AN ALGORITHM FOR LOCATING CANDIDATE LABELLING BOXES WITHIN A POLYGON[*]

Jan W. van Roessel[**]
TGS Technology, Inc.
EROS Data Center
Technique Development and Applications Branch
Sioux Falls, South Dakota 57198

ABSTRACT

Vector-based geographic information systems usually require annotation, such as a polygon number or attribute data, in a suitable location within a polygon. Traditional methods usually compute the polygon centroid, test the centroid for inclusion or exclusion, and select some alternative point when the centroid falls outside the polygon. Two problems are associated with this approach: (1) the text can be centered on the point, but may be placed in a visually awkward place, and (2) part of the text may fall outside the polygon and may overlap other polygon boundaries or other text labels. An algorithm is presented that circumvents both of these problems, by computing a number of horizontal candidate labelling rectangles (boxes) within a polygon from which a suitable selection can be made or from which one may conclude that the text label does not fit the polygon.

INTRODUCTION

The placement of feature annotation on maps is an important and difficult problem in automated cartography; it has been the subject of extensive research in recent years. In particular, the placement of labels in the vicinity of point features has been a subject of investigation as reported by Cromley (1986), Langran and Poiker (1986), and Mower (1986). However, two other types of problems also exist, namely, labelling of lines and polygons for which the applied methodology is still rather primitive in many systems. This paper will address the polygon labelling problem.

In a number of systems the polygon centroid (or a derivative) is used to locate a label within a polygon. As the centroid may fall outside the polygon, a fairly typical approach is to check this fact with a point-in-polygon method and then to shift the label to an arbitrary point inside the polygon. This method often leads to labels that appear in awkward locations and may partially overlap with the polygon boundary. The computed centroid is only a single point, whereas a label can best be represented by a minimum box bounding the text of the label. This minimum box must then be located such that (1) the box is wholly contained with the polygon, and (2) the box appears at a pleasing location. One may even want to repeat the text label at several locations when the polygon is large or complex.

The objective for this paper is to present a method for computing a number of candidate labelling boxes from which one or more suitable boxes may be selected. If no box can be found that is large enough to contain the label, one may then conclude that the label will not fit and some other placement action must be

* Publication authorized by the Director, U.S. Geological Survey.
** Work performed under U.S. Geological Survey contract 14-08-0001-20129.

taken.

In a subsequent process one may select a number of labelling boxes from among the candidate boxes, taking into consideration the characteristics of the polygon and the label. This process will only be briefly discussed.

APPROACH

The basic idea is to first divide the polygon into horizontal *strips*, where each strip boundary line passes through a vertex, and then to place vertical line segments on the polygon boundary segments located within the strips from which the boxes can be created by "sweeping" over these vertical segments in a left-to-right direction.

The polygon is first divided into strips by "drawing" a horizontal line through each of its vertices, as shown in figure 1. For a polygon with *N* vertices,

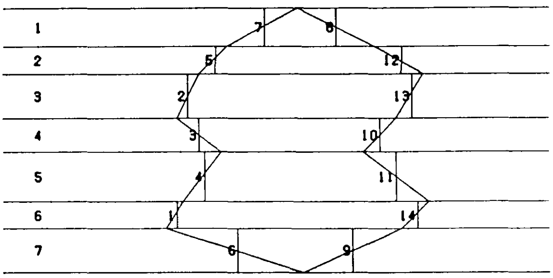

Figure 1.--A simple polygon with strips and vertical line segments.

this divides the plane into *N+1* strips, but the strips above and below the vertex points with maximum and minimum *y* coordinates are not of interest, so that *N-1* strips fully contain the polygon.

The polygon line segments are subdivided by the strips such that each strip has an even number of divided segments located within the strips. The subdivided line segments will be referred to as *strip segments.* There are two strip segments in a strip for simple (no islands) convex polygons, and a multiple of two for non-convex polygons or polygons with islands (complex polygons).

