Wm. Randolph Franklin  
Electrical, Computer, and Systems Engineering Dept. 
Rensselaer Polytechnic Institute  
Troy, NY, 12181

A SIMPLIFIED MAP OVERLAY ALGORITHM *

ABSTRACT

This paper presents a new, simpler, algorithm for calculating the overlay graph, resulting from the intersection of the two input maps or planar graphs. The first phase uses an adaptive grid to determine which edges of the first input graph intersect edges of the second. Now all the nodes and edges of the overlay graph are known. The second phase, a planar graph traversal, determines the new polygons and their labels. Separately connected components and nested islands are handled automatically. Because of the separation of the two steps, combining maps with tens of thousands of edges affects only the first step, and the data structures, being simpler, can, if necessary, be implemented in secondary storage without direct access. Optional postprocessing phases can cycle the chains of edges to produce the polygons explicitly, or detect "sliver" polygons and merge them into their larger neighbors.

* This material is based upon work supported by the National Science Foundation under Grant No. ECS 80-21504.
INTRODUCTION

In order to have a complete and useful computer cartography system, we must be able to access a wide variety of databases, perform ever more complex operations on them, and finally display the results. One publicly available world database is 7. If it has too many points, we can reduce it 2. Various packages for displaying the resulting map, in both 2-D and 3-D, are described in 10, 11, 12, 17, 18, 22, 23. We can annotate the map using character fonts 32. These maps can be used in business applications 24 provided we use proper standards to construct them 25, 29. If we wish to scan convert the vector plot to a raster format, algorithms are available 14. The resulting raster frame buffer can be displayed 8, and annotated 30, to a higher apparent resolution than the display device permits. Raster maps are also easy to overlay pixel by pixel. Finally, an excellent description of the general graphic techniques that we need for all this is given in 13.

What the missing from the above list of capabilities is one of the central and most complicated operations, that of overlaying two vector maps to produce a third map, each of whose polygons contains area from only one polygon of each of the input maps. A good existing algorithm is 31, while 20 summarizes various methods.

The new algorithm presented here builds on concepts from computational geometry. Unlike previous methods, it does not
require a band of active chains and a set of partially completed resultant polygons that are updated as each new chain intersection is found. Here, the geometry and the topology are determined in two separate steps. The first, geometric, step intersects all the edges of the two input maps to create new edges and nodes. This step is purely local; the fact that the edges form chains and polygons is irrelevant. A efficient intersection algorithm for 50,000 edges has been implemented. The second, topological, step determines the new chains and polygons. The relevant information at this stage is the nodes and their neighborhoods, so that chains with thousands of edges do not affect this step at all.

The algorithm's steps can be broadly divided into three classes: the execution of some operation independently on all the elements of a set, the external sorting of a set on some key, and the aggregation of certain consecutive elements of a set with a single sequential pass through it. Thus the algorithm can, if necessary, be implemented on external storage with only sequential access.

DATA STRUCTURES

The key to this algorithm is the selection of the data structures. Internally, there are two input maps and one output map that consist of nodes, chains, and polygons. We call the two input maps A and B. A node is a point which is the endpoint of
one or more chains. It contains the following components:

a) a unique i.d. number by which it is referenced,
b) its (x,y) cartesian coordinates,
c) the number of chains that it is the endpoint of, and
d) the ordered (positive rotation) list of the numbers of those chains.

Since nodes may sometimes be accessed by their coordinate, for example to see whether a node exists at a certain position yet, a hash table or related data structure is suitable. A given node number is not used for a node of more than one map. Thus if map A has a node #123, then map B does not. This is also true for chain and polygon numbers, below.

A **chain** is a piecewise linear portion of the border between two adjacent polygons running between two nodes. A chain has a single polygon adjacent along its whole left side and another single polygon along its whole right side. A chain consists of:

a) a unique i.d. number,
b) the i.d. numbers of the nodes that are its endpoints,
c) the number of points in the chain, including the endpoints,
d) the (x,y) coordinates of each point,
e) the i.d. numbers of the left and right polygons, and
f) which input map this chain is from.

The part of the chain between \((x_i,y_i)\) and \((x_{i+1},y_{i+1})\) is
Figure 1: The Nodes, Edges, Chains, and Polygons of a Simple Map

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(3,1)</td>
<td>3</td>
<td>1,2,5</td>
</tr>
<tr>
<td>2</td>
<td>(3,3)</td>
<td>3</td>
<td>5,2,1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chain</th>
<th>End</th>
<th>No.</th>
<th>Coordinates</th>
<th>Left &amp; Right</th>
<th>Adj. Polygons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,2</td>
<td>3</td>
<td>(3,1),(5,2),(3,3)</td>
<td>2,0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1,2</td>
<td>2</td>
<td>(3,1),(3,3)</td>
<td>1,2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1,2</td>
<td>4</td>
<td>(3,1),(1,1),(1,3),(3,3)</td>
<td>0,1</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Internal Data Structure For the Simple Map
called an edge. A polygon is a region of the map whose border is
an alternating sequence of nodes and chains. It has an i.d.
number by which it is referenced, and also various other
attributes that are irrelevant to this algorithm. An output
polygon is eventually labelled with the numbers of the two input
polygons that overlay it. Figure 1 shows an example of this data
structure for a simple map, and Table 1 shows the internal data
structure. For example, chain 1 has two edges: ((3,1), (5,2))
and ((5,2), (3,3)).

