M + {(v_6, h_2)}.
And, F := F + {h_6}.

Figure 10: An maximum matching example of a polygon with $k = k_H = 1$.

(a) $lse = 2$, $rse = 1$.

(b) $lse = 1$, $rse = 2$.

Since (b) is symmertric to (a), we consider only (a) further. From Theorem 4(a) it follows that *lre* = 1. If the single left reflex edge is extended in one direction until it meets a horizontal edge (see Figure 11) then the polygon is divided into two parts P_1 (left of the reflex extension) and P_2. Both P_1 and P_2 have $k = k_H = 1$. This partitioning into P_1 and P_2 may be done by beginning at the bottom of the left reflex edge and moving around the polygon until the walk first reaches the *x*-coordinate of this left reflex edge. Once P_1 and P_2 have been identified, their vertical chords can be numbered left to right as for the case $k = 1$ (Section 3.2). In each case, numbers are assigned beginning with the number 1. Horizontal chords that span both P_1 and P_2 will have two FIRST and LAST values associated with them; one being the first and last vertical chord of P_1 they intersect and the other being the first and last vertical chord of P_2 that is intersected. Since there are no right reflex edges, the FIRST and LAST values associated with the horizontal chords may be found in linear time as follows:

(1) Start at the left support edge of P_1 and proceed as for the case $k = 1$ (Section 3.2). Stop this algorithm when the right boundary of P_1 is reached. The horizontal segments that continue into P_2 are saved.

(2) Start at the left support edge of P_2 and proceed as for the case $k = 1$. When the left reflex edge is reached add in the saved horizontal segments of P_1 and continue.

A maximum matching of horizontal and vertical chords is obtained as follows:

(1) Begin at the leftmost vertical chord of P_1 and proceed left to right as for the case $k = 1$ (Section 3.2). A doubly linked list of intersecting horizontal chords ordered by *y*-coordinate is maintained as in the case $k = 1$. When each vertical chord is examined in this order, it is a matchable chord. This follows from the observation that since *P* has no right reflex edge, the neighborhood of the intersecting horizontal chord with smallest right end point is contained in the neighborhood of every other intersecting horizontal chord. Further, the absence of a right reflex edge implies that the intersecting horizontal chord with the smallest right end point is either at the front or the end of the doubly linked list (i.e., it is either the one with the highest or the lowest *y*-coordinate). This matching of vertical chords of P_1

to horizontal chords terminates when the right boundary of P_1 is reached. The doubly linked list of horizontal segments is saved.

(2) Begin with the leftmost vertical chord of P_2 and proceed left to right matching horizontal and vertical chords as in (1). Stop when the left to right sweep reaches the left reflex edge. To the doubly linked list of horizontal segments append the doubly linked list saved in (1) and continue the left to right matching sweep.

The above two step process results in a maximum matching as when a vertical chord is matched it is a matchable chord (Theorem 2).

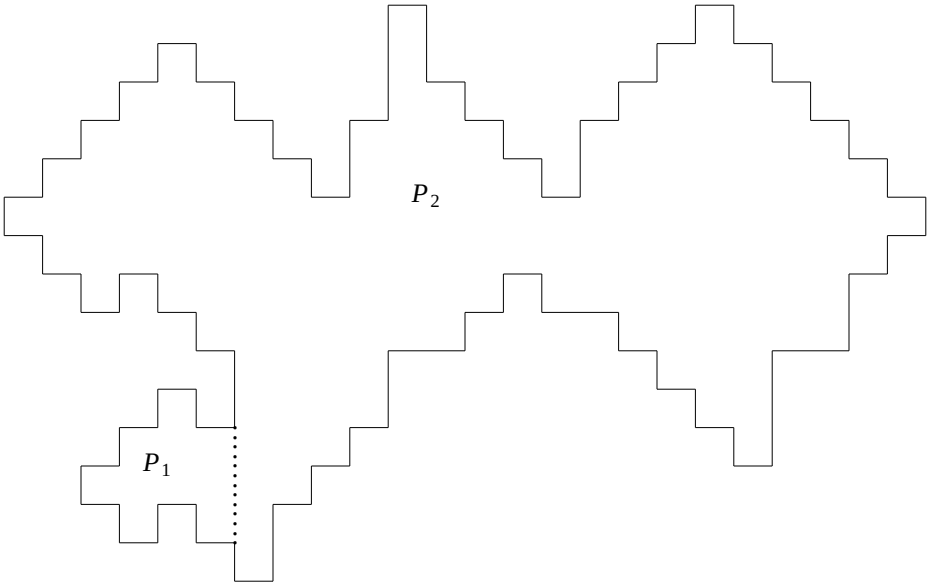


Figure 11: A polygon with $k = k_H = 2$.

To compute the MIS we use the strategy used in the case $k = 1$. I.e., we begin with all vertices of V (both in P_1 and P_2) in the independent set and then process the free horizontal chords. The stack scheme of MaxInd1 is replaced by union-find structures [14] for each of P_1 and P_2. For each of P_1 and P_2 we maintain a set of equivalence classes of vertical chords in P_1 and P_2, respectively. Each equivalence class represents an interval of processed vertical chords. A free horizontal chord that is in both P_1 and P_2 needs to be processed in both polygons. The complexity of the resulting scheme is $O(n, \alpha(n,n))$ where α is the inverse of the Ackermann's function

[14].

