Calculating the Area of Overlaid Polygons Without Constructing the Overlay

Wm Randolph Franklin
Venkateshkumar Sivaswami
David Sun
Mohan Kankanhalli
Chandrasekhar Narayanaswami
Rensselæ r Polytechnic Institute
Troy, NY, 12180 USA

Phone: +1 (518) 276-6077,
Fax: +1 (518) 276-6261
Internet: wrf@ecse.rpi.edu

June 13, 1994

Abstract

An algorithm and implementation for calculating the areas of overlaid polygons without calculating the overlay itself, is presented. OVERPROP is useful when the sole purpose of overlaying two maps is to find some mass property of the resulting polygons, or for an areal interpolation of data from one map to the other. Finding the areas of all the output polygons is both simpler and more robust than finding the polygons themselves. OVERPROP works from a reduced representation of each map as a set of "half-edges"; with no global topology. It uses the uniform grid to find edge intersections. The method is not statistical, but is exact within the arithmetic precision of the machine. It is well suited to a parallel machine, and could be extended to overlaying more than two maps simultaneously, and to determining other properties of the output polygons, such as perimeter or center of mass. OVERPROP has been implemented as a C program, and is very fast. The execution time on a Sun Sparcstation 10/30 to combine US counties, with 55068 vertices and 2985 polygons with hydrography polygons, with 76215 vertices and 2075 polygons, was 19 CPU seconds, excluding I/O time. This included the subtask of locating all the points

of each map in a specific polygon of the other, which is a useful application in its own right.

1 Introduction

In the map overlay problem, two maps, or planar graphs, are combined to make a third, each of whose polygons represents the intersection of one polygon from the first input map with one from the second. For instance, if the first map's polygons are countries and the second watersheds, then one output polygon might be that part of Canada that drains into Hudson Bay. The overlay problem is one of the more difficult computational issues in GIS since the algorithm is complex, the data are large, and the numerical inaccuracy of computers can confound even correct algorithms, White, 1977, White et al., 1989. Thematic map overlays were used by the Irish Railway Commissioners in about the 18 century, Bartlett, 1993. For a history of other manual methods, see Coppock, 1988. The first operational GIS to use overlay was Roger Tomlinson's CGIS. There have been several theses on overlay, Aronson, 1982, Guevara, 1983, Lam, 1977, Wagner, 1987, Wagner, 1991. See Marble, 1990 for some of the broader implications of overlay.

Often the overlay operation is performed only to find some property of the output polygons, such as their area, or even just to list the non-empty ones. The polygons' actual vertices and edges are not used otherwise. Nevertheless, the usual method is to find the overlay polygons first, and then to calculate the desired properties.

For example, assume that we need the population of each hydrographic polygon in the USA, but have only the population of each county. One reasonable method to find the population of one particular hydrographic polygon is areal interpolation, as follows. Find each county that it overlaps, the overlapping area with each county, and then the percentage of each county that is overlapped. Assign the same percentage of the county's population to the hydrographic polygon. This does assume that each county's population is uniformly distributed, which is false, but perhaps approximately correct.

Our algorithm, OVERPROP, presents a simpler method for

finding properties of overlay polygons. It is much more efficient, can process the largest databases, and is less sensitive to numerical errors since they cannot cause topological difficulties. It is based on our simpler, local topological data structures for representing polygons and polyhedra. Its speed comes from the uniform grid data structure. These concepts have been described in the context of computer aided design (CAD) most recently in Franklin et al., 1990. A preliminary description of this algorithm and implementation were presented in Franklin and Sivaswami, 1990, Franklin, 1990; this paper contains tests of an improved implementation on larger maps and more details. Wu and Franklin, 1990 have done overlaying with rational numbers in Prolog, and Sun, 1989 and Sivaswami, 1990 have looked at related issues. One parallel implementation of the uniform grid is Hopkins and Healey, 1990.

OVERPROP is fast enough to be used dynamically during a data investigation. It is not necessary explicitly to overlay all the data layers at the start of a job, as some commercial GIS packages do, unless the complete overlay is needed for other reasons, such as display, which is often the case.

Monte Carlo methods could be used approximately to calculate overlay areas. However, the accuracy improves with the square root of the number of sample points, which makes high accuracy infeasible. Also Monte Carlo methods do not extend to determining other properties, such as perimeter.

The rest of the paper is as follows. First, we will see the logical data structures, and the theoretical formulæ underlying the algorithm. We give the algorithm in two stages. First, a brief conceptual description ignores efficiency considerations. Then we see a more detailed description. Finally, we see several deeper issues, such as controlling numerical errors, design tradeoffs, and several possible extensions.

2 Major Data Structures

Conceptually, each input map is a planar graph composed of vertices and edges partitioning the two-dimensional plane into polygons. The graph may be disconnected with the components either nested or disjoint. Each polygon has a name, or

identification number, which may be non-unique when one logical region is composed of several polygons, such as two parts of Michigan state. OVERPROP neither knows nor cares, but returns the total area of all of them. By convention, the outside of the map is polygon #0.

In our map data structure, the polygons are not represented explicitly, but could be recreated if desired since each edge knows its neighboring polygons. We assume that each input map is a set of vertex coordinates: (x, y), and a set of tagged edges: $\{(v_1, v_2, p_l, p_r)\}$, where v_1 is the vertex that is the edge's first endpoint, and v_2 is the second. (p_l, p_r) are the names of the polygons to the left and right.

In fact, this program reads chains of points in the Harvard Odyssey CDB format, and immediately splits them into the individual edges. See section 7.1 below for more details.

Each edge of each polygon has two endpoints (vertices), each of which defines a "half-edge", *e*. Since a data structure edge borders two polygons, it decomposes into four half-edges. Although the half-edges are not explicit data structure elements, they can be derived easily, and are important in the following algorithm. A half-edge contains the information (v, t_x, t_y, n_x, n_y). v is the number of the vertex that is the endpoint. (t_x, t_y) is a unit tangent vector from the endpoint along the edge, and (n_x, n_y) is a unit normal vector perpendicular to the tangent, and pointing into the polygon. For example, see edge AB in figure 1. The edge is $\sqrt{2} \approx 1.4$ long, so the unit tangent from A to B is $(0.7, 0.7)$. The unit normal to that, pointing into polygon P, is $(-0.7, 0.7)$. Therefore, the half-edge for the P side of the A end of the edge is $(\mathcal{A}, 0.7, 0.7, -0.7, 0.7)$. The other three half edges are the Q side of the A end: $(A, 0.7, 0.7, 0.7, -0.7)$, and the P and Q sides, respectively of the B end: $(\mathcal{B}, -0.7, -0.7, -0.7, 0.7)$ and $(B, -0.7, -0.7, 0.7, -0.7)$.

