Reassembling Polygons from Edges

Richard I. Hartley

CS-85-04

April, 1985

Reassembling Polygons from Edges

Richard I. Hartley

Visitor to VLSI Group,
Department of Electrical Engineering,
University of Waterloo,
Waterloo, Ontario N2L 3G1
CANADA

ABSTRACT

Many geometrical algorithms accept as input a set of polygons specified by their sequence of vertices and give as output a set of polygons which result from some operation on the input polygons. For instance, the output may represent the contour of the union of the polygons. In many cases a plane-sweep algorithm is used for such a problem. Typically such an algorithm will output the edges in the order that they would be encountered by a sweep line. Thus, one is left with the problem of determining which polygon each edge belongs to and ordering the edges in their natural cyclic order around each polygon. This is the problem considered in this paper, where it will be referred to as reassembling the set of edges. Although it is relatively easy to reassemble a set of edges of polygons in a single sweep, a naive approach will use $O(N)$ space where N is the number of edges. This paper gives a space-efficient algorithm for doing the reassembly. It involves a forward and a backward sweep over the data.

1. Introduction

There have been a number of papers which have been concerned with doing various sorts of logical operations on sets of polygons. For instance, [OW] look at the contour problem for sets of polygons, [OWW] consider the more general problem of arbitrary boolean operations, whereas [NP] consider intersection problems. The output from such algorithms is typically a list of edges given in the order in which they would encounter a sweep line, without any indication of whether two edges belong to the same polygon, or if they do, what the order of the edges around the individual polygons might be. The problem of assigning to each edge a *polygon number* designating which polygon it is in and a *sequence number* giving its order around the perimeter of the polygon is the subject of this paper. Such a process will be known as *reassembly*. If one is not too concerned by space usage, it is indeed relatively easy to reassemble a set of polygon edges using a plane sweep algorithm. As the plane sweep progresses, edges are chained together in lists representing their sequential order around the perimeter of each polygon. When the polygon closes on itself, completing the circle, the edges may be output in order by running down the list. Thus, one can solve the reassembly problem by maintaining a certain amount of past history. This will, however involve considerable memory requirement, for in a typical application, that of VLSI design, the power lines may be polygons with $O(N)$ edges where N is the total number of edges in the entire layout.

This problem was recognised by Szymanski and Van Wyk [SvW] who were interested in space efficient extraction and electrical connectivity algorithms. They gave an algorithm which will take an unsorted set of edges and determine which polygon each edge belongs to. This is half the task of reassembly. Szymanski and Van Wyk used a double sweep to achieve this numbering. In the first sweep, a temporary file is built which is read backwards during a second sweep, and the required output is generated. The point of their algorithm is that only $O(C)$ space is required, where C is the maximum number of edges crossed by a sweep-line in the plane. This is achieved by keeping no past history of the sweep in memory. All the required information is incorporated in the temporary file. Since for a standard design, C will be $O(\sqrt{N})$, this will represent a considerable saving of space. The algorithm given here also uses a forward and a backward sweep to achieve a complete numbering of the edges of all the polygons in a space efficient manner.

An application in which it may be important to have the edges completely ordered is that of producing a plot of the set of polygons (perhaps representing a VLSI design) using a mechanical plotter. Here it is plainly wasteful to plot the edges in the order they are encountered by a sweep line, for this means raising the pen and moving to another part of the design after plotting each line. It is clearly preferable to plot complete polygons at once without removing the pen from the paper. This requires the ordering of edges which is the subject of this paper.

Another application is in the resizing problem in mask generation. In this case, we are given a set of polygons in various layers. Appropriate boolean operations on the layers are carried out in order to generate the mask layers. Before lithography, however, it is often desirable to resize the mask by over or undersizing it by a small amount. This comes down to drawing a polygon slightly inside or slightly outside each polygon in the mask. In order to do this, it is convenient to have the edges of each polygon in order.

2. Outline of the algorithm

The input to the reassembly algorithm will consist of a set of edges of polygons. As a first step, these will be ordered according to their maximum y-coordinate. That is the edges will be arranged in the order in which a sweep line sweeping upwards through the plane will encounter their higher end point. Note that this is the natural order in which edges are output by a plane-sweeping algorithm in many problems, and so no sorting is required. In fact, rather than accepting input from a file, the algorithm to be described can be tacked immediately on to the end of a plane-sweep implementing, say a contour algorithm. It will also be assumed that each edge is supplied with an orientation which points one way around the perimeter of a polygon. If this is not the case, then minor changes would be needed in our algorithm. For convenience, we will make the usual assumption that the edges are in general position. By this is meant that no two vertices have the same y-coordinate (which in turn means that there are no horizontal edges and that no two vertices coincide). It will not be assumed, however that the polygons are non-overlapping, although this will be a common application.