3.4 $k = 3$

From Theorem 4 we see that one of the following must be true:

(a) $lse = 3$, $rse = 1$.

(b) $lse = rse = 2$.

(c) $lse = 1$, $rse = 3$.

Since (a) and (c) are symmetric we need only consider (a) and (b). Figure 12(a) gives an example for $lse = 3$ and $rse = 1$ while Figures 12 (b), (c), (d), and (e) are examples for the case $lse = rse = 2$. When $lse = 3$ and $rse = 1$ the polygon has no right reflex edge and has two left reflex edges (Theorem 4(a)). By extending the two left reflex edges at one end each we can decompose the polygon into three polygons with $k = 1$. Since there is no right reflex edge we can find the maximum matching by extending the strategy of Section 3.3 to the case when P is partitioned into three polygons with $k = 1$ each. We first match the chords in P_1 and then continue to P_2. The matching in P_2 commences at its leftmost vertical chord. When the processing reaches the right boundary of P_1, the doubly linked list saved from P_1 is appended and we continue till the right boundary of P_2 is reached. Now we start matching in P_3. The processing begins at the leftmost vertical chord in P_3. When the right boundary of P_2 is reached, the doubly linked list saved at the end of the processing of P_2 is appended and we continue up to the rightmost vertical chord in P_3. The correctness of this scheme is established in the same way as we established the correctness of the $k = 2$ scheme.

When $lse = rse = 2$ the polygon has one right reflex edge and one left reflex edge. The partitioning into P_1, P_2, and P_3 is done as follows:

(1) Determine the number of vertical support edges encountered in a walk that begins at the top of the right reflex edge and ends at the top of the left reflex edge. The walk follows the polygon edges but does not go over the reflex edges. This latter restriction uniquely specifies the walk.

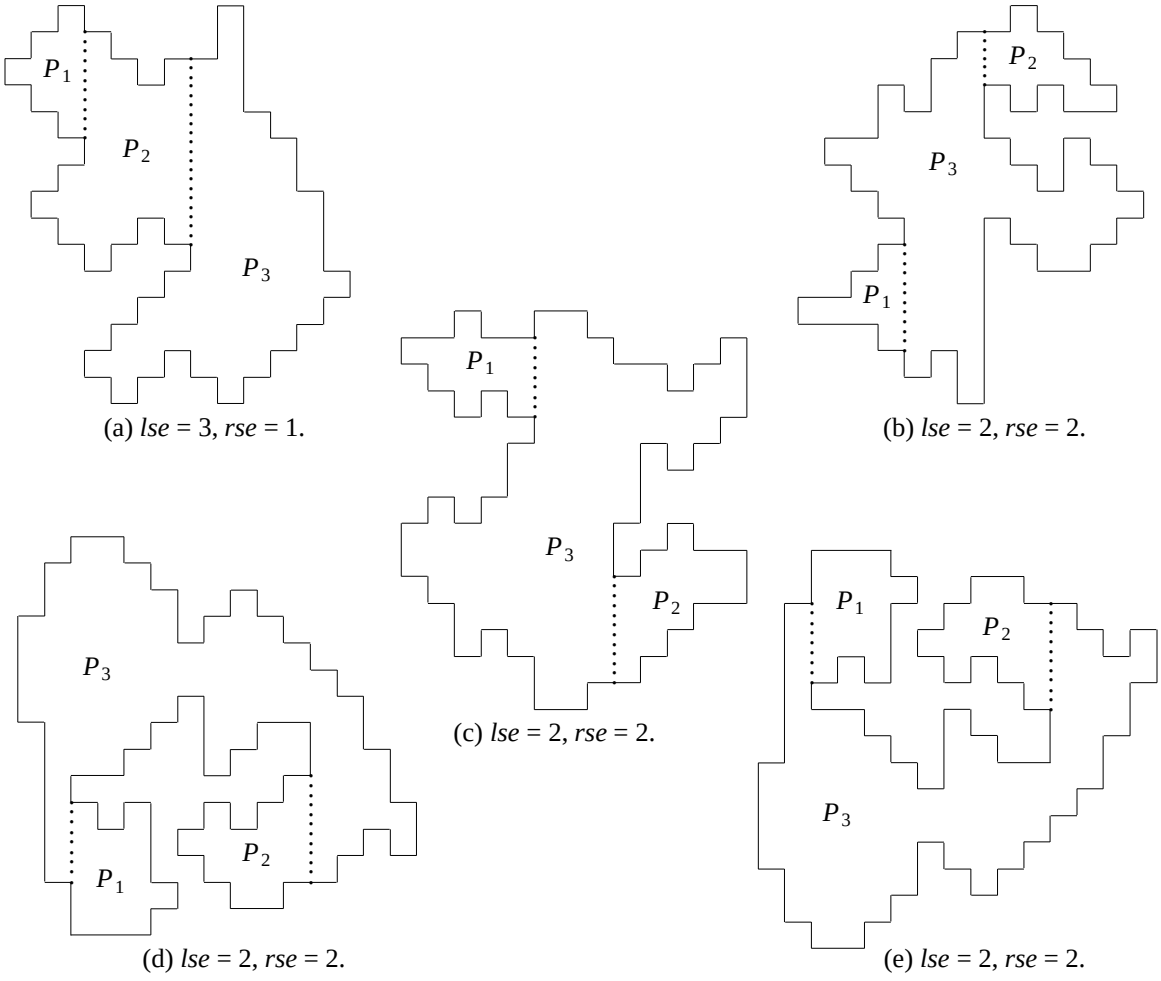


Figure 12: Example polygons with $k = k_H = 3$.