A polygon number of the output map is an ordered pair (p_1, p_2) of the two input polygons whose intersection forms this particular output polygon. Internally, to save space, the hash-addresses of the input polygons are used instead of their id numbers.

3 Theoretical Basis of the Algorithm

The algorithm is built on the following principle. Many properties of a polygon, such as area, center of mass, and moments of inertia, may be calculated separately for each half-edge of the polygon and then summed. These are *extensive* properties in a thermodynamic sense, in that the total value for an object may be found by integrating over its area or volume. This concept is also related to Greene's theorem in calculus, where an integral over an area is transformed into an equivalent integral over the boundary of the area. Formally, let polygon P have the set of half-edges E. Let $F(P)$ be some desired function of the polygon, such as area. Then for these F, there exists a functional that returns a new function f_F such that

$$F(P) = \sum_{e \epsilon E} f_F(e)$$

For example,

$$f_{area} = \frac{1}{2}(v_x t_x + v_y t_y)(v_x n_x + v_y n_y) \qquad (1)$$

and

$$f_{perimeter} = -(v_x t_x + v_y t_y)$$

The basis of these formulæ is described in more detail in Franklin et al., 1990, Franklin, 1987. However, the idea can be seen from figure 2, where $\triangle ABC$ is a polygon whose area we wish to determine. We partition it into six triangles by dropping perpendiculars from the origin, O, to each side. Each smaller triangle, such as $\triangle AFO$, is determined by the local topology of the external edge AB at the vertex A. That is, $\triangle AFO$ is completely determined by the location of A (relative to O) and the direction of B relative to A. The total area of $\triangle ABC$ can be found by summing the areas of the six smaller triangles. However, the area of each smaller triangle is one term of the formula 1.

So if we can find the half-edges of the output polygons we can find the areas. The output half-edges needed by the formulæ are of two types,

- those derived from an endpoint of an original edge of one of the input maps, or
- those derived from an intersection of two input map edges.

For each original endpoint we also need to know which polygon of the other map this point is in. For each intersection of two edges we already know the names of the relevant polygons.

4 Brief Algorithm

A sketch of the algorithm, ignoring efficiency considerations, now appears as follows.

1. Initialize a data structure to contain a list of triples, $(\mathcal{P}, \mathcal{Q}, partial - area)$, to accumulate the areas of the output polygons. $\mathcal{P}$ and $\mathcal{Q}$ are the names of the polygons from the first and second input maps respectively.

2. Find all intersections of any edge of map 1 with any edge of map 2. Each intersection will generate 8 half-edges, as shown in figure 3.

3. For each half-edge resulting from an intersection, find the tangent and normal vectors and the relevant polygons from each input map. Note that in figure 3 there are two polygons from map 1, one on each side of edge 1, and also two from map 2, one on each side of edge 2. Each of the 4 possible pairs of the input polygons will apply to 2 of the 8 half-edges. For example, half-edges 1 and 5 are part of the output polygon that is the intersection of the polygon on map 1 above edge 1 with the polygon on map 2 to the right of edge 2.

4. Apply the formula to each half-edge to calculate its contribution to the relevant output polygon's area.

5. Now we must calculate and process the half-edges not resulting from an intersection. For each endpoint of each edge of map 1, determine which polygon of map 2 contains it. Repeat for the edges of map 2.

6. Split each edge of each input map into four half-edges.

7. Apply the formula to these half-edges and store their partial areas.

8. Extract the final area of each non-empty output polygon, which was the goal.

5 Detailed, Efficient, Algorithm

The algorithm presented in the last section is of value only if it is efficient enough. For instance, if finding all intersections of input edges takes quadratic time in their number, then maps with 10,000 edges, and thus 50,000,000 possible intersections, will be difficult to process. This section tells how to make the algorithm practical.

5.1 General numerical speed

On many computers, floating point computations are slower than integer, sometimes by a factor of three or more. Initially, when using 4 byte integers, we scale the map coordinates to the range $[-M, M]$ in X and Y, where M=20,000. This allows us to calculate edge equations exactly since there is no overflow in calculating the numbers in the equation $(y_2 - y_1)x + (x_1 - x_2)y - x_1y_2 + x_2y_1 = 0$; the largest number is under 800,000,000. Then we can substitute a point into the equation to determine which side of the edge it is on, again without any roundoff, since the largest number computed will be absolutely less than $4M^2 = 1,600,000,000$.

Nevertheless on some machines, floating point computations are now as fast as integers, so this step may not be as necessary in the future. Indeed, the Intel i860 processor does not even do integer multiplication. It multiplies two integers by treating them as floats to save space on the chip.

5.2 Using an expandable array to store the edges in each cell

We use a square array with one expandable array per cell, as shown in figure 4, to store which edges of each map pass thru each cell. Section 7.4 lists other possibilities. The expandable array data structure has the following properties.

- The grid is a square array of pointers, one per cell.
- Each pointer is initially null.
- When the first edge is inserted into a cell, then space for, say, three edges is allocated, and the cell pointer made to point to it. In C, the allocation would be done with the `malloc` routine. We must also allocate two counters to record the number of edges of each map in this cell.
- If a fourth edge is inserted into the cell, then reallocate a bigger array, and copy the old data over. In C, the `realloc` does all this in one operation. The efficiency question is how often this reallocation occurs. An example shows that this is not often. Imagine that every time we overflow, then we reallocate the array with twice the size. Now, add one million edges one by one. There will be only 20 reallocations. The average edge will be copied only once. In fact, one edge will be copied 20 times, 2 edges 19 times, ..., 250000 edges once, and 500000 edges not at all.

  In general, if the array size is grown by a factor α each time (here $\alpha = 2$) from one element to a large size, then each element is copied an average of $1/(1 - \alpha)$ times (here 1).