Within a strip, a vertical line called a *vertical segment* can be placed through each strip segment to guide the formation of the candidate boxes (see figure 1). A vertical segment can be placed anywhere between the vertex points of a strip segment, but the strip segment midpoint has been selected. Other choices are the innermost or outermost vertex of a strip segment. The objective is to compute a set of maximal boxes, such that each box cannot be expanded further in the horizontal and vertical directions without crossing the polygon boundary. The present method only approaches this condition because each vertical segment becomes a part of a box, which, because of the midpoint location of the segment, will be partially outside the polygon boundary. An alternative is to locate the vertical segments at the innermost vertex of the strip segment; this guarantees that each box will be wholly inside the polygon, but some boxes will have zero area, so the space within the polygon will not be used as efficiently. Experience has shown that with the density at which natural resource applications polygons are usually digitized, the midpoint choice produces boxes that may have little "corners" outside the polygon. But in most cases, because the text box is usually smaller than the candidate labelling box, this is never graphically revealed.

The left and right boundaries of a box are formed by lines coincident with a left and right vertical segment, hereafter referred to as the left defining segment (*LDS*) and right defining segment (*RDS*). The bottom and top boundaries of the box are coincident with the respective lower and upper strip boundaries of the lowest and highest strip still contained within the box.

For instance, the box shown in figure 2 is constructed with left and right

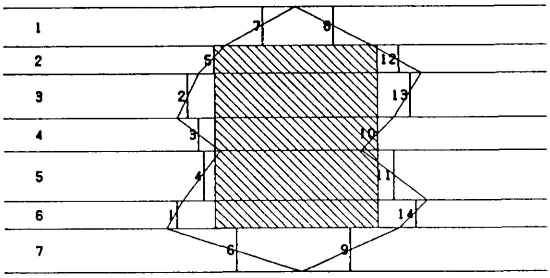

Figure 2.--Simple polygon with sample box.

boundaries through vertical segments 5 and 10, and with the bottom and top boundaries coinciding with the lower and upper boundaries of strips 6 and 2. In general, a box is denoted as: *box(left, right, bot, top)*, where *left* and *right* are the vertical segment numbers of the *LDS* and *RDS* and *bot* and *top* are the bottom and top strip numbers.

After the polygon has been divided into strips and the vertical segments have been generated, the vertical segments are sorted from left to right by increasing x coordinate. A vertical segment number is then assigned which is the sequence number in this sorted order.

Each vertical segment has an associated strip number $s(i)$, where i is the vertical segment number in sorted order, and $i = 1, \ldots, n$. The x coordinate of a vertical segment is denoted by $x(i)$. The vertical segments in a strip can be ordered from left to right as consecutive left and right pairs. In the left-to-right scan, only one pair is active in a strip at a time; a pair becomes inactive as soon as the right segment has been processed. Denote the left segment of this active pair by $l(s)$ and the right segment by $r(s)$. Then the following conditions hold for a maximal box: *boxmax(left, right, bot, top)*:

$$Pleft: \; x(left) \text{ is } \max(x(l(s))), \; s=bot,\ldots,top$$

$$Pright: \; x(right) \text{ is } \min(x(r(s))), \; s=bot,\ldots,top$$

From these two conditions, two resultant conditions for the top and bottom of the box can be derived, knowing that *Pleft* and *Pright* must be violated in the strips directly above and below the box, otherwise the box could be extended in these directions:

$$Ptop: \; x(l(top-1)) > x(l(top)) \textbf{ or } x(r(top-1)) < x(r(top))$$

$$Pbot: \; x(l(bot+1)) > x(l(bot)) \textbf{ or } x(r(bot+1)) < x(r(bot))$$

Another important condition is that the *LDS* and *RDS* of the box must coincide with the box boundary and therefore the following condition holds:

$$Pinc: \; bot \leq s(left) \leq top \textbf{ and } bot \leq s(right) \leq top$$

The approach taken for constructing a set of maximal boxes from the polygon is based on the above five conditions.

Each left vertical segment is potentially the *LDS* of one or more boxes. Condition *Pleft* states that the *LDS* has the largest x coordinate of all active left vertical segments in the strip range of the box; therefore only segments to the left of the *LDS* can influence the vertical range of the box, left vertical segments to the right have no bearing on it whatsoever. This suggests sorting the vertical segments by x coordinate and processing them in sorted order. Each segment can then be entered into a working array in which the boxes are created, and can be processed against other segments already entered to determine the range of the box for which it is the *LDS*.

The working array may consist of a number of rows and columns, each row corresponding to a strip in the polygon, and the columns representing left, right, bottom, and top elements for boxes that are being generated. The working array row corresponding to a strip will be referred to as the *strip entry*. Defining segments are entered into the working array at the strip entry of the strip in which they occur. Finished boxes in the working array may be inspected for selection immediately after they are generated or they may be saved in an output array for later processing.