(2) The number of vertical support edges encountered in the above walk is either 0, 2, or 4. If the number is 0, then extend both reflex edges downward (Figure 12(d)). If the number is 2, then extend the right reflex edge upward and the left reflex edge downward in case the bottom of the right reflex edge is at a higher *y*-coordinate than the top of the left reflex edge (Figure 12(b)); otherwise extend the right reflex edge downward and the left reflex edge upward (Figure 12(c)). If the number is 4, then extend both reflex edges upward (Figure 12(e)).

A maximum matching of vertical chords to horizontal chords is obtained as follows:

(1) [Number of vertical support edges in walk is 0 (Figure 12(d))].

(a) Match chords in P_1 using the strategy of Section 3.2. However, P_1 is processed right to left. Stop when the left boundary of P_1 is reached. Horizontal chords of P_1 that extend into P_3 are now processed. If the chord was matched to a vertical chord of P_1, it is marked as matched. If it was not matched, its right end point is changed to be the left boundary of P_1.

(b) Match chords in P_2 left to right using the strategy of Section 3.2. Stop when the right boundary of P_2 is reached and save the doubly linked list of horizontal chords.

(c) Match chords in P_3 left to right. Already matched horizontal chords are ignored from the processing. When the right boundary of P_2 is reached, the doubly linked list saved when the processing of P_2 finished is appended to the current doubly linked list.

The correctness of the above matching strategy follows from the following observations:

(a) When a vertical chord is examined, it is a matchable chord.

(b) If this vertical chord is in P_1, then it may be matched to the horizontal intersecting chord with maximum left end point. This is either the bottom most or topmost intersecting chord.

(c) If it is a vertical chord of P_2, it can be matched to the horizontal intersecting chord with the smallest right end point.

(d) If it is a P_3 vertical chord, it is to be matched to the horizontal intersecting chord that is as yet unmatched and has smallest right end point. Since all vertical chords of P_1 that could be matched to a horizontal chord have already been matched, the right end points of unmatched horizontal chords that extend into P_1 may be considered to be the left boundary of P_1. This does not affect the set of free vertical chords each such horizontal chord intersects.

(2) [Number of vertical support edges in walk is 2 and the bottom of the right reflex edge is above the top of the left reflex edge, (Figure 12(b))].

(a) Match chords in P_1, left to right, using the strategy of Section 3.2. Stop when the right boundary of P_1 is reached and save the doubly linked list of free horizontal chords that extend into P_3.

(b) Match the chords in P_2, right to left, using the strategy of Section 3.2. Stop when the left boundary of P_2 is reached. Mark the matched horizontal chords of P_2 that extend into P_3 as matched. Change the right end points of the unmatched horizontal chords of P_2 that extend into P_3 to be the left boundary of P_2.

(c) Match the chords in P_3 left to right. When the right boundary of P_1 is reached, append the saved doubly linked list.

The fact that this strategy results in a maximum matching may be established in a manner analagous to that used for case (1). The strategies for the remaining two cases (Figures 12(c) and (e)) are similar.

The MIS is now found in $O(n, \alpha(n,n))$ time using MaxInd1 and union-find structures for each of P_1, P_2, P_3.