This method does not waste space for multiple pointers or cell numbers, but requires only two extra words per non-empty cell, for the two counters. We can also randomly access edges in a cell. In fact, in the program, the syntax is that of indexing an array. Unlike linked lists, a popular alternative, expandable arrays keep the data for each cell contiguous, so that fewer disk accesses are required when paging the virtual memory that the allocator would use when real memory was insufficient. If the available space is so marginal that the expandable array overhead is intolerable, then one time versus space tradeoff is as follows. Modify the program to run once to accumulate only the number of edges in each cell, but not their names. Then allocate

exactly the required space for each cell, and run the program again to do the calculations.

We do need space for the whole cell matrix even if most cells are empty. This space could be reduced by using a hash table.

5.3 Finding intersections

We use Franklin's uniform grid method Franklin et al., 1990, Franklin et al., 1989. It has these properties:

- simplicity, which implies ease of implementation,
- parallelizability,
- a typical time that is linear in input+output size,
- high speed, and
- exact output, since the grid is only a scaffolding to improve the speed.

For typical grid resolutions, the number of grid cells is proportional to the input+output size, so traversing the cell matrix does not affect the asymptotic time. The grid resolution, typically 100×100 has no relation to the resolution of the coordinate representation, here 20000.

The common objection is that the uniform grid is too simple for irregular data, since an adversary can easily produce bad input by placing many edges in one cell, to cause the pair-by-pair comparison of all the edges in each cell to take too much time. The answer is that in practice, it works, quite well, unless there are orders of magnitude variations. The robustness with respect to different choices of grid size is exemplified by using it to find all edge intersections in the United States Geological Survey Digital Line Graph sampler tape, for Chicamauga, Tennessee, which has both urban and rural road densities.

In that database, there are 116,896 edges, of an average length of 0.0023 of the map size. The edges' lengths are very skewed, with the standard deviation at 0.0081, or four times the average. There are 144,666 intersections in all, 135,875 of them where the endpoints of edges touch. We tested a large range of different grid resolutions, and observed a factor of three

variation from the optimum grid size to cause only a 20% to 50% increase in time. This is a factor of 100 in the number of grid cells between the low and high grid resolutions, wherein the time remains reasonable. I.e, choosing the exact, optimum, grid size is not critical. This also explains why uneven data can be handled. A grid chosen for the average edge length and density will be different than the optimum grid for the data in specific regions of the scene; however so long as it is within a factor of three of the optimum grid resolution in that region, there is no problem. Figure 5 shows how the CPU time varies with the grid size when finding all the edge intersections in this database. As the grid gets finer, it takes more time to insert the edges into the cells. However, since each cell has fewer edges, processing the cells is quicker, until there are so many cells that the fixed per-cell overhead starts to dominate.

5.4 Locating the polygon containing each edge endpoint

The naive method compares each point against each edge of the other map. However, using a uniform grid again, this can be done in expected time $\theta(N + M)$ if there are N points to test against a map with M edges. (The notation $\theta(N + M)$ stands for any quantity that grows proportionally to $N + M$ as they grow. Since it ignores constant multiplicative factors, like $2(M + N)$ or $10(M + N)$, it expresses something that remains valid as the computer hardware changes. Of course, this notation becomes useless if it hides a factor that is too large.)

We use an optimized version of an implementation of a one-dimensional grid, or slab, method by Sivaswami, 1990. Sample execution times are part of the example below. 55,068 points were each located in a map with 2075 polygons defined by 76,215 vertices. The location time on a 1993-vintage Sun 10/30 Sparcstation workstation was 2.33 CPU seconds, or 42 microseconds per point. There was also preprocessing time, but that is impossible to separate from the rest of the overlay processing.

If the user is feeling pessimistic about the data, and worried that the uniform grid might fail, then there do exist more complicated but worst-case optimal methods, such as section

2.2.2, *Location of a Point in a Planar Subdivision*, of Preparata and Shamos, 1985, or Sarnak and Tarjan, 1986. A typical time for these might be $\theta((N + M) \log M)$. However, they are more difficult to implement. Some newer techniques work by using a random sample of a subset of the data to build a structure used for the point location. Sometimes the data structure is designed to be dynamic, so that if some of the polygons change, it does not need to be completely rebuilt. These ideas are presented in Agarwal, 1991, Tamassia, 1991, Stewart, 1991, Lee and Preparata, 1977, and Preparata and Tamassia, 1989. However, many of these algorithms may have yet to be implemented.

5.5 Creating the one dimensional slab structure

This section describes our point location algorithm in more detail.

1. Each slab is one column of cells from the same grid used later for edge intersection. The edges in the cells in a column are gathered together and sorted primarily by their minimum Y coordinates, and secondarily by their ID number. This comparison predicate assures that two edges will not compare the same unless they are the same edge. This results in duplicates of the same edge being adjacent after the sort, so they can be deleted. Duplicates occur when the same edge falls in more than one cell in the column.

2. Finally a selection sort is performed on the unduplicated edges in each slab. The criterion is the partial order where one edge is less than another if, from a viewpoint located at $y = -\infty$, the former edge (at least partly) obscures the latter one. A partial order is a relation that is not defined for all pairs of objects. E.g., if we have a group of objects, some of whom hide others, then “hiding” is a partial order. This partial order was used early in Prism-map by Franklin, 1978. After this step, the edges in each slab are ordered so that if one edge at least partially hides another, then the latter edge is later in the list.

3. To determine the polygon containing a point, first the slab containing the point is determined. Then, the location of the point in the partially ordered edges is determined, and

a ray is fired up from the point to the first edge that it hits, in the order that the edges are stored in the slab. Then the point is in this edge's left or right neighboring polygon depending on whether the edge's first endpoint is to the right or left of its second endpoint, respectively.