Conditions *Ptop* and *Pbot* suggest how to determine the bottom and top limits of a box. The idea is to probe upwards and downwards until strips are encountered that violate *Pleft* and *Pright*. In the downward direction *Pleft* becomes invalid when $x(l(bot+1)) > x(l(bot))$. But since the violating segment has an x coordinate greater than that of the *LDS* it cannot as yet have been entered into the working array. And since the strips for a box are adjacent, it suffices to scan downward in the working array until an unused strip entry is encountered. The number of the strip before the empty entry then is recorded in the bottom column for the working array strip entry of the *LDS*. The same procedure is followed for the top. A set of adjacent active working array entries between two unused entries at the top and bottom will be referred to as a *cluster*.

When processing vertical segments by sorted x coordinate, *Pright* implies that each encountered right segment automatically becomes the *RDS* for one or more boxes. The problem is to determine the boxes that are terminated by the segment. Conditions *Ptop* and *Pbot* hold for all working array entries in the cluster because of the left segment conditions $x(l(top-1)) > x(l(top))$ and $x(l(bot+1)) > x(l(bot))$. However, the *RDS* must also be on the box boundary. Therefore, subject to *Pinc*, all unfinished boxes in the cluster can be terminated by the *RDS* and be turned into completed boxes. Not all entries in the cluster may qualify.

The next problem is to consider which cluster entries that have produced boxes with the current *RDS* can live on in the left-to-right scan to produce more boxes, and how their top and bottom limits should be adjusted. Certainly the strip entry for the *RDS* must be terminated because the minimum right segment has been encountered. However, the other entries (satisfying *Pinc*) can continue if the top and bottom limits are adjusted. The strip entry for the *RDS* can therefore be reverted to the unused condition so that the cluster is either reduced by one strip (if the *RDS* strip is at the bottom or the top of the cluster) or the cluster is split into two parts. Considering the right segment conditions $x(r(top-1)) < x(r(top))$ and $x(r(bot+1)) < x(r(bot))$ of *Ptop* and *Pbot*, these can now be interpreted to mean that the strip of the *RDS* becomes $bot+1$ as well as $top-1$ for the other boxes in the new clusters (assuming without loss of generality that the *RDS* splits the cluster into two new clusters.) This means that the top and bottom columns for the entries in the working array for the new clusters must be adjusted to reflect these new limits.

The overall problem of generating candidate labelling boxes can be efficiently separated into two problems: (1) to compute the vertical segments, given the polygon definition, and (2) to compute the boxes from the vertical segments.

In the next section, the approach for computing the vertical segments will be discussed, but, because of space limitations, an algorithm will not be presented. Instead the emphasis of this paper will be on the precise algorithm for computing the boxes, which follows thereafter.

COMPUTING THE VERTICAL SEGMENTS

The approach used for computing the vertical segments has been to use a line-sweep approach (see, for instance, Sedgewick, 1983, chapter 24). The vertices are sorted on the *y* coordinate to divide the space into strips. Each *y* coordinate is then processed in turn, and its associated line segment number is either entered or deleted from an "active list" of line segments. With each *y*, a new strip is defined, of which the lower limit is set to the upper limit of the previous strip, and for which the new *y* becomes the upper limit. Line segments in the active band are then clipped against these limits and the *x* coordinates of the midpoints of the clipped segments are computed and entered into a list, together with the strip number in which they occur.

If the *y* coordinate following the current *y* coordinate is greater than the previous *y* coordinate in the polygon boundary, the line is removed from the active list. If this is not the case, the line segment is recorded in the list.

When the line sweep is completed, the list of strip numbers of the strip segment midpoints representing the vertical segments, is sorted by the *x* coordinate. This sorted list is then input to the algorithm discussed in the next section.

COMPUTING THE LABELLING BOXES

The algorithm for computing the boxes will be explained through a stepwise refinement process. The objective is to arrive at a fully developed procedure presented in Pascal. In the initial step there is only the procedure heading and ending, together with a characterization of as yet undefined code in the middle. The characterization will be presented as a Pascal comment in {} brackets. At each step this characterization will be further refined, but the entire set of derived code will not be repeated at each step; only the local expansion will be presented. The entire algorithm is then given at the end of the section.