The output of the algorithm will be the same edges in the opposite order, each with two labels attached, the *polygon number* of that edge, specifying which polygon it belongs to, and the *sequence number* of the edge. The sequence number of the edge will be such that if one numbers the edges of the polygon starting with the highest vertex and proceeding in the direction of orientation of the edges, then each edge will receive its own sequence number. If required, a simple sort on these two keys will sort the output file into polygons and the edges into order.

The key concept in the reassembly algorithm is that of a derived polygon which will now be defined. Given a polygon P, the *derived polygon* of P, denoted P' is the polygon, the vertices of which are the maxima of P in the order they occur around the perimeter of P. By a maximum is meant a vertex which lies above its two incident edges. Figure 1 shows a polygon with its derived polygon shown in dotted lines.

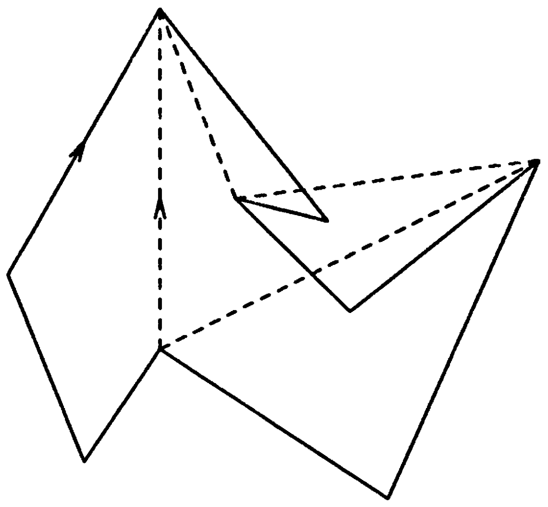

figure 1.

The sequence of edges hanging between two maxima is called a *chain*. This is a pun, for on the one hand the edges in a chain hang between the two maxima like a catenary, and on the other hand, it will be possible to assign an ordering to the edges in the chain. The second derived polygon of P is the derived polygon of P', and so on. Inductively one can define the i-th derived polygon, $P^{(i)}$. An edge of the i-th derived polygon is referred to as an *i-edge* and a chain made up of i-

edges will be called an *i-chain*. They may also be referred to as edges or chains of *level* i. The following piece of notation will be useful. Given an i-edge, $e^{(i)}$, denote by $De^{(i)}$ the $(i+1)$-edge which corresponds to the i-chain containing $e^{(i)}$. We will say that $e^{(i)}$ *belongs* to the edge $De^{(i)}$.

It is quite possible, indeed likely, that the derived polygons of a simple polygon may not be simple (that is will have self intersections). This will concern us not at all.

Let C denote the maximum number of 0-edges which meet any horizontal cross-section of the plane. It is a simple observation that no cross-section can meet more than C edges of any fixed level, i. More will be said about this point later on.

A *local ordering* of edges is an assignment of *local sequence numbers* to the edges of a chain. These local sequence numbers are integers assigned to the edges in the chain which increase with respect to the orientation of edges in the chain. For the time being sequence numbers will be consecutive integers.

It is important to note that $P^{(i+1)}$ has at most half as many edges as $P^{(i)}$, and so the length of the derived sequence is bounded by the logarithm of the number of vertices of P. At the penultimate level, the $(m-1)$-st derived polygon will have a single maximum and a single minimum and hence will consist of a single chain. Then, at the next and final level, the m-th derived polygon will consist of a single edge both of whose endpoints are equal (this will be called a *null edge*), after which the derived sequence will terminate. The reassembly algorithm uses this finiteness of the derived series as the basis for recursion. The procedure reassemble given below takes as input a file P_i containing the set of i-edges and gives as output a file S_i which contains the i-edges complete with polygon numbers and sequence numbers, that is, reassembled. It makes use of a temporary file I_i. The outline of the complete reassembly algorithm follows.

```
procedure reassemble ( i : integer );
    begin
    Assign local sequence numbers to the $i$-edges which are read from
    the file $P_i$; Write the edges with their local sequence numbers out
    to file $I_i$;

    Produce the file $P_{i+1}$;

    if $P_{i+1}$ is not empty then reassemble (i+1);

    Combine $S_{i+1}$ (which gives the sequence numbers of the $(i+1)$-
    edges, that is of the $i$-chains) and the local sequence numbers (giv-
    ing the order within the $i$-chains) contained in the file $I_i$ to produce
    the reassembled file $S_i$
    end;

begin (* Main program *)
reassemble (0)
end.
```

Let us look a little more closely at what is happening here. The file P_0 is read and two output files are produced, I_0 and P_1. The procedure then calls itself recursively, whereupon P_1 is read and I_1 and P_2 are produced. This continues for m stages until eventually P_{m+1} is empty. At this point we have $m+1$ locally ordered files, I_0 to I_m. This will be called the first phase of the algorithm. It may be represented schematically by figure 2.