5.6 Degenerate cases

These occur when a vertex of one map falls on an edge or vertex of the other. Although this has previously been a very difficult problem, there is now a complete theoretical, and practical, solution to the problem of degeneracies, the Simulation of Simplicity technique of Edelsbrunner and Mücke, 1988. Briefly, this pretends to add an infinitesimal to the coordinates of the second map. By definition, all first order infinitesimals are less than all positive finite numbers. Different orders of infinitesimals may be used; all second order ones are less than all first order ones, etc. A delightful book on this is by Knuth, 1974. The effect of infinitesimals is to prevent any comparison tests from returning an equality. We do not actually store infinitesimals, but instead determine what their effect would be on any test, and code a modified test accordingly. The modified test is longer but generally not significantly slower. For example, suppose that we are testing a point (x, y) against a line $ax + by + c = 0$, and that we are using simulation of simplicity to pretend to shift the point by (ϵ, ϵ^2). Thus testing the point $(x + \epsilon, y + \epsilon^2)$ against $ax + by + c = 0$ goes as follows. Note that so long as (x, y) is not on the line, then the same answer is returned as before.

```
d=ax+by+c;
if (d ≠ 0) then return signum(d);
else if (a ≠ 0) then return signum(a);
else return signum(b);
```

`signum` returns –1, 0, or 1 according to the sign of its argument. Note that the extra tests are called only in the degenerate case. Now, one might deduce this particular test without resort to simulation of simplicity. However this technique provides a general, theoretically well-founded, method for creating all such tests.

6 Error Control

Numerical roundoff errors, and the numerous sliver polygons generated even if there is no roundoff, help make the polygon overlay problem so hard. With OVERPROP, slight errors in calculating intersections vertex coordinates will cause proportionally slight errors in the areas. OVERPROP also does not care about roundoff errors that cause one chain of edges to cross another erroneously as shown in figure 6. This will either cause the output to be off by an amount proportional to the geometric error in spite of this topological error, or it will create a spurious new object and report it with an insignificant area.

However, although OVERPROP does not explicitly use the topology of the polygons, it implicitly assumes that it is topologically clean, or consistent in the following ways.

- There are no missing edges, and no gaps between edges of a polygon.
- The polygon names stored with the edges are all correct.

In fact the sensitivity of OVERPROP to these input errors may be used approximately to test for consistency by the following method.

1. Repeatedly randomly translate and rotate the input maps.
2. Find the areas with OVERPROP.
3. If the answers are different by more than numerical roundoff, then the input is bad. If the answers agree after several random operations, then the input is almost certainly internally consistent.

The maximum roundoff error can be determined, and spurious output polygons will have areas under this. Multiple random tests such as this have been used in other applications, such as primality testing. Nevertheless, a complete test requires analyzing the topological structure.

7 Design Tradeoffs and Choices Not Taken

The data structures and algorithms used in OVERPROP were selected from many possibilities. Factors considered included efficiency of execution and efficiency of implementation, i.e. simplicity. Some decisions included the following.

7.1 Storing the vertices in a separate array and having the edges contain vertex numbers instead of vertex coordinates

This is the choice between immediate versus indirect data. Storing the vertex coordinates immediately in the edges makes processing the edges cheaper. Depending on wordlengths, the storage will be the same or greater. If a vertex position takes 2 bytes each for x and y, and a vertex number takes 4 bytes, then the storage is identical. On the other extreme, if each coordinate takes 4 bytes, but a vertex number takes only 2, then the indirect method will be smaller. With memory prices for workstations at $40 per megabyte and falling, then memory is not so important as before. However, big programs will always execute more slowly than otherwise equivalent small programs because of bus bandwidth limitations, and because the big programs will use the cache less efficiently.

One problem with not storing vertices separately is that then we would not be able to store associated information, such as what polygon of the other map contained which vertex, and would need to recompute this for every edge on that vertex.

7.2 Using integers instead of reals for coordinates

This saves space and execution time. However, attention must be paid to scaling since that is no longer automatic. Also, some current workstations process reals as fast as integers in some cases. Scaling with either reals or integers may cause intersections to vanish or appear, although this will happen

more with the coarser-grained integers. See Franklin, 1986 for some problems with integers in graphics. Here, the errors in the areas will be on the order of the square of the resolution. One advantage of OVERPROP is that it is less sensitive to these errors since it uses less topology.

7.3 Splitting chains into separate edges

This choice makes the data structure bigger, but each data element is now fixed length instead of variable. This is a similar concept to a relational database satisfying Codd's third normal form Ullman, 1988. Also the algorithm is simpler with fixed length items, and parallelizes better. Whether this was the correct choice depends on one's tolerance for complexity.

7.4 Storing the (cell, edge) information

The number of edges in a cell is quite variable, as shown in figure 8. Most cells may be empty, and most of the remainder have only one edge. However, a few cells may have many edges. What data structure is appropriate then? In addition to the expandable array actually used, as described in section 5.2, there are several possibilities, as follows.

- *Maintain a linked list of the edges in each cell.* This allows the number of edges in a cell to grow without a specific limit. However, the pointers waste 50% of the space, the random allocation of elements may thrash virtual memory, and we can't randomly access the edges in a cell, but must follow the list in order.

- *Append to a common* (cell,edge) *list, and then sort it to bring together the edges in each cell.* This also allows a variable number of edges per cell, and also allows the edges in each cell to be accessed in random order. However this still wastes 50% of memory, storing the same cell numbers again and again in many different list elements. Also, many

system sort routines are surprisingly bad, partly because they are written to be so general, so the user might need to implement this himself for efficiency.

8 Implementation and Testing

We implemented OVERPROP as C program, and present times on a Sun Sparcstation 10/30 running Unix The test case overlaid the following two maps, as shown in figure 7.

Map Name	Vertices	Edges	Polygons	Chains
US Counties	55068	46116	2985	8941
Hydrography Polygons	76215	69835	2075	6380

The following times are for a 500×500 grid. Reading the data is so slow because we stored the maps in an ASCII format for flexibility. For comparison, simply reading the two megabyte county file takes under 0.1 CPU second, while copying it to another file takes about 0.3 CPU second. Therefore if the input were prepared in a binary format compatible with the program's internal data structures, then the reading time would practically vanish. Of course, other parts of the program could be sped up also.

The accumulate-areas step refers to applying the half-edge formulæ to the new vertices, once they have been found and classified.