The external file format of a map is much simpler. The only
explicit data structure is the set of simplified chains; the
nodes and polygons are determined from this. The external chain
format is:

a) number of points including the endpoints,
b) the list of coordinates of the points, and
c) the numbers of the left and right polygons.

Various special topological cases can arise, that do not
affect the operation of the algorithm. They include:

a) A node may have only two adjacent chains, and in the limit
every chain might have only one edge. However, it is more
efficient to combine two chains if the node between them has
no other adjacent chains.

b) A node may have only two adjacent chains, which are the same
chain. This arises if there is an island.

c) A node may have only one adjacent chain, in which case the "meaning" of this chain is questionable. A regularized set operation 26, 28 would delete such "spikes".

d) A chain that is not a spike may have the same polygon adjacent on both sides. Such a construction may be used to prevent an isolated island.

These cases are shown in figure 2. A good description of some related topology is 21.

PREPROCESSING THE INPUT

The first step is to read the external maps and convert them to the internal format. At this point, we assume that the input is error-free. Cleaning up messy input, which is a necessary part of a production system, will be considered later. The
preprocessing algorithm is:

1. Initialize the data structures described in the previous section.

2. Open the two input map files and read their chains into the chain data structure.

3. (Identify the nodes.) For each endpoint of each chain:

   a) Determine whether there already exists a node with those coordinates. (This can be done in constant time if the nodes are hashed by coordinate.)
   b) If not, insert this endpoint as a new node.
   c) In either case, add the node number to the data structure of this chain.
   d) Add the chain number to the list of chains incident on this node.

There is no need to determine the polygon data structure at this time. If there are too many chains, the data structure can be written to a file as it is determined. If there is insufficient main storage to store even the nodes, then a hash table cannot be used. The following process will identify the nodes instead:

a) As the chains are read, write each node, together with the number of the chain it came from and which end, to a file.
b) Sort the file by the node coordinates.

c) Read the file, noting when consecutive nodes have the same coordinate, and assigning unique nodes sequential i.d. numbers. Write a new file with node numbers, coordinates, and the chains they fall on. At this point we also have the list of chains incident on each node.

d) Sort this file by chain numbers.

e) (Now we have a file ordered by chain number containing the numbers of the nodes at each end). Merge this file into the chain file.

THE OVERLAY ALGORITHM

The actual intersection algorithm uses one new data structure to record a list of the intersections of the chains. Each intersection point causes two records with these components to be created:

a) The number of the first chain that this intersection is on,

b) the number of the edge of this chain where the intersection is,

c) the node number of the intersection point, and

d) the number of the other chain.

The second record records this intersection from the point of view of the other chain. For example, see figure 3 which shows map A (solid lines) overlaid on map B (dashed lines).
Figure 3: Overlaying Map A (solid lines) on Map B (dashed lines)

<table>
<thead>
<tr>
<th>Chain No.</th>
<th>Edge No.</th>
<th>Node No.</th>
<th>Other Chain No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Intersection Data Structure
Ci and Pi mean node #i, chain #i, and polygon #i, respectively. (Ni) represents a new node that is formed where two edges intersect. The intersection data structure for this map is listed in table 2. In this, each chain's edges are numbered from one up, going from the lower numbered node. The overlay algorithm is:

1. Initialize the intersection data structure described above.

2. Find all the intersections of edges of map A with edges of map B. For some fast algorithms, see 3, 4, 5, 15, 16, 27. For the related problem of intersecting chains as chains, see 6. As each intersection point is found, enter it into the node table returning the new node number, and put two new records into the list of intersections. In the example of figure 3, there are four such intersections, nodes 4 to 8. It may happen that a chain intersects a node. In this case, an existing node number is returned from the node table. As before, if there is insufficient main memory to store all the nodes, the new nodes can be uniquely identified in a two step process using sequential files. Also the list of intersections can actually be a sequential file.

3. Sort this list of intersections by the first two components, i.e. chain number and edge number. This will give us all the intersections of each chain in order along it. For example, chain 3 has intersections at nodes 5 and 7. As external sort requiring little memory can be used.
4. (Cut the old chains to make the new chains.) Make a single pass simultaneously through the list of chains and the sorted list of intersections. Cut each chain at each intersection of that chain with chains of the other map. Thus if a chain has N intersections, there will be N+1 new chains. Delete the old chain and insert the new chains. Insert the numbers of the two new chains into the lists of adjacent chains of the relevant nodes. After this step, we have all the nodes and chains of the new overlaid map. This requires that only one chain be in memory at a time. In the example, chain 3 is split into three chains: from node 1 to 7, from 7 to 5, and from 5 to 2. The first new chain has one edge, and the latter two, two edges each.