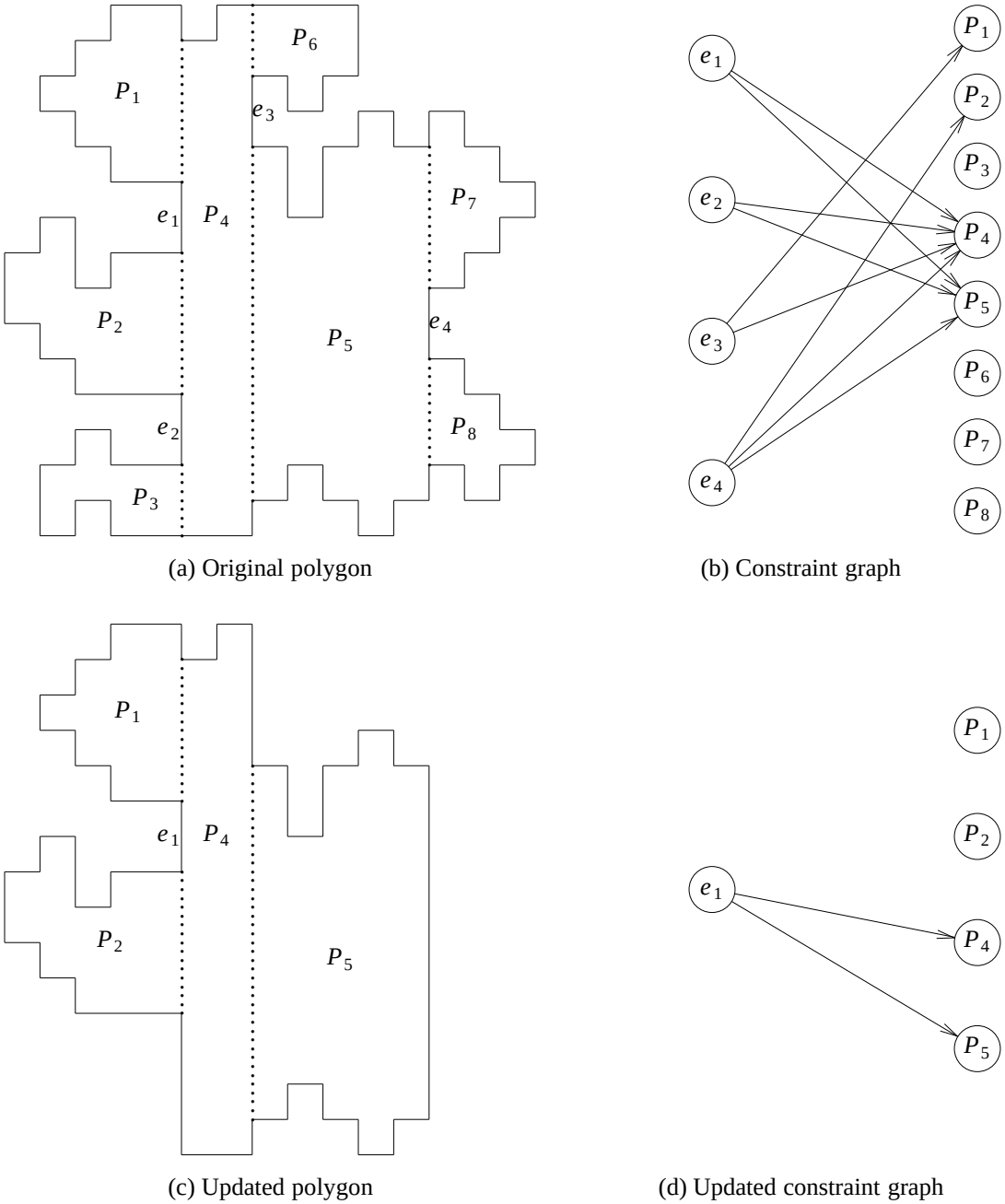
3.5 $k > 3$

We explicitly consider only the case $k = k_H$. The strategy is to decompose the polygon into $2k$ subpolygons with $k_H = 1$ and then use the matching strategy of Section 3.2 to each of these in a suitable order. The decomposition is easily done by extending the vertical reflex edges in both directions (Figure 13(a)). This takes $O(n \log k)$ time if a range search tree [15] is used to store the end points of the at most k - 1 vertical reflex edges. The resulting polygons have no left or right reflex edges. From Theorem 4(c) it follows that each polygon has $lse = rse = 1$. From Theorem 4(d) it follows that $k_H = 1$ for each polygon.

In order to apply the strategy of Section 3.2 (or its modification to a right to left sweep) to a subpolygon, the following must be true at the time the matching in the subpolygon begins:

(1) The subpolygon should contain no vertical reflex edge.

(2) Either all subpolygons that abut it on the left or all that abut it on the right must have been



(a) Original polygon (b) Constraint graph

(c) Updated polygon (d) Updated constraint graph

Figure 13: Example polygon with $k = k_H > 3$.

processed. In case the former is true, the subpolygon is processed left to right; otherwise it is processed right to left.

(3) The subpolygon should not contain two unmatched horizontal chords such that one passes

on one side of a vertical reflex edge and the other passes on the other side of this reflex edge. For this test, the current end points of chords are used. Recall from Section 3.4 that when a subpolygon is processed right to left the right end points of some chords are updated. Similarly, we assume, that when a subpolygon is processed left to right, the left end points are updated to become the right boundary of the subpolygon.

Note that since the subpolygons are constructed by extending all vertical reflex edges, all of them satisfy condition (1). So we need only be concerned with conditions (2) and (3). For condition (2), we can maintain, with each subpolygon, a count of the number of unprocessed subpolygons that abut it on the left and the number that abut it on the right. Also, with each subpolygon we maintain a list of the subpolygons it abuts on the left as well as on the right. When a subpolygon is processed, this list is traversed and the left abut and right abut counts of the subpolygons on this list are decremented. For condition (3), we maintain a *constraint graph* CG. This is a bipartite graph with one vertex for each subpolygon and one for each vertical reflex edge. There is an edge between a subpolygon vertex and a reflex edge vertex iff both ends of the reflex edge can be joined to the interior of the subpolygon by a horizontal line that is wholly within the polygon. Figure 13(b) gives the constraint graph for Figure 13(a).

Subpolygon vertices with degree zero in the constraint graph and which satisfy condition (2) above can be processed first in any order. So these are added to the front of the order being constructed. These subpolygons may be deleted from the original polygon to get the polygon of Figure 13(c). Since the deletion of each subpolygon reduces the number of vertical reflex edges by 1, the new polygon has fewer reflex edges than the original one. A new constraint graph for this polygon is constructed (Figure 13(d)) and the subpolygons with degree zero that also satisfy condition (2) above are added to the order being constructed. So, so long as the constraint graph has a subpolygon vertex of degree zero that also satisfies condition (2), we can repeat this process to obtain the desired order. For the example of Figure 13(a) we obtain the order P_3, P_6, P_7, P_8, P_1, P_2, P_4, and P_5. The following theorem shows that every constraint graph has at least one subpolygon vertex whose degree is zero and for which its current left abut or right abut count is

also zero.