Operation	CPU Time (seconds)
Read map data	40.30
Scale vertices	0.25
Extract edges from chains	0.80
Calculate input polygon areas	1.23
Make grid	0.37
Insert edges into grid	3.03
Intersect edges	2.43
Locate map 1 vertices in map 2	2.33
Locate map 2 vertices in map 1	3.17
Accumulate output areas	5.05
Print the areas	9.00
Total CPU time	67.97
Total excluding I/O	18.67

The efficiency of the grid for edge intersections is shown by the fact that of 3,514,730,940 possible edge pairs, only 168,781 pairs were tested, and of those, and 11,207 different intersections were found. Another 7307 duplicate intersections were found and discarded. Duplicates occur when a pair of intersecting edges both pass thru the same several cells.

A common question relates to the uniformity of the data in the cells. An uneven distribution would seem to be inefficient. This problem is worse as the grid gets finer, the data gets more granular, and the distribution less uniform. In fact this has never been a problem for any real data ever tested, either in overlay-area calculation, or in other related programs in computer aided design. Although bad examples are easy to fabricate, the grid structure is tolerant of the amounts of variation that occur in real data.

In this example, with the 500×500 grid, the worst cell had 25 edges. 60% of the cells were empty, and another 13% had only one edge, so 73% of the cells required no intersection testing. 26,383 cells had so many edges that they had to have their space increased. The histogram of number of cells containing each number of edges is shown in figure 8.

9 Possible Extensions

9.1 Producing the Overlay Polygons

Sometimes we need the output polygons themselves, not just their areas. Note that we already determine the vertices, and the direction from each vertex to the next. It is conceptually simple to link the adjacent vertices together and so have a complete map overlay system. Of course, there are practical problems, such as numerical roundoff, to consider, otherwise topological errors, such as the wrong vertices being linked, could cause the output to be meaningless. Producing the overlay polygons themselves would probably double the size and execution time of the program.

9.2 The Cross-Area Problem

One major application of map overlay is to interpolate data from one map to another. Let polygon $\mathcal{P}_i$ of map 1 have population p_i. Assume that we want the population of polygon $\mathcal{Q}_j$ of map 2. Let the area of $\mathcal{Q}_j$ intersected with each $\mathcal{P}_i$ be a_{ij}, and let the area of each $\mathcal{P}_i$ be a_i. Then an areal interpolation to the population of $\mathcal{Q}_j$ is

$$\sum_i \frac{a_{ij}}{a_i} p_i$$

That assumes that the population is evenly distributed within each polygon of map 1. This is common practice, since it's difficult to do better without approximating the population by a smooth surface.

Note that here we need only the areas of the overlay polygons, not the polygons themselves.

9.3 Overlaying multiple input maps simultaneously

These ideas extend to finding the areas of the results of overlaying more than two input maps simultaneously. As with

two maps, the output half-edges are derived from input edge endpoints and intersections of edges. What is new is that the location of each half-edge must be determined in every input map. In this case, the advantage of OVERPROP compared to actually finding a sequence of more and more complicated intermediate maps would be even greater. There would also be no artifacts resulting from the order in which the several input maps were processed, since they would all be used together.

9.4 Three Dimensional Overlaying

In three dimensions, we may have a solid object that is partitioned into tetrahedra in two different ways. The ideas of this paper can be extended into finding the volumes of the all the non-empty intersections of any two tetrahedra Franklin and Kankanhalli, 1993. This could be useful in processing geological databases. Implementing this 3-D algorithm should be feasible, although harder than doing two dimensional case.

10 Summary

OVERPROP very efficiently calculates the areas of the intersection polygons resulting from overlaying two maps. Unlike other polygon overlay algorithms, it uses no explicit global topology. There is no tracing chains of edges. Polygons completely contained inside other polygons are not a problem, the correct answer is produced with the area of the outer polygon excluding the inner area. Polygons may also have multiple separate components. These ideas are also applicable to implementation on parallel machines.

OVERPROP is a result of an investigation into how little topology we actually need explicitly to store, and concurrently, by how much can special cases be reduced. Sometimes a more complete topology is needed, for instance to draw the output polygons, but for mass calculations it is not.

11 Acknowledgements

This work was supported by NSF Presidential Young Investigator grant CCR-8351942, and by NSF grant CCR-9102553. Partial support for this work was provided by the Directorate for Computer and Information Science and Engineering, NSF Grant No. CDA-8805910. We also used equipment at the Computer Science Department and Rensselæ r Design Research Center at RPI. Part of this work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to DARPA and the Air Force Systems Command, Rome Air Development Center (RADC), Griffiss Air Force Base, NY, under contract # F306002-88-C-0031. Part of the research reported here was made possible through the support of the New Jersey Commission on Science and Technology and the Rutgers University CAIP Center's Industrial Members. Finally, the referees' many detailed and constructive comments greatly improved this paper.

References

Agarwal, P. K. (1991). Geometric partitioning and its applications. In Goodman, J. E., Pollack, R., and Steiger, W., editors, *Computational Geometry: Papers from the DIMACS special year*. Amer. Math. Soc.

Aronson, P. B. (1982). A comparitive analysis of three polygon overlay algorithms. Master's thesis, State University of New York at Buffal.

Bartlett, D. (1993). Overlay analysis and GIS. Usenet posting to comp.infosystems.gis. 14 Dec.

Coppock, T. (1988). The analogue to digital revolution: A view from an unreconstructed geographer. *American Cartographer*, 15:263–275.

Edelsbrunner, H. and Mücke, E. P. (1988). Simulation of simplicity: A technique to cope with degenerate cases in

geometric algorithms. In *Symposium on Computational Geometry*, pages 118–132.

Franklin, W. R. (1978). Combinatorics of hidden surface algorithms. Technical Report TR-12-78, Center Res. Comput., Harvard Univ., Cambridge, MA.

Franklin, W. R. (1986). Problems with raster graphics algorithms. In Kessener, L., Peters, F., and van Lierop, M., editors, *Data Structures for Raster Graphics , proceedings of a Workshop held at Steensel, The Netherlands, June 24–28, 1985*. Springer-Verlag EurographicSeminars.

Franklin, W. R. (1987). Polygon properties calculated from the vertex neighborhoods. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 110–118.

Franklin, W. R. (1990). Calculating map overlay polygon' areas without explicitly calculating the polygons — implementation. In *4th International Symposium on Spatial Data Handling*, pages 151–160, Zürich.