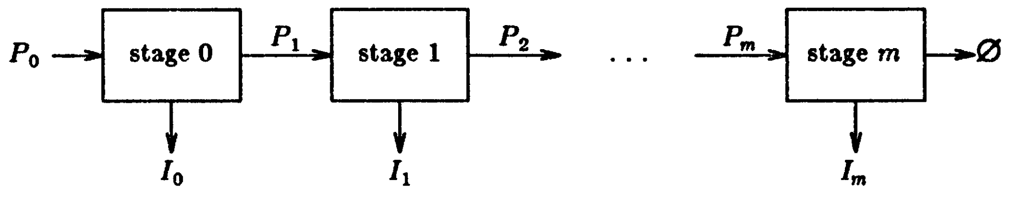

figure 2.

During the second phase of the algorithm, we work our way back up through the levels of recursion. At the i-th stage, file I_i is combined with the reassembled file S_{i+1} to produce S_i. This may be represented schematically by figure 3.

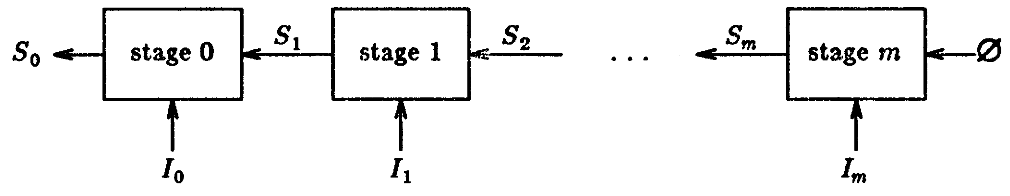

figure 3.

3. Details of phase 1 - an upwards sweep.

Suppose that we are considering the i-th stage of phase 1, the input edges being i-edges. Imagine a line sweeping upwards past a polygon and that we are trying to assign local sequence numbers to the edges as we see them. Each chain contains just one minimum. This minimum point divides the chain into a *rising branch* and a *falling branch*. The sweep line will first encounter a chain at the minimum and will then sweep simultaneously up the rising branch and backwards up the falling branch. If the two edges meeting at a minimum are numbered -1 and 0 and if the edges of the rising branch are numbered by counting upwards from 0 and the edges of the falling branch are numbered by counting backwards from -1 as they are met by the sweep line, then the edges of the chain will be numbered in order. Figure 4 shows a chain with the local ordering of its edges.

The local sequence numbers can be assigned to each edge as soon as it is seen and each edge can be written out immediately to the file I_i along with its local sequence number and a label representing which chain it belongs to. In order to do this, it is necessary to maintain a data structure keeping information about the state of each chain which currently intersects the sweep line. The required information for each chain will be the vertex currently at the top of both the rising and falling branches, the number of edges in each of the two branches and a label which has been assigned to the chain. These labels must be reused when possible but in such a way that no two chains which are simultaneously

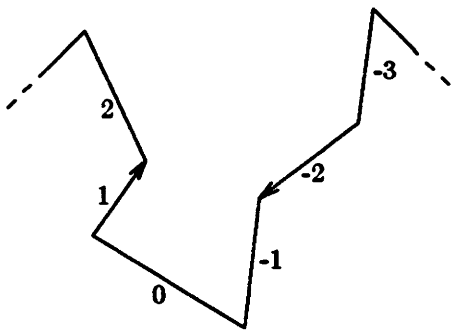

figure 4.

active have the same label. Thus, when the sweep line finishes passing over a chain, its label becomes available for reuse. Since there can be no more than C chains meeting the sweep line at any time, a total of C labels will be used and the space requirement will be $O(C)$. When a new edge is read in, it is necessary to find which chain it belongs to by comparing its bottom vertex with the vertices at the top of the branches of the currently active chains. If it does not match any of these vertices, this means, of course that the edge belongs to a new chain. The search will require $O(\log C)$ time and so there will be a total time requirement of $O(n_i \log C)$ to assign local sequence numbers and chain labels to each edge. Here, n_i is the total number of i-edges. Since $n_i \leq n_0/2^i$, summing over i gives a total time requirement of $O(N \log C)$ for searching.

Now, during the sweep just described, the individual chains are being built from the bottom up. When the chain is fully built, both the rising and falling branches being complete, the vertices at the top of each of these branches are the two vertices of an edge of the derived polygon. This edge is written out along with its label (that is, the label which was assigned to the completed chain) to the output file P_{i+1} of $(i+1)$-edges. Notice that the edges in this file are ordered according to their higher end point, just as the edges in the original input file were. At this time, the completed chain should be deleted from the chain table and the label on the chain be made available for reassignment.