Theorem 5: The constraint graph of every rectilinear hole-free polygon (whether original or updated) has at least one subpolygon vertex with degree zero and such that the corresponding subpolygon has a left or right about count of zero.

Proof: Without loss of generality, assume that $lse \geq rse$. Since $rse = rre + 1$ (Theorem 4(c)), it follows that $lse > rre$. The number of subpolygons one of whose sides is a left support edge of the polygon is lse. Each right reflex edge can be connected to at most one of these in the constraint graph. Since $lse > rre$, at least one of these left support edge subpolygon vertices is not connected to a right reflex edge vertex in the constraint graph. Further, no left support edge subpolygon vertex can be connected to a left reflex edge vertex. Hence there is at least one left support edge subpolygon vertex with degree zero. Every subpolygon, constructed as described above, that has a left support edge has a left abut count of zero. □

Using the above strategy and appropriate data structures a maximum matching can be found in $O(kn)$ time. The MIS can be found in $O(kn\alpha(n, n))$ time using MaxInd1 and union-find structures.

4 O($n\log k$) Algorithm

This algorithm is an adaptation of the $O(n\log\log n)$ MNC algorithm of [8]. This latter algorithm has the following steps:

Step 1: Decompose the rectilinear polygon P into vertically convex subpolygons. This decomposition is done in such a way that a maximum matching of chords can be found by processing the subpolygons in the order in which they are created.

Step 2: Process the subpolygons in the order created to obtain a maximum matching. Free chords from each subpolygon are added to a queue in the order in which they are determined to be free by the maximum matching algorithm.

Step 3: Obtain an MIS of chords by processing the maximum matching and free chord queue obtained in Step 2.

Step 4: Draw the independent chords in the independent set of Step 3 to get a set of subpolygons. Draw all vertical chords in the resulting subpolygons to get rectangles.

Step 3 can be done in O(n) time [8] and Step 4 takes O($n\log k$) time. We show how steps 1 and 2 can be done in O($n\log k$) time. We consider the case $k = k_V$. The case $k = k_H$ is similar. The basic strategy is to start at any horizontal support edge and advance into the polygon (i.e., if we start at a bottom support edge, then we move upwards and if we start at a top support edge, we move downwards) until we reach either a vertical or horizontal reflex edge. This is done by moving along the contour of the polygon beginning at the horizontal support edge. The moving is done in both directions keeping the y-coordinate of the position in both directions the same (this is the same as using a horizontal scan line with endpoints on the polygon contour identified by the above walk). To determine if a horizontal reflex edge is reached, we maintain a range search tree of horizontal reflex edges. Since the number of such edges is k - 1, the time to determine if such an edge is reached in O($\log k$). For vertical reflex edges, we simply see if the current contour edge is a vertical reflex edge (this test cannot be extended to horizontal reflex edges as the next such edge to be encountered may not lie on the portion of the contour being traversed currently). When a vertical reflex edge is reached, it may be extended downwards if we start at a bottom support edge or upwards if we start at a top support edge to obtain a vertical convex subpolygon. The situation is more complex when a horizontal reflex edge is reached. We go through a detailed example to illustrate the various cases that arise. A high level description of the decomposition algorithm is given in Figure 14. In this algorithm $\overline{next}$ denotes the opposite of *next*. So, if *next* = top, $\overline{next}$ = bottom and if *next* = left, $\overline{next}$ = right. *SP* is a list of the subpolygons created. The subpolygons appear in this list in the order created.

Consider the polygon of Figure 15(a). Suppose we begin at the horizontal support edge s_1s_2. Since this is a bottom support edge, we set *next* = top and move a horizontal scan line upwards from s_1s_2. Vertices a, b, and c, the bottom ends of vertical reflex edges, are encountered in order and subpolygons P_1, P_2, and P_3 are added to *SP* in this order. Next, the scan line reaches a horizontal reflex edge. We stop processing and resume at another horizontal support edge. Fig-

```
Procedure Decompose(P,SP);
{Decompose the original polygon P with k = k_V into SP, a list of vertical convex subpolygons.}
begin
    SP := ∅;
    repeat
        Pick a horizontal support edge hse;
        Scanline := hse;
        if hse is a bottom edge then next := top else next := bottom;
        repeat
            Move Scanline along the boundary to the next side until a horizontal edge is
            encountered;
            if Scanline encloses a horizontal support edge then
            begin
                Save the last support edge and the remaining polygon P;
                Stop;
            end
            else if Scanline encloses a next reflex edge then
                if only one region X on the next side is unvisited then
                    shift Scanline to the entrance of X and continue
                else
                    Mark all of the next reflex edges visited on this side and exit to outer
                    loop;
            else if Scanline encloses a next reflex edge then
                if only one region Y on the next side is unvisited then
                    turn around to the entrance of Y and reverse next
                else
                    Mark the current region as visited and exit to the outer loop;
            else
            begin
                if a left reflex edge is encountered then
                    Make a vertical cut on the left side to get P_left and add to the end of SP;
                if a right reflex edge is encountered then
                    Make a vertical cut on the right side to get P_right and add to the end of
                    SP;
            end;
        until false;
    until false;
end;{of Decompose}
```