The procedure is entered with a list of strip numbers for the vertical segments, which are in sorted order. This is the only input required for this algorithm, given that the output boxes are defined in terms of the *LDS* and *RDS*, and upper and lower strip numbers. Thus, the following code is first proposed:

Step 1

```
procedure boxes(strip: intarray; n:integer;
   var w: workarray; var box: boxarray);
var i, s, b, t, m, nr: integer;
begin
{produce the boxes}
end;
```

boxarray is a type of an array of records, each record with four fields: *left*, *right*, *bot*, and *top*. The working array is *w*, while *box* is the array that will contain the boxes on output. The working array is initially filled with zeros, and has a valid *w[0]* record. These records will be referred to as entries as before, where each entry has a *left*, *right*, *bot*, and *top* element (the record fields).

Although, for reasons of clarity, the procedure has an output array containing the finished boxes, one might instead inspect them on the fly. With this approach other information, such as the y coordinates of the upper and lower strip limits, and the *x* coordinates of the *LDS* and *RDS* must be imported to the procedure.

The process of producing the boxes is driven by the sorted vertical segments. For each segment the strip number of that segment, *s*, is an important variable. The working array entry corresponding to *s* will be referred to as the *current entry*. The following refinement is therefore made:

Step 2 {produce the boxes}

```
for i:= 1 to n do
   begin
   s:= strip[i];
   {process a vertical segment}
   end;
```

To process a vertical segment, two necessary actions must be taken. First, within the working array, the limits of the affected cluster must be established. Second, it must be decided whether the vertical segment is an *LDS* or an *RDS*, and appropriate action must be taken in each case. This leads to step 3:

Step 3 {process a vertical segment}

```
{establish cluster limits}
if w[s].left = 0 then
   begin
   {process LDS}
   end
else
   begin
   {process RDS}
   end;
end;
```

The *LDS* or *RDS* decision is made by checking the current entry. Since the *LDS* must come before the *RDS*, it suffices to check whether the left element of the current entry is empty (zero).

To understand how the cluster limits can best be determined, it is first necessary to know how the *LDS* is processed, which is shown in the following step:

Step 4 {process LDS}

```
w[s].left:= i; w[s].right:= 0; w[s].bot:= b; w[s].top:= t;
```

This step entails that the left element is set to the vertical strip number *i*, that the right side of the potential box is as yet unknown (*0*), and that the bottom and top are set equal to the cluster limits, *b* and *t*. It is necessary to assign *0* to the right element, even though *w* is initialized to *0*, because entries may be recycled in concave and complex polygons.

To find the cluster limits, the simplest idea is to scan the working array from the current entry in the upward and downward direction until empty entries are encountered. There is a better method, however, that makes use of information already entered in the cluster for previous segments. A cluster grows with new working array entries, which either begin a new cluster or are added to the top or bottom of an existing one . Consider the latter situation for the moment.

Four cases arise: new bottom and top for the current entry must be found, and for each it must be considered whether the current entry is at the top or the bottom of the existing cluster. If the new entry is at the the bottom, at position s, then the new bottom is known: $w[s].bot = w[s-1].bot+1$ (strip numbers increase towards the bottom of the polygon). To find out whether the entry is at the bottom, one needs only to decrement the entry number, and see whether this entry is empty. If not, the current entry is at the top of the cluster.

If the current entry is at the bottom, $w[s-1].top$ should contain the top of the cluster as established for some earlier state, not necessarily the previous state, depending on whether entries were made at the top or bottom of the cluster. In this case visiting $w[w[s-1].top].top$ should yield a better estimate of the current top. However, this entry could point to itself, because the first entry for a cluster is necessarily confined to the strip for which it applies. Therefore, to make progress the next entry up ($w[w[s-1].top-1].top$) needs to be inspected. This progression is further pursued until the current top is reached.