5. Identify the regions of the new map using a planar traversal algorithm.

6. Determine which polygons of the old map each polygon of the new map corresponds to. There are two cases here:

   a) The chains bordering a given new polygon came from both of the original maps. In this case, the overlaid polygons from original maps A and B are the neighboring polygons for those two chains.

   b) The chains bordering a given new polygon came from only one of the original maps, and so this polygon is just one of the original polygons. In this case, to determine which polygon of the other map completely contains this one, we
must use a planar location algorithm.

**EXECUTION TIME**

Much of the algorithm requires time proportional to the greater of the size of the input or the output maps since it requires sequential passes through that amount of data. The only potentially nonlinear operation is the determination of the edge intersections. If there are \( N \) input edges there may be up to \( N^2 \) intersections, but then the complexity of the output map will also be \( N^2 \) so this is still linear in the size of the output. There could only be a problem if we had to spend \( N^2 \) time to find about \( N \) intersections. This can happen with some of the intersection algorithms described, although it has probability zero if the input is randomly distributed, which it isn't. However, input maps are not arbitrarily uneven, since the distribution of edges satisfies various requirements such as a Lipschitz condition \( ^9 \), i.e. that arbitrarily small areas of the map do not have arbitrarily large numbers of edges. Thus these algorithms have a fast expected speed. Others of the intersection algorithms listed have a slower expected speed but a guaranteed worst case time of \( N \log(N) \). Thus this overlay algorithm is quite fast.

The edge intersection algorithm has been implemented and tested by Varol Akman \(^1 \). The implementation is a Flecs (Fortran preprocessor) program running on a Prime 500 or 750. The largest
example tested had 50,000 small random edges and took a couple of minutes. This intersection algorithm has also demonstrated its usefulness when incorporated into other programs, such as to determine hidden surfaces. The planar graph algorithm has also been implemented and tested.

ERRONEOUS INPUT

This method can be extended to validate the topological and geometric consistency of the input. Some common errors, how they are detected, and how they may be repaired are:

a) Chains from one map intersecting themselves or each other: This is detected if all edges are tested against each other, instead of only testing edges of map A against those of map B. A semi-automatic repair is to add new nodes and smaller chains and polygons if this happens. Then the small polygons can be merged with their larger neighbors.

b) Chains not meeting at a node: This will cause there to be nodes with only one incident chain. One repair is to search for another nearby node and merge the two. However, since this moves one endpoint of the incident edge, it may cause a chain self-intersection error.

c) Chains completely missing: If the polygons of the planar graph corresponding to this map are calculated, then one
polygon will appear to have two different i.d. numbers on
different chains of its border. This can be reported to the
user and the polygon i.d.'s made the same to allow the run to
proceed and produce at least a partial output.

d) Erroneous adjacent polygons specified in chains: This will
have similar symptoms to the previous case.

e) Glitches in the point coordinates: They will probably cause
chains of a map to intersect themselves. Many glitches can
also be caught by looking for two adjacent long edges whose
other endpoints are close. The midpoint can be replaced by
the average of the two other points.

It is possible to introduce self-intersection errors in an
originally correct map by generalizing or reducing the chains.
The above procedures can be used to clean up the reduced map.

COINCIDENTAL INPUT

Chains from input map A almost coinciding with parts of
chains of map B may occur. For example, if map A is the nations
of South America, and map B is the river drainage basins, then
part of the border between Argentina and Chile will occur on the
river basin map. The two chains will not have the same
endpoints, but will approximately overlay each other for part of
their length. Many small "sliver" polygons will result from
overlying these two maps, and the more accurately the maps are digitized, the more will result\textsuperscript{19}. Before giving a method for deleting them, we must first characterize a sliver. One possibility would be any polygon whose area is smaller than a certain value, but a better is to determine the polygon's smallest "diameter" or the closest pair of parallel lines that enclose it.

Slivers will be deleted by coalescing them with neighboring big polygons. It is not sufficient to merely combine a sliver polygon with one of its neighbors since then several sliver polygons might coalesce to form a non-sliver polygon. Thus, the algorithm for removing sliver polygons by coalescing them with their neighbors is:

1. For each polygon, calculate its measure of "sliverness", be it area, smallest diameter, or whatever.

2. For each sliver polygon, find the longest chain bordering it that has a non-sliver polygon on the other side. Coalesce the two polygons by deleting this chain and renumbering the adjacent polygon fields of the other chains adjacent to the former sliver polygon.

3. Repeat step 2 for as long as there is a sliver polygon adjacent to a non-sliver polygon. Because of the coalescing, a sliver polygon that is originally adjacent only to other slivers will eventually be made adjacent to a non-sliver,
unless this whole connected component of the graph is nothing but slivers.

4. If this whole component is nothing but slivers, delete it entirely.

SUMMARY

It is possible to perform a complicated global operation, such as overlaying two maps or planar graphs by means of mostly local operations that use simple data structures, require less main memory, and still execute fast. It is not necessary to thread along the chains to form the new map.

REFERENCES