Franklin, W. R., Chandrasekhar, N., Kankanhalli, M., Akman, V., and Wu, P. Y. (1990). Efficient geometric operations for CAD. In Wozny, M. J., Turner, J. U., and Preiss, K., editors, *Geometric Modeling for Product Engineering*, pages 485–498. Elsevier Science Publishers B.V. (North-Holland). Selected and expanded papers from the IFIP WG 5.2/NSF Working Conference on Geometric Modeling, Rensselaerville, USA, 18-22 September 1988.

Franklin, W. R. and Kankanhalli, M. S. (1993). Volumes from overlaying 3-D triangulations in parallel. In Abel, D. and Ooi, B., editors, *Advances in Spatial Databases: Third Intl. Symp., SSD'93*, volume 692 of *Lecture Notes in Computer Science*, pages 477–489. Springer-Verlag.

Franklin, W. R., Narayanaswami, C., Kankanhalli, M., Sun, D., Zhou, M.-C., and Wu, P. Y. (1989). Uniform grids: A technique for intersection detection on serial and parallel machines. In *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography*, pages 100–109, Baltimore, Maryland.

Franklin, W. R. and Sivaswami, V. (1990). OVERPROP — calculating areas of map overlay polygons without

calculating the overlay. In *Second National Conference on Geographic Information Systems*, pages 1646–1654, Ottawa.

Guevara, A. J. (1983). *A Framework for the Analysis of Geographic Information System Procedures: The Polygon Overlay Problem, Computational Complexity and Polyline Intersection*. PhD thesis, State University of New York at Buffalo.

Hopkins, S. and Healey, R. G. (1990). A parallel implementation of Franklin's uniform grid technique for line intersection detection on a large transputer array. In Brassel, K. and Kishimoto, H., editors, *4th International Symposium on Spatial Data Handling*, pages 95–104, Zürich.

Knuth, D. E. (1974). *Surreal Numbers: How Two Ex-students Turned on to Pure Mathematics and Found Total Happiness: A Mathematical Novelette*. Addison-Wesley.

Lam, N. (1977). Polygon overlay: An examination of an algorithm and related problem. Master's thesis, University of Western Ontario.

Lee, D. T. and Preparata, F. P. (1977). Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606.

Marble, D. F. (1990). The potential methodological impact of geographic information systems on the social sciences. In Allen, K. M., Green, S. W., and Zubrow, E. B., editors, *Interpreting Space: GIS and Archaeology*. Taylor & Francis.

Preparata, F. P. and Shamos, M. I. (1985). *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science Springer-Verlag.

Preparata, F. P. and Tamassia, R. (1989). Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18:811–830.

Sarnak, N. and Tarjan, R. E. (1986). Planar point location using persistent search trees. *Commun. ACM*, 29:669–679.

Sivaswami, V. (1990). Point inclusion testing in polygons and point location in planar graphs using the uniform grid technique. Master's thesis, Rensselaer Polytechnic Institute, Electrical, Computer, and Systems Engineering Dept.

Stewart, A. J. (1991). Robust point location in approximate polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 179–182.

Sun, D. (1989). Implementation of a fast map overlay system in C. Master's thesis, Rensselaer Polytechnic Institute, Electrical, Computer, and Systems Engineering Dept.

Tamassia, R. (1991). An incremental reconstruction method for dynamic planar point location. *Inform. Process. Lett.*, 37:79–83.

Ullman, J. D. (1988). *Principles of Database and Knowledge-base Systems*. Principles of computer science series, 14. Computer Science Press.

Wagner, D. F. (1987). A comparison and evaluation of three polygon overlay algorithms. unpublished M.A. paper, The Ohio State University.

Wagner, D. F. (1991). *Development and Proof-of-Concept of a Comprehensive Performance Evaluation Methodology for Geographic Information Systems*. PhD thesis, The Ohio State University.

White, D. (1977). A new method of polygon overlay. In *An Advanced Study Symposium on Topological Data Structures for Geographic Information Systems*, Cambridge, MA, USA, 02138. Laboratory for Computer Graphics and Spatial Analysis, Harvard University.

White, D., Maizel, M., Chan, K., and Corson-Rikert, J. (1989). Polygon overlay to support point sample mapping: The national resources inventory. In *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography*, pages 384–390, Baltimore, Maryland.

Wu, P. Y. and Franklin, W. R. (1990). A logic programming approach to cartographic map overlay. *Canadian Computational Intelligence Journal*, 6(2):61–70.