Figure 14

ure 15(b) shows the current situation. Subpolygons P_1, P_2, and P_3 have been eliminated and the hatched area indicates a region of the polygon that has been scaned. Suppose we pick the bottom support edge s_3s_4. *next* is set to top and we advance upwards. The vertical reflex edges with vertices *d*, *e*, and *f* at the bottom ends are found. As a result, subpolygons P_4, P_5, and P_6 are added to the queue *SP*. Processing stops when the next horizontal reflex edge is reached. It resumes at a

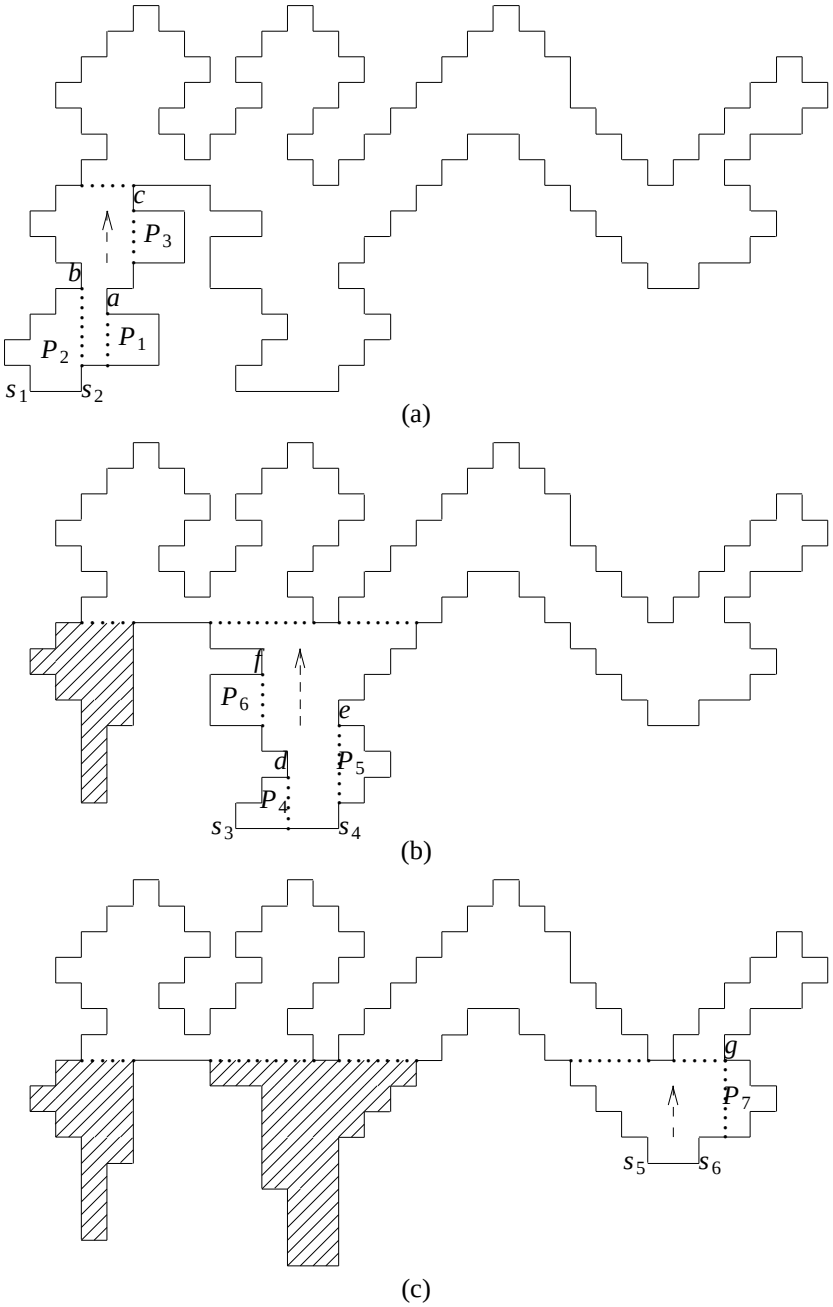


Figure 15: An exhibition of algorithm Decompose (continued).

new horizontal support edege.

Going on to Figure 15(c), the horizontal support edge s_5s_6 is picked and subpolygon P_7 is appended to *SP*. Next, we pick the top support edge s_7s_8 and set *next* = bottom (Figure 15(d)). No subpolygon is generated in this area. Then, another top support edge s_9s_{10} is considered and we

move down until a horizontal reflex edge is encountered (Figure 15(e)). Then, we turn the direction 3 times, down to up, up to down, and down to up, respectively (Figure 15(f)). No subpolygon is added to *SP* as no vertical reflex edge is encountered. Finally, the top support edge $s_{11}s_{12}$ is handled (Figure 15(g)). The subpolygons P_8 and P_9 are added to *SP* before a horizontal

reflex is found.