At the end of the first phase we will have files I_i for $0 \leq i \leq m$. The file I_i will contain along with each i-edge, $e^{(i)}$ the following information : label of $e^{(i)}$; label of $De^{(i)}$; local sequence number of $e^{(i)}$. The formal description of this first phase algorithm as just described will be the same as that given in section 7 with the following exceptions: there will be no reference to level, falling_length or rising_length; where falling_length appears on the right hand side of an assignment, replace it by 0 and replace rising_length by 1; replace the recursive call to treatedge by a write to the file P_{out}.

4. Details of the second phase - a downward sweep.

During stage i of the second phase of the algorithm, the file S_{i+1} containing the reassembled $(i+1)$-edges is combined with I_i containing the locally ordered i-edges to produce the file S_i. We think of this step in terms of a downwards plane sweep. At all times during the sweep we maintain a look-up table containing information about all $(i+1)$-edges currently meeting the sweep line. This table will be an array indexed by the label of the $(i+1)$-edge and as data it will contain the polygon number and the sequence number of that edge as read from the file S_{i+1}. The edges in the file S_{i+1} will be arranged in the order that the downward sweep will first encounter them. Thus, when the sweep encounters a new $(i+1)$-edge, that edge is read from the file and placed in the look-up table. Since there are a maximum of C labels in use, this table will require $O(C)$ space. Edges will be read one by one from the file I_i and the $(i+1)$-edge which they belong to will be looked up in the table. (Remember that I_i contains for each edge $e^{(i)}$, the label of the $(i+1)$-edge it belongs to.)

Notice that the i-edges are arranged in I_i in such a way that by reading the file backwards, we will encounter them in the order they will be met by the downward sweep. Suppose then we have read a new i-edge, $e^{(i)}$ from I_i, and that we have found the $(i+1)$-edge, $De^{(i)}$ to which $e^{(i)}$ belongs, by looking it up in the table. The following information is now known about the edge $e^{(i)}$: label of $e^{(i)}$; polygon number of $De^{(i)}$ (and hence of $e^{(i)}$); sequence number of $De^{(i)}$; local sequence number of $e^{(i)}$; This may be written out to a temporary file. The pairs (sequence number of $De^{(i)}$, local sequence number of $e^{(i)}$) may subsequently be lexicographically ordered within each polygon to give the sequence numbers of the i-edges. In other words, these pairs may be compressed to single sequence numbers. This may require sorting, but it can be done in $O(n \log n)$ time and constant space, so it will be ignored. Besides, in the next section, it will be seen how to get the sequence numbers directly without any such sorting.

During the sweep it will often occur that an i-edge, the next edge in I_i, and an $(i+1)$-edge, the next edge in S_{i+1} are both met by the sweep line at the same time. In order for the look-up to succeed, the $(i+1)$-edge must be read into the look-up table before the i-edge is processed.

Sometimes, we will come across a null edge in the file I_i. This means that a polygon disappeared at this stage in phase 1 and hence the null edge belongs to no $(i+1)$-edge. Correspondingly, in our backwards traversal of the files I_i during phase 2, we are meeting this new polygon for the first time. The appropriate action is to assign a new polygon number to this edge and call it edge number 1 in chain number 1.

With n_i defined as before, the i-th stage of this sweep will take time $O(n_i)$. Summing over i we get a total time requirement of $O(n)$, ignoring the time required for compressing sequence numbers.

The algorithm just described is much the same as that formally presented in section 7, except that instead of the assignment statements updating the table, the records are written out to the file S_{out} and read in again at the next stage.

5. First improvement - obtaining sequence numbers directly.

The algorithm as presented is in fairly rough form, though fine for theoretical purposes. However it is amenable to a number of aesthetic and practical improvements.

The first troublesome feature is the fact that the sequence numbers we get are in fact ordered pairs which need to be compressed either at the end of the pass, or between consecutive stages. It would be nice if we could avoid having to do this by assigning integer sequence numbers at once. This is indeed possible using a different definition of sequence number. Define the *reference vertex* for an edge as follows. For a 0-edge (an edge of the original set of polygons), the reference vertex is the vertex at the tail of the edge with respect to its orientation. An i-edge, $e^{(i)}$, corresponds to a chain of $(i-1)$-edges. The reference vertex for $e^{(i)}$ is the minimum vertex on that chain. Note that the reference vertex is not in fact one of the two vertices of the edge, but rather a vertex of the original polygon. Next define the *sequence number* of an edge to be the number of 0-edges lying between the highest point on the polygon and the reference vertex of the edge counting around the perimeter in the direction of orientation. Denote this by $\sigma e^{(i)}$. Note that it is the sequence numbers of the 0-edges that we ultimately wish to calculate.

Define the *local offset* of an edge $e^{(i)}$ to be the integer $\lambda e^{(i)}$ defined to equal $\sigma e^{(i)} - \sigma D e^{(i)}$. In the case where $e^{(i)}$ is a null edge, define $\lambda e^{(i)} = \sigma e^{(i)}$. Rearranging this definition, we get

$$\sigma e^{(i)} = \sigma D e^{(i)} + \lambda e^{(i)} \tag{5.1}$$

if $e^{(i)}$ is not a null edge. This suggests that the sequence number of $e^{(i)}$ can be determined by adding the sequence number of $De^{(i)}$ (which may be found by induction) to its local offset.