This process can be illustrated with the following snapshot of the working array for one of the states related to figure 1

```
j  l r b t
-  - - - -
1 | 0 0 0 0                  where: j = working array
2 | 5 0 6 2  <-- top                    entry
3 | 2 0 3 3                           l = left
4 | 3 0 4 3  cluster                  r = right
5 | 4 0 6 3                           b = bottom
6 | 1 0 6 6  <-- bot                  t = top
7 | 0 0 0 0  <-- current entry
8 | 0 0 0 0
```

where vertical segment 6 must be entered, which lies in strip 7. As $w[7].left = 0$, it must be a left segment, and also since $w[8].left = 0$ the entry occurs at the bottom of the cluster. Therefore $w[7].bot := w[6].bot+1$, and the as yet incomplete working array entry 7 is set to: 6 0 7 0. To determine the top entry note that $w[6].top = 6$, and therefore points to itself. However, $w[w[6].top-1].top = 3$, and thus progress is made. But again $w[3].top = 3$, pointing to itself, so that $w[w[3].top-1].top = 2$ needs to be inspected. This entry again points to itself, but decrementing by 1 points into an empty entry, so that the top of the cluster has been found. The complete entry for strip 7 becomes 6 0 7 2.

This leads to the following code for finding the top and bottom limits of the cluster for the current entry:

Step 5 {establish cluster limits}

```
b:=s+1;
while (w[b].left>0) do b:=w[b].bot+1;
b:=b-1; t:=s-1;
while (w[t].left>0) do t:=w[t].top-1;
t:=t+1;
```

It can easily be verified that the loops of step 5 will yield *b=s* and *t=s* when a new cluster is established in an empty part of the working array.

The remainder of the algorithm is dedicated to processing the *RDS*. On encountering a right segment, the following actions need to be performed:

Step 6 {process RDS}

```
for m:=t to b do
   begin
   if (w[m].bot<=s) and (s<=w[m].top) then
      begin
      {insert right limit}
      {output finished box}
      {update working array entry}
      end;
   end;
```

Each action must be performed for each of the working array entries within the cluster. Therefore, all actions are nested within a loop going from the top to the bottom of the cluster as established in the previous step. The test directly following the loop header enforces the right segment condition of *Pinc*. A design consideration at this point is whether the test may be better combined with the do loop in a "while do" loop. This option was not chosen considering the fact that the bottom entries in the cluster monotically increase (monotonic meaning a positive or zero step increment) away from the beginning original starting entry of the cluster in both the bottom and the top direction, while similarly the top entries decrease. Therefore, there may be more than one window where *Pinc* holds. Coping with these window limits would seem much more complex than performing a single test within the loop.

Inserting the right limit for the boxes to be generated from the cluster is simply a matter of inserting the *RDS* number in the right slot of the working array:

Step 7 {insert right limit}

```
w[m].right:=i;
```

With this insertion, the box is complete and can be output in a form depending on further processing to be performed on the box. It may be inspected for size or some related criterion immediately, or it may be stored in an output array for later processing. For the purpose of this paper it will simply be stored in an output array:

Step 8 {output finished box}

```
nb:=nb+1; box[nb]:=w[m];
```

The final step for each working entry in the cluster is to update the status of each entry. For the current entry, corresponding to the strip number of the *RDS*, the box has been completed, and hence the entry can be recycled for the next box. The entry ($m = s$) is therefore marked empty by setting *w[m].left:=0.*

For working array entries above and below the current entry, conditions *Ptop* and *Pbot* must now be enforced, because the strip of the *RDS* now becomes the strip at *top-1* and *bot+1* for the right segment conditions of *Ptop* and *Pbot.* Since *top* and *bot* in both conditions equal the current strip number, for strips above *s (s>m) s = bot+1*, therefore *bot = s-1*, so that *w[m].bot* must be set to *s-1*. Similarly, for strips below *w[m].top* must be updated to *s+1*. In step 9, therefore different actions are taken, depending on whether the entry indicated by the loop index is above, below, or at the current entry:

Step 9 {update working array entry}

```
if s>m then
   begin w[m].top:=s+1; w[m].right:=0; end;
if s=m then w[m].left:=0;
if s<m then
   begin w[m].bot:=s-1; w[m].right:=0; end;
```

This completes the stepwise development. Some overall improvements for efficiency can be made. Note that the *RDS* can be entered directly into the output array so that the working array actually does not need a right element. This results in the following algorithm, where *workarray* only has *left*, *bot*, and *top*, but *outarray* has all four elements:

Boxes Algorithm

```
procedure boxes(strip: intarray; n:integer;
    var w: workarray; var box: outarray);
var i, s, b, t, m, nb: integer;
begin
for i:=1 to n do
   begin
   s:=strip[i]; b:=s+1;
   while(w[b].left>0) do b:=w[b].bot+1;
   b:=b-1; t:=s-1;
   while(w[t].left>0) do t:=w[t].top-1;
   t:=t+1;
   if w[s].left=0 then
      begin w[s].left:=i; w[s].bot:=b; w[s].top:=t;end
   else
      begin
      for m:=t to b do
         begin
         if (w[m].bot<=s) and (s<=w[m].top) then
            begin nb:=nb+1; box[nb].left:=w[m].left;
             box[nb].right:=i; box[nb].bot:=w[m].bot;
             box[nb].top:=w[m].top;
             if s>m then w[m].top:=s+1;
             if s=m then w[m].left:=0;
             if s<m then w[m].bot:=s-1;
            end;
         end;
      end;
   end;
end;
```

NUMBER OF BOXES GENERATED

For practical usage of the algorithm, it is important to have some notion of the number of boxes that are generated as a function of the number of polygon vertices. Each box must be inspected as to its suitability to contain a label, and therefore the total number of boxes must be reasonable.

Each box is generated from within the cluster loop, which is nested within the vertical line segment loop. The number of boxes generated thus depends on the number of vertical segments, as well as the extent of the cluster for each line segment. Both of these factors depend on the geometry of the polygon. For a convex polygon in which there are no duplicate y coordinates (each y generates a unique strip boundary), the number of strips is $N-1$ and the number of vertical segments is $2N-2$. The left segments build up a cluster with $N-1$ entries. The right segments generate boxes. With each *RDS* only as many boxes as the size of the cluster can be generated, but the number is restricted because of the *Pinc* test right after the beginning of the cluster loop, where the influence of this test is determined by the geometry of the polygon. With each *RDS*, the size of the cluster is decreased by one so that for the first right segment at most $N-1$ boxes can be generated, for the second $N-2$, etc., establishing an upper limit for convex polygons without islands of $(N-1)(N-2)/2$, from which it may be concluded that for convex polygons the number of boxes is $O(N^2)$.

For concave and complex polygons, the strips may contain a total of $O(N^2)$ vertical segments (Preparata and Shamos, 1985). This upper limit is approached in polygons with $O(N)$ spikes, each of which would behave similarly to a convex polygon from which one may conclude that a total of $O(N^3)$ boxes may be generated.

However, the average-case behavior of the algorithm is of more practical interest than the above worst-case complexity. But average-case complexity is a function of the spatial distribution of the vertices and is therefore nearly intractable.

Instead, an analysis was performed on 341 reasonably complex soils polygons with a minimum number of 4, an average number of 77, and a maximum number of 513 vertices. A maximum number of 84,536 boxes for a single polygon was generated for a polygon with 429 vertices, yielding a ratio of *number of boxes*$/N^2$ of 0.46. A maximum ratio of 0.74 was obtained for a polygon with 42 vertices and a minimum ratio of 0.06 for a polygon with 44 vertices. The average ratio was 0.19.

A least-squares fit for the data in the test data set was obtained for the model $y = a(x^b)$ where y is the number of boxes and x the number of vertices, with a resultant correlation coefficient of 0.98 and estimates for a and b of 0.18 and 1.99, respectively. Scatter plots of the residuals did not reveal any remaining trends.

Figure 3 shows a portion of the test data, with labels placed with the algorithm, where the lines in the label respectively represent the polygon number, the number of vertices, the number of strips, the number of vertical line segments, and the number of boxes. Labels that did not fit were replaced with the polygon number, placed within a small rectangle.