The subpolygons P_{10} and P_{11} are appended to *SP* when we turn around from going down to up and complete the whole procedure Decompose (Figure 15(h)). The remaining vertical convex polygon P_{12} is added to *SP* (Figure 15 (i)).

Step 2 can actually be done concurrently with Step 1. The strategy of Section 3.2 can be used to obtain a maximum matching and the queue of free chords in $O(n)$ time. Step 3 is now implemented as in [8] and Step 4 is the same as that used for the algorithm of Section 3.

5 Experimental Results

We programmed three algorithms for the MNC problem. These are:

1. $O(n \log k)$ ... The algorithm of Section 4.
2. $\sim O(kn)$ ... This uses the algorithms described in Sections 3.2, 3.3, and 3.4 for $k = 1$, 2, and 3, respectively. For $k > 3$, it uses the decomposition scheme of Section 4 and the independent set strategy of Section 3.5. We did not use the decomposition scheme of Section 3.5 as preliminary tests indicated it was slower than that of Section 4.
3. $O(n^{1.5} \log n)$ ... This is the algorithm of [1].

All algorithms were programmed in Pascal and run on an Apollo DN3500 workstation. Tables 1-5 give the average run times, in seconds, of the three programs for the case of randomly generated rectilinear polygons with $k = 1$, 2, 3, 5, and 10, respectively. (Rectilinear polygons were generated randomly and then classified according to their k value.) The column labeled *size* gives the number of vertices in the polygon. For each value of *size*, 50 random polygons were generated. The columns labeled improve (%) give the amount by which the run time of the $O(n^{1.5} \log n)$ algorithm exceeds this algorithm's run time. As can be seen, for any fixed k, the $O(n \log k)$ and $\sim O(kn)$ programs outperform the $O(n^{1.5} \log n)$ program for suitably large polygons. The percentage improvement in run time increases as the polygon size increases. For $k = 1$, and 2, the $\sim O(kn)$ program is faster than the $O(n \log k)$ program. For $k = 3$, these two programs are quite competitive and for $k > 3$ the $O(n \log k)$ program is superior.

Table 6 gives the measured run time for the UNISYS polygon set. The $O(n \log k)$ and $\sim O(kn)$ programs outperformed the $O(n^{1.5} \log n)$ one for polygons with no more than 40 vertices. The $O(n^{1.5} \log n)$ program was slightly faster for the larger polygons in this test set.

size	$O(n\log k)$		$\sim O(nk)$		$O(n^{1.5}\log n)$
	T_{avg}	*Improve*(%)	T_{avg}	*Improve*(%)	T_{avg}
≈ 20	2.52	70.63	2.28	88.60	4.30
≈ 30	3.40	88.24	2.95	116.95	6.40
≈ 40	4.31	122.74	3.82	151.31	9.60
≈ 50	5.59	119.32	4.92	149.19	12.26
≈ 60	6.28	130.25	5.56	160.07	14.46
≈ 70	7.48	152.54	6.59	186.65	18.89
≈ 80	8.74	180.66	7.65	220.65	24.53
≈ 90	9.89	174.32	8.70	211.84	27.13
≈100	10.28	163.04	9.10	197.14	27.04
≈200	21.40	217.10	18.11	274.71	67.86
≈300	31.99	252.80	27.42	311.60	112.86

Table 1 $k = 1$

size	$O(n\log k)$		$\sim O(nk)$		$O(n^{1.5}\log n)$
	T_{avg}	*Improve*(%)	T_{avg}	*Improve*(%)	T_{avg}
≈ 20	2.63	30.04	2.76	23.91	3.42
≈ 30	3.40	65.00	3.35	67.46	5.61
≈ 40	4.69	93.39	4.52	100.66	9.07
≈ 50	5.88	139.80	5.63	150.44	14.10
≈ 60	6.86	138.92	6.65	146.47	16.39
≈ 70	8.01	156.80	7.52	173.54	20.57
≈ 80	8.95	159.22	8.45	174.56	23.20
≈ 90	10.12	119.66	9.79	127.07	22.23
≈100	11.04	145.65	10.79	151.34	27.12
≈200	23.22	166.84	22.66	173.43	61.96
≈300	33.14	217.65	31.47	234.51	105.27

Table 2 $k = 2$

6 Conclusions

We have developed two algorithms to obtain the MNC of rectilinear polygons with small k. Both are significantly faster than the $O(n^{1.5}\log n)$ algorithm of [1] on polygons whose size is large relative to k. For $k = 1$ and 2 the $\sim O(kn)$ algorithm is recommended while for $k > 2$ the $O(n\log k)$ algorithm is recommended.