We would like to calculate local offsets of i-edges during an upwards plane sweep, just as we calculated the local sequence numbers in the original algorithm. This is possible without any great alteration as long as we carry two pieces of extra information around with each edge, namely its *rising length* and its *falling length*. The falling (or descending) length of an edge $e^{(i)}$, denoted by $\delta e^{(i)}$ is the number of 0-edges which lie between the starting vertex of the edge and its reference vertex. Similarly, the rising length, $\rho e^{(i)}$ is the number of 0-edges between the reference vertex and the final vertex of the edge. Clearly, $\delta e^{(i)} + \rho e^{(i)}$ gives a measure of the *length* of the edge. Notice that for a null edge, we have

$$\sigma e^{(i)} = \delta e^{(i)} = \lambda e^{(i)} \tag{5.2}$$

It should be clear now how local offsets are assigned to edges during the first phase of the algorithm instead of the local order numbers previously discussed. On the backward sweep, the sequence numbers are determined using (5.1) or (5.2). The full algorithm is given later.

6. Simplifying file accesses.

It may be thought unpleasant that the number of files required is $O(\log N)$. This can be avoided by noting that no more than two of the files P_i need exist at one time, and the same is true for the files S_i. Furthermore, the files I_i are read in backwards order and each file is read backwards. Thus, it is possible to string them all together with markers between the separate stages. In this way, we can get away with three files only.

There is still excessive reading and writing of files, since files P_i are written and then read in the same order. It would be possible to pipe the output from stage i to the input of stage $i+1$. A similar thing could be done during phase 2. Unfortunately, it would no longer be possible to string the files I_i end to end as just suggested since they are created simultaneously. It turns out, however that the output from all the stages of phase 1 may be combined, all mixed up together. This turns out to be not only possible, but advantageous, as it simplifies the algorithm. Instead of thinking of piping the various stages of the phase 1 forward sweep together, however, we will have them call each other recursively.

Consider the forward sweep. This will be implemented as a single procedure treatedge which will be passed each of the input edges in turn. As before, edges will be provided with local offsets and chain labels and written out to a file I. When a chain is completed and an edge of the derived polygon is thereby found, instead of writing it to a file as before, it is passed recursively to the procedure treatedge. Thus the file, I will contain edges of all levels mixed together. Note, however that an $(i+1)$-edge, $e^{(i+1)}$ is not written out until all the i-edges making up the corresponding i-chain have been written. Conversely, reading the file backwards, we will come across the entry representing the edge $e^{(i+1)}$ before encountering any of the i-edges which go to make it up. Because of this, the table look-up will always find the correct $(i+1)$-edge.

The edges of all levels are labelled with one series of labels. In general there will be $\log N$ stages running at once, and so $O(C\log N)$ labels used and that amount of storage required for the chain tables. Thus we pay a $\log N$ space penalty for simultaneous execution. In fact, the number of levels can not be greater than the logarithm of the number of edges in the largest polygon and so the penalty may not be great. In the forward sweep, the chain tables for chains of different levels must be kept separate, and so each edge must have its level recorded with it. In the backwards sweep (phase 2), this is not necessary, since lookup is by chain label.

7. Formal description of the algorithm.

A formal description of the algorithm is given next in some sort of quasi-Pascal. For convenience, values of records are specified by listing their fields inside braces. We start with the format of the input, intermediate and output files.

```
type
  edge_type = record
              top_vertex    : point;
              bottom_vertex: point;
              orientation   : (up, down);
              label         : integer;
              level         : integer
          end;

  P_record = record
             edge           : edge_type;
             falling_length : integer;
             rising_length  : integer;
           end;

  I_record = record
             edge           : edge_type;
             chain_label    : integer;
             local_offset   : integer;
           end;

  S_record = record
            edge            : edge_type;
            polygon_number      : integer;
            sequence_number     : integer;
          end;
```

During the first phase, we will keep an array of chain tables, one for each level. The data contained in these table is described by the following statements.

```
type
  branch  = record
            length          : integer;
            top             : point;
            is_complete     : boolean;
          end;

  table_entry =  record
                 chain_label : integer;
                 rising_branch      : branch;
                 falling_branch     : branch
               end;
```

The following functions and procedures will be used to access this table.

```
function Insert ( vertex : point ) returns ^table_entry;
function find_in_rising_branch ( vertex : point )
                    returns ^table_entry;
function find_in_falling_branch ( vertex : point )
                    returns ^table_entry;
procedure delete ( node : ^table_entry );
```

The function find_in_rising_branch will look in the table associated with the level in question for a table_entry, T, with T.rising_branch.top = vertex and will return its location if found. Otherwise it will return a NULL pointer. Similarly, find_in_falling_branch will search for the vertex in the falling branch.