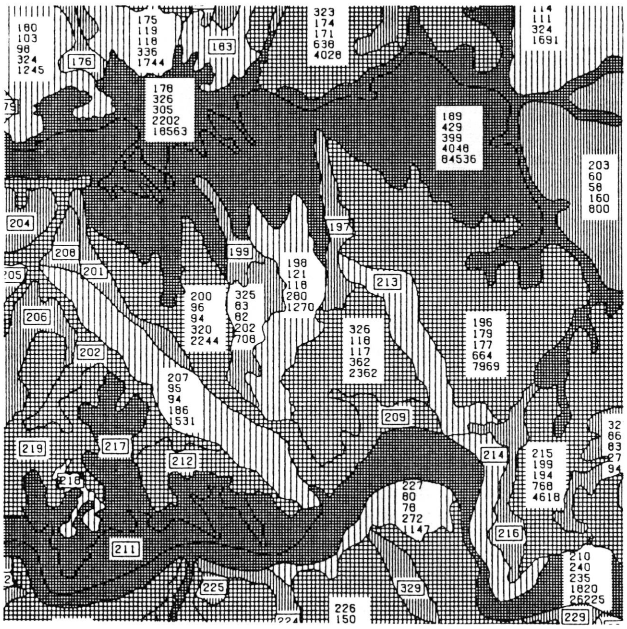

Figure 3.--Portion of test data set with label lines representing polygon number, number of vertices, number of strips, number of vertical line segments, and number of boxes generated.

One problem with the algorithm is the potentially large number of boxes that may be generated. Several solutions can be suggested. Unusually large polygons may be "weeded" to reduce the number of vertices. An alternative is to halt the box generation process when a sufficient number of boxes has been inspected. A third solution is to divide the polygon recursively into smaller polygons, which can be processsed using existing array sizes.

PERFORMANCE

The performance of the algorithm is closely related to the number of boxes generated. Time for the main loop is proportional to the number of vertical segments. For each left segment the cluster limits must be found. Because previous top and bottom information is used, not all cluster entries have to be inspected. Therefore, processing for a left segment is certainly $O(N^2)$ for convex and $O(N^3)$ for other types of polygons, because of the arguments used for the upper limits of the number of boxes. For each right segment the cluster limits are established, and then the cluster is scanned from top to bottom. An alternative arrangement would be to replace the top-to-bottom cluster loop with two loops scanning out from the current entry, while testing for the cluster limits. This would dispense with finding the cluster limits beforehand with the *while do* loops. This approach was tried for the test data set, and timings were performed, but no difference in performance could be detected. In both cases however, performance for the right segment would also be $O(N^2)$ and $O(N^3)$ for the two types of polygons, so that these limits also represent the worst-case complexity for the entire polygon.

Storage for the algorithm is the working array which must have as many entries as the number of strips (+1) so that basic storage is $O(N)$. Although the algorithm as presented stores the completed boxes in an output array, storing the completed boxes may not be practical, given the large number of boxes that may be generated. Therefore, since output storage is not a requirement because boxes may be inspected immediately, storage for the algorithm is *O(N).*

SELECTING LABELLING BOXES

The algorithm may produce a large number of candidate labelling boxes from which only one or a few need to be selected to actually place the label. If only one box is to be considered, one might select the box with the maximum area. Each box, as it is generated, can then be inspected for area, and if the current box is larger than the previous largest box, it becomes the largest box, and so on. However, a much better visual result is obtained when the height-width ratio of the box somehow matches the height-width ratio of the minimum bounding rectangle of the block of text to be placed. Therefore, to select a better box, one may search for the maximum of a function of both total area and the aspect ratios of the label and the box. The following function has provided good results, and was used for figure 3:

$$f(a, rl, rb) = a\ e^{-|(rl-rb)|\ 0.4}$$

where a is the area of the box, rl is the height/width ratio of the label, and rb is the corresponding ratio for the box under consideration. This function modulates area according to the differences of the aspect ratios.

More complex strategies are in order for placing multiple labels. The number of labels might be determined based on the total area of the polygon, the area-perimeter ratio of the polygon as an index of the sinuosity of the polygon, the size and aspect of the label, etc. An additional problem with multiple labels is that the distance between labels probably should be optimized as balanced against the size of the selected boxes leading to the consideration of mathematical programming techniques.

REFERENCES

Cromley, Robert, G., 1986, A spatial allocation analysis of the point annotation problem: Proceedings of the Second International Symposium on Spatial Data Handling, International Geographical Union, p. 38-49.

Langran, Gail E. and Thomas K. Poiker, 1986, Integration of name placement and name selection: Proceedings of the Second International Symposium on Spatial Data Handling, International Geographical Union, p. 50-64.

Mower, James E., 1986, Name placement of point features through constraint propagation: Proceedings of the Second International Symposium on Spatial Data Handling, International Geographical Union, p. 65-73.

Preparata, P., and M. Shamos, 1985, Computational Geometry; Springer-Verlag, New York.

Sedgewick, Robert, 1983, Algorithms; Addison-Wesley, Reading, Mass.