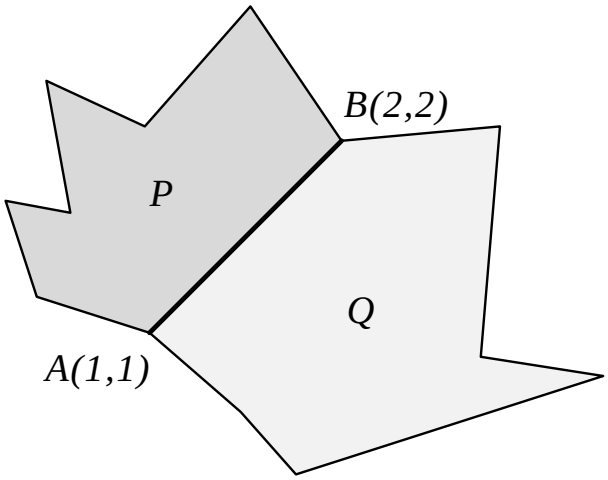

Figure 1: Splitting an Edge into Half-Edges

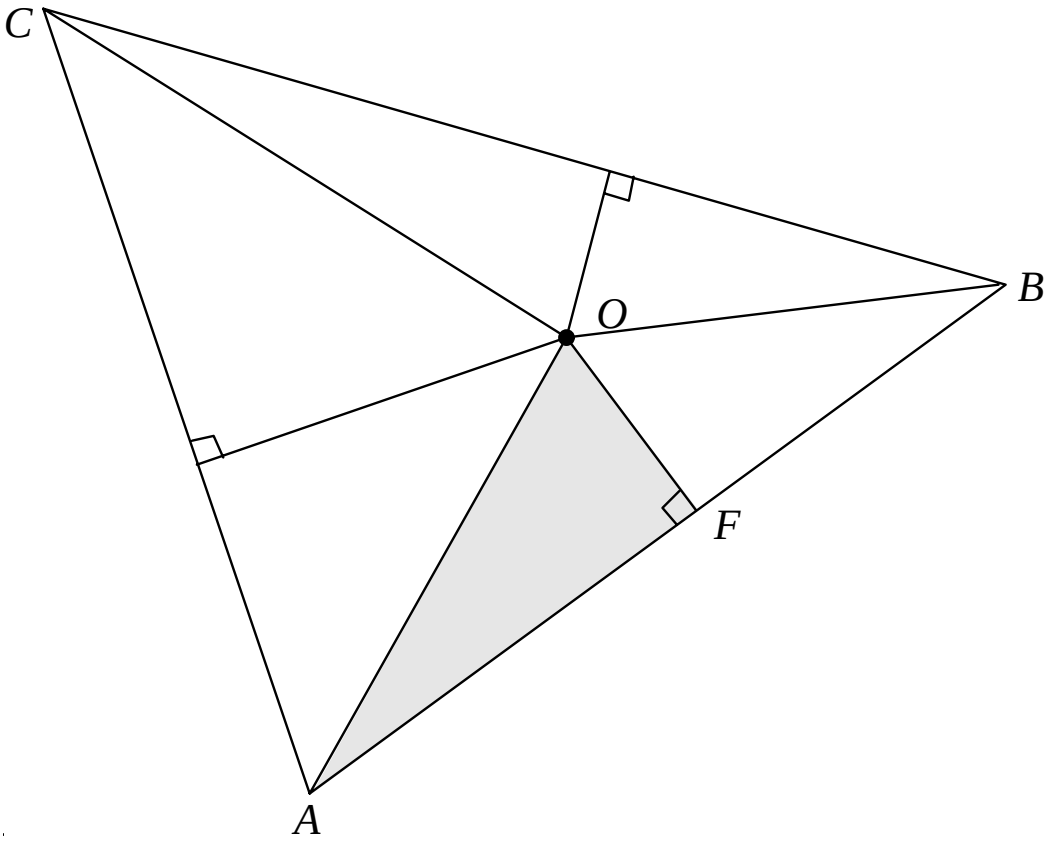

Figure 2: Idea Behind the Local Topological Formulæ

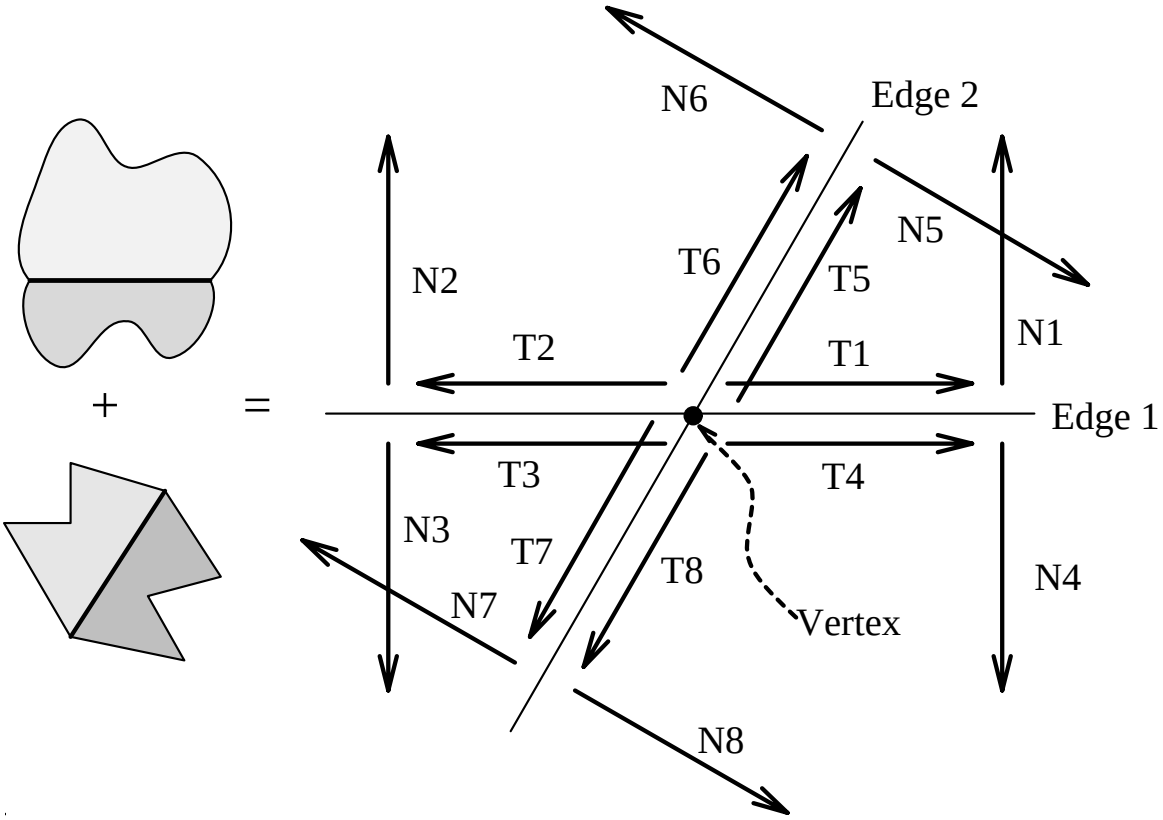

Figure 3: The 8 Half-Edges Derived from One Intersection