Insert will generate a new label, lab, and insert the record

```
{ chain_label := lab,
  rising_branch :=
      { length := 0, top := vertex, is_complete := false },
  falling_branch :=
      { length := 0, top := vertex, is_complete := false }}
```

in the table for the appropriate level.

Finally, delete will delete the node pointed to. It is now possible to give the main routine of the first sweep.

```
procedure treatedge ( input : P_record );
  var T1, T2 : ^table_entry;
  begin

  (* Check whether this is a null edge *)
  if input.edge.top_vertex = input.edge.bottom_vertex then
    begin (* See (5.2) *)
    write ( { edge := input.edge,
          chain_label := NULL,
          local_offset := input.edge.falling_length } ) to file I ;
    return
    end;

  (* Consider first the case where the edge is oriented upwards. *)
  if input.edge.orientation = up then
    begin

    (* Find which chain the input edge belongs to *)
    T1 := find_in_rising_branch
          (input.edge.bottom_vertex, input.edge.level);
    if T1 = NULL then
      T1 := Insert (input.edge.bottom_vertex, input.edge.level);
  (* Write out a new record to file I *)
  write ( { edge := input.edge,
          chain_label := T1^.chain_label;
          local_offset := T1^.rising_branch.length
               + input.edge.falling_length } ) to file I ;

  (* Update the chain *)
  T1^.rising_branch.top := input.edge.top_vertex;
  T1^.rising_branch.length := T1^.rising_branch.length +
               input.edge.falling_length + input.edge.rising_length;

  (* Now determine whether we are at a maximum. *)
  T2 := find_in_falling_branch
        ( input.edge.top_vertex, input.edge.level );
  if T2 = NULL then return;

  (* We are at a maximum.  Output (i+1)-edges as necessary. *)
  T2^.falling_branch.is_complete := true;
  if T2^.rising_branch.is_complete then
    begin
    treatedge ({ edge := {
                     top_vertex := T2^.falling_branch.top,
                     bottom_vertex := T2^.rising_branch.top,
                     orientation := down,
                     label := T2^.chain_label,
                     level := input.edge.level + 1 },
               falling_length := T2^.falling_branch.length,
               rising_length := T2^.rising_branch.length });
    delete ( T2 );
    end;

  T1^.rising_branch.is_complete := true;
  if T1^.falling_branch.is_complete then
    begin
    < similar to the treatment of T2 >
    end;

  end; (* Of the case where edge.orientation = up *)

if input.edge.orientation = down then
  begin
  <similar to the up case >
  end;

end;
```

This procedure should be passed each of the 0-edges in turn specifying edge.level := 0, edge.label := anything (since labels on 0-edges are never used), falling_length := 0, rising_length := 1.

Next consider the second sweep of the algorithm.

```
 var table = array [1..C] of record
                      polygon_number : integer;
                      sequence_number  : integer;
                    end;
   newpolygon : integer;
   input : I_record;
   D_edge : S_record;

begin
newpolygon := 1;

while records still left in file I do
  begin
  read ( input ) from file I ( reading the file backwards );

  if input.chain_label = NULL then  (* This is a null edge *)
    begin  (* See (5.2) *)
    table [input.edge.label] :=
              { polygon_number := newpolygon,
              sequence_number := input.edge.local_offset };
    newpolygon := newpolygon + 1
    end

  else
    begin
    D_edge := table [input.chain_label];

    if input.edge.level = 0 then
      write ( { edge := input.edge,
             polygon_number := D_edge.polygon_number,
             sequence_number := input.local_offset +
                      D_edge.sequence_number
           } ) to output  (* See (5.1) *)

    else table [ input.edge.label ] :=
           { polygon_number := D_edge.polygon_number,
             sequence_number := <same as above> };
    end
  end
end.
```

The algorithm as presented assumed that the edges were oriented. In the case where we are presented with unoriented edges, certain changes need to be made. One can assign arbitrarily an orientation to each edge. The orientations will not be consistent around the perimeters of polygons. When an edge is read in, it will be necessary to search for its lower vertex in both branches of each chain. The exact details of computing offsets will depend on whether the orientation of the edge is consistent with the orientation of the chain or not. The reader is invited to fill in the details.

8. Avoiding logarithmic time searches.

The part of this algorithm which forces it to have time bound of $O(n \log n)$ instead of linear time is in the searching of the chain tables. As each i-edge is considered, a search is made to determine which i-chain it belongs to. Further, if it happens to adjoin a maximum, then another search is necessary to determine the other i-chain which meets it at that maximum. At levels higher than 0 this is in fact wasted effort, for this information may be deduced at the moment that the i-edge is generated in stage $i-1$ and can be passed along with the edge to the next stage. The i-chains may then be kept in an array indexed by their chain number, and search time will be constant instead of logarithmic.