7 References

[1] H. Imai and T. Asano, "Efficient Algorithms for Geometric Graph Search Problems", SIAM

size	$O(n\log k)$		$\sim O(nk)$		$O(n^{1.5}\log n)$
	T_{avg}	*Improve*(%)	T_{avg}	*Improve*(%)	T_{avg}
≈ 20	2.95	-4.75	3.21	-12.46	2.81
≈ 30	3.59	40.39	3.62	39.23	5.04
≈ 40	4.90	75.71	4.82	78.63	8.61
≈ 50	6.07	95.39	5.93	100.00	11.86
≈ 60	7.29	112.21	7.07	118.81	15.47
≈ 70	8.10	142.59	7.78	152.57	19.65
≈ 80	9.53	147.32	8.93	163.94	23.57
≈ 90	10.35	121.16	10.92	109.62	22.89
≈100	11.33	131.86	11.99	119.10	26.27
≈200	23.90	176.03	23.47	181.08	65.97
≈300	34.84	177.44	34.46	180.50	96.66

Table 3 $k = 3$

size	$O(n\log k)$		$\sim O(nk)$		$O(n^{1.5}\log n)$
	T_{avg}	*Improve*(%)	T_{avg}	*Improve*(%)	T_{avg}
≈ 20	6.39	-52.11	6.62	-53.78	3.06
≈ 30	7.29	-35.67	7.34	-36.10	4.69
≈ 40	8.95	-12.07	8.97	-12.26	7.87
≈ 50	10.51	12.65	10.55	12.23	11.84
≈ 60	12.21	21.62	12.31	20.63	14.85
≈ 70	13.45	29.29	13.53	28.53	17.39
≈ 80	14.94	43.51	15.07	42.27	21.44
≈ 90	16.37	52.54	16.60	50.42	24.97
≈100	18.34	58.89	18.61	56.58	29.14
≈200	32.94	96.42	35.06	84.54	64.70
≈300	46.52	99.87	50.03	85.85	92.98

Table 4 $k = 5$

size	$O(n\log k)$		$\sim O(nk)$		$O(n^{1.5}\log n)$
	T_{avg}	*Improve*(%)	T_{avg}	*Improve*(%)	T_{avg}
≈ 60	16.29	-13.26	17.09	-17.32	14.13
≈ 70	17.01	-8.70	17.65	-12.01	15.53
≈ 80	18.56	4.20	19.34	0.00	19.34
≈ 90	20.37	15.02	21.36	9.69	23.43
≈100	21.85	19.59	22.98	13.71	26.13
≈200	36.85	62.33	39.50	51.44	59.82
≈300	37.40	98.26	40.39	83.59	74.15

Table 5 $k = 10$

size	*polygon#*	$O(n\log k)$		$\sim O(nk)$		$O(n^{1.5}\log n)$
		T_{avg}	*Improve*(%)	T_{avg}	*Improve*(%)	T_{avg}
≤ 10	1739	0.84	1.19	0.78	8.97	0.85
≤ 20	926	1.27	5.51	1.25	7.20	1.34
≤ 30	84	2.78	56.83	2.79	56.27	4.36
≤ 40	27	4.51	144.35	4.42	149.32	11.02
≤ 90	52	7.83	-0.89	7.99	-2.88	7.76

Table 6 Times for UNISYS polygon set

J. Computing, Vol 15, No 2, May 1986, pp. 478-494.

[2] E. Lodi, F. Luccio, C. Mugnai and L. Pagli, "On Two-dimensional Data Organization I", Fundamenta Informaticae 2 (1979), pp. 211-226.

[3] K. D. Gourley and D. M. Green, "A Polygon-to-Rectangle Conversion Algorithm", IEEE Computer Graphics, Vol 3, No. 1, Jan/Feb 1983, pp. 31-36.

[4] J. K. Ousterhout, "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools", IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol CAD-3, No. 1, Jan. 1984, pp. 87 - 100.

[5] J. P. Cohoon, "Fast Channel Graph Construction", Department of Computer Science, University of Virginia.

[6] S. Nahar and S. Sahni, "A Time and Space Efficient Net Extractor", 23rd Design Automation Conference, 1986, pp. 411-417.

[7] T. Ohtsuki, "Minimum Dissection of Rectilinear Regions", Proceedings 1982 International Symposium on Circuits And Systems (ISCAS), 1982, pp. 1210-1213.

[8] W. T. Liou, J. J. M. Tan, and R. C. T. Lee, "Minimum Partitioning Simple Rectilinear Polygons in O($n$loglog$n$) time", Proceedings of the Fifth annual symposium on Computational Geometry, pp344-353, 1989.

[9] R. E. Tarjan and C. J. Van Wyk, "Triangulating a simple polygon", SIAM Jr on Computing, 17, 1, 1988, pp. 143-178.

[10] S. Nahar and S. Sahni, "A Fast Algorithm for Polygon Decomposition", IEEE Trans. on

Computer Aided Design of Integrated Circuits and Systems, Vol. CAD-7, No. 4, April 1988, pp. 478 - 483.

[11] E. L. Lawler, "Combinatiorial Optimization: Networks and Matroids", Holt, Rinehart and Winston, New York, 1976.

[12] W. Lipski, Jr. and F. P. Preparata, "Efficient Algorithms For Finding Maximum Matching In Convex Bipartite Graphs And Related Problems", Acta Informatica 15, 1981, pp.329-346.

[13] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ Algorithm for Maximum matchings in bipartite graphs", SIAM J. Comput. 2, 1973, pp. 225-231.

[14] E. Horowitz and S. Sahni, "Fundamentals of Data Structures In Pascal", Computer Science Press, Inc., Second Edition, Maryland, 1986

[15] F. P. Preparata and M. I. Shamos, "Computational Geometry", Springer-Verlag Inc., New York, 1985.