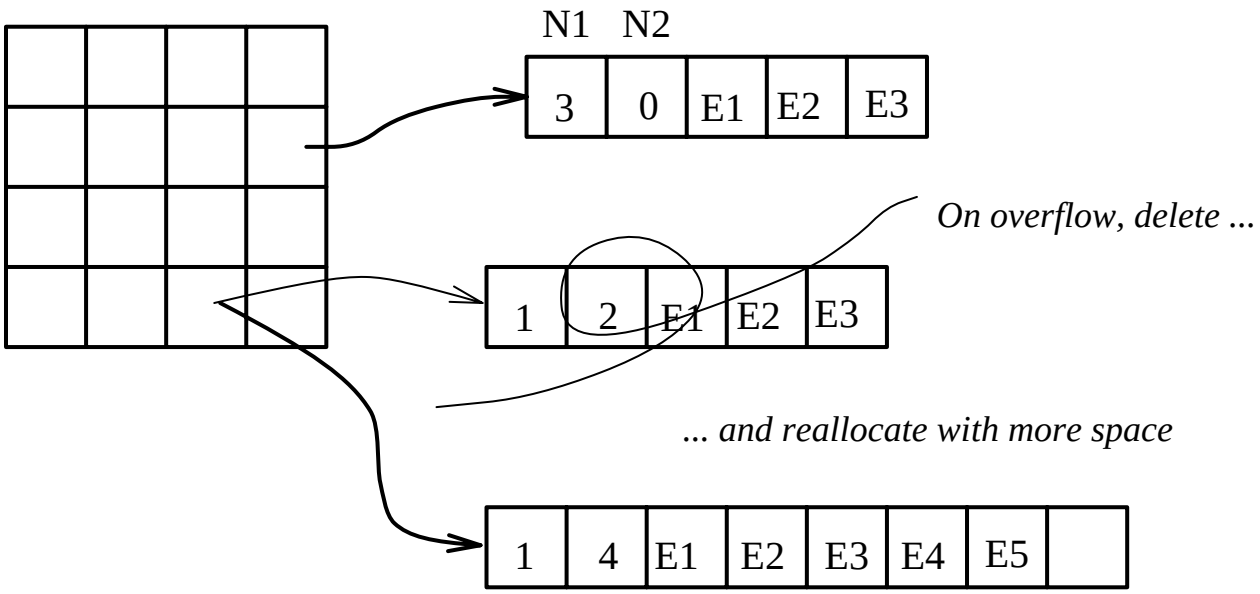

Figure 4: Expandable Array for Storing Edges in Cells

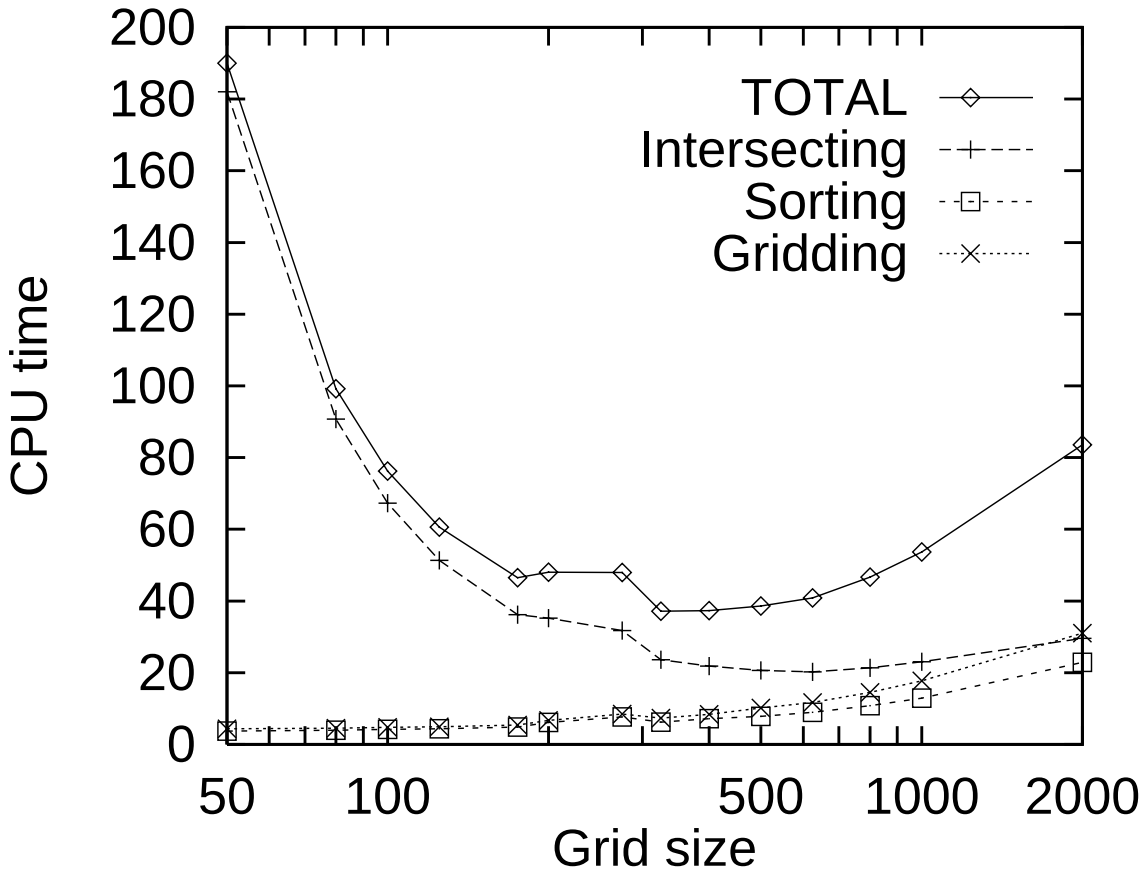

Figure 5: Intersection Time vs Grid Size

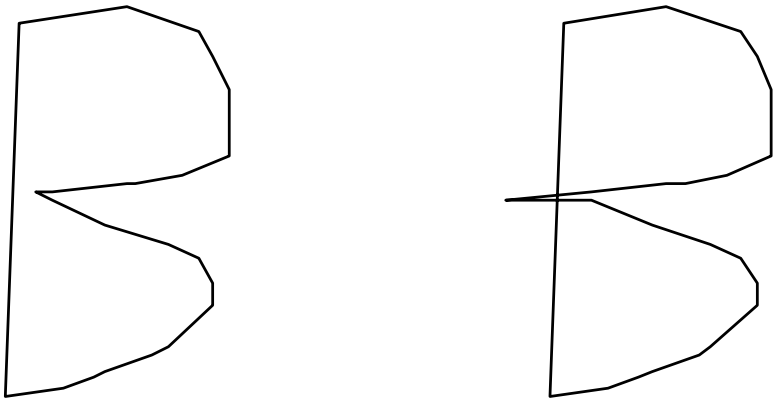

Figure 6: Topological Error Caused by Numerical Inaccuracy