More explicitly, suppose two i-chains, $c_1^{(i)}$ and $c_2^{(i)}$ meet at a point p. Suppose, however, that when the sweep passes p, neither of the two chains is yet complete. This will mean that the two corresponding $(i+1)$-edges, $e_1^{(i+1)}$ and $e_2^{(i+1)}$ must form a minimum at p. In other words, a new $(i+1)$-chain will start at that point. A new label should be generated for this new chain and recorded in the nodes which service the i-chains $c_1^{(i)}$ and $c_2^{(i)}$ (let us say in a new field called next_level_chain_label). Later, when the i-chains are complete, and the corresponding $(i+1)$-edges are passed on to the next level, the label for the chain they belong to can be passed as well. Figure 5 a shows the situation described here.

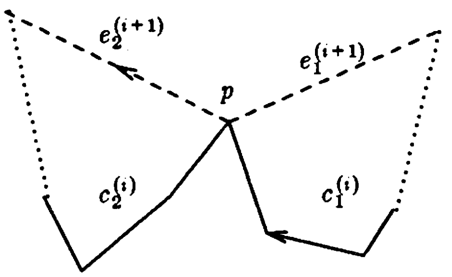

figure 5a.

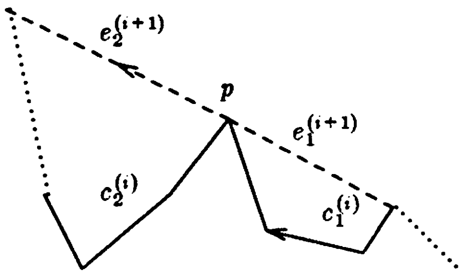

figure 5b.

Similarly, suppose that $c_1^{(i)}$ is completed at point p, whereas $c_2^{(i)}$ is not. This means that p will be an *elbow* at the next level. In other words, $e_1^{(i+1)}$ and $e_2^{(i+1)}$ belong to the same $(i+1)$-chain. The appropriate action is to copy the field next_level_chain_label from the node for $c_1^{(i)}$ to that of $c_2^{(i)}$. The edge $e_1^{(i+1)}$ is then passed to the next level as before. This is illustrated in figure 5 b.

The case where both the i-chains are complete at p corresponds to a maximum at the next level. Details are omitted.

Thus, searches at levels greater than 0 can be done in constant time. At level 0 the information required to avoid a logarithmic search may not be available. If, however, the input to the reassembly algorithm comes from a preliminary plane-sweep, for instance a contour or boolean operations algorithm, then by a minor adjustment, it is often possible that the required information can be supplied in additional time linear in the size of the output. In such a case, the reassembly algorithm can be tacked on to the end of the preliminary algorithm at the cost only of a time increase linear in the size of the output.

9. Examples and Counter-examples

In this final section an attempt is made to anticipate certain questions which may occur to the reader concerning the properties of derived polygons. Suppose that we have a polygon with n edges. Is it not possible to find a better bound than $O(\log n)$ for its derived length? The answer to this is no. In fact, given any polygon, P_0, it is easy to construct another polygon P_1 which has twice as many vertices as P_0 and such that $P_0{=}P_1{}'$. This is done simply by attaching a small "V" between each pair of consecutive vertices of P_0. Starting with P_0 having a single null edge, we can construct, inductively, a polygon P_m which has 2^m edges and derived length m.

The next example is meant to approximate the power distribution (Vdd) net of a VLSI chip. Fix a constant c. Define a 1-comb to be a thin vertical rectangle. Now define inductively an i-comb to be the polygon obtained by taking c $(i-1)$-combs, rotating them 90 degrees clockwise and attaching them to the right of a vertical stem. Figure 6 shows a 3-comb with $c{=}3$.

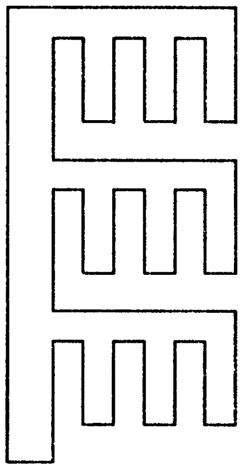

figure 6.

Now an m-comb has $O(c^m)$ edges. Rotate it slightly to put it in general position. It can be seen that it has derived length m which is $O(\log n)$ where n is the number of vertices.

It was remarked earlier that if C_i is the maximum number of i-edges which meet any sweep line, then for all i, $C_i \leq C_0 = C$. For this reason, when we run all stages at once, as discussed above, the total space requirement is not larger than $O(C\log n)$. The question naturally occurs whether any relation such as $C_{i+1} < k.C_i$ may not hold with k a constant less than one. If this were the case, then summing over i, we would get a space requirement of $O(C)$ instead of $O(C\log n)$. In fact, this is not the case. The following example shows that, indeed the space requirement can be $O(C\log n)$.

Define an order 1 saw-tooth, denoted s_1, to be a simple minimum. We will enclose it in a box to show how important it is. Now, the order i saw-tooth, s_i, is defined inductively by taking two copies of s_{i-1}, placing one above and to the right of the other, joining them as shown in figure 7 and enclosing in a box.

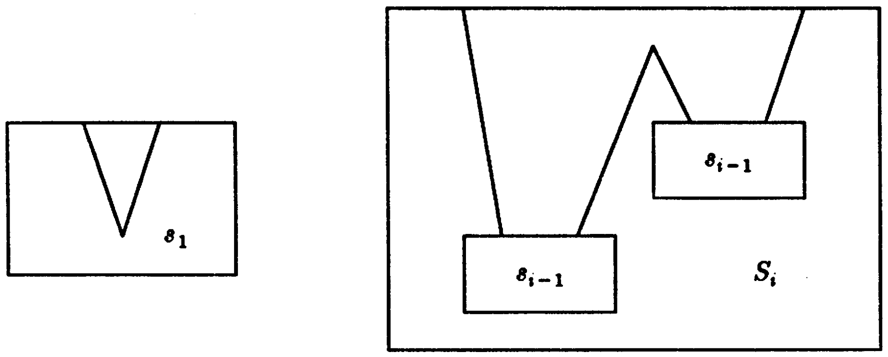

figure 7.

An important thing to note about saw-teeth is that a cross-section meets at most $2i$ edges of s_i and in fact this maximum is achieved. The proof is a simple induction. Now define an m-saw to be the polygon made up of saw teeth of orders 1 to m placed and connected as shown in figure 8. (Presumably a saw with teeth of orders 1 to c would be called a c-saw.) Note that the cross-section shown in the figure meets P_m in $2m$ points, and that is the maximum possible. Counting edges reveals that an m-saw has $n = 2^{m+1} - 1$ edges. Now, the decisive property of saws is that the derived polygon of P_i is simply P_{i-1}. Each saw tooth, s_i becomes a tooth s_{i-1}, and s_1 disappears. The cross-section in figure 8 will meet $2(m-1)$ edges of the derived polygon and so on. In fact, counting edges of all the derived polygons, the cross-section meets a total of $m(m+1)$ edges. Since $m = ½C$ and $m+1 = \lceil \log n \rceil$, this equals $½C \lceil \log n \rceil$ which is $O(C\log n)$ as required.

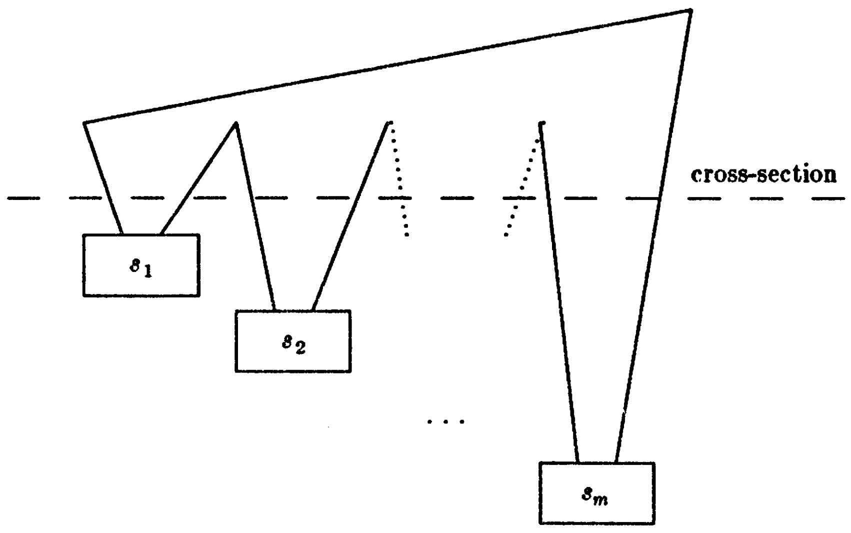

figure 8.

References

[NP] Nievergelt, J. and Preparata, F.P., Plane sweeping algorithms for intersecting geometric figures. *Communications of the ACM* **25**, (1982), 739-747.

[OW] Ottmann, T. and Wood, D., The contour problem for polygons. *University of Waterloo Technical Report CS-84-33*, (1984)

[OWW]
Ottmann, T., Widmayer, P. and Wood, D., A fast algorithm for the Boolean masking problem. *University of Waterloo, Computer Science Technical Report CS-82-37*, (1982).

[SvW]
Szymanski, T.G. and Van Wyk, C.J., Space efficient algorithms for VLSI artwork analysis, *Proceedings of the 20th IEEE Design Automation Conference* (1983), 734-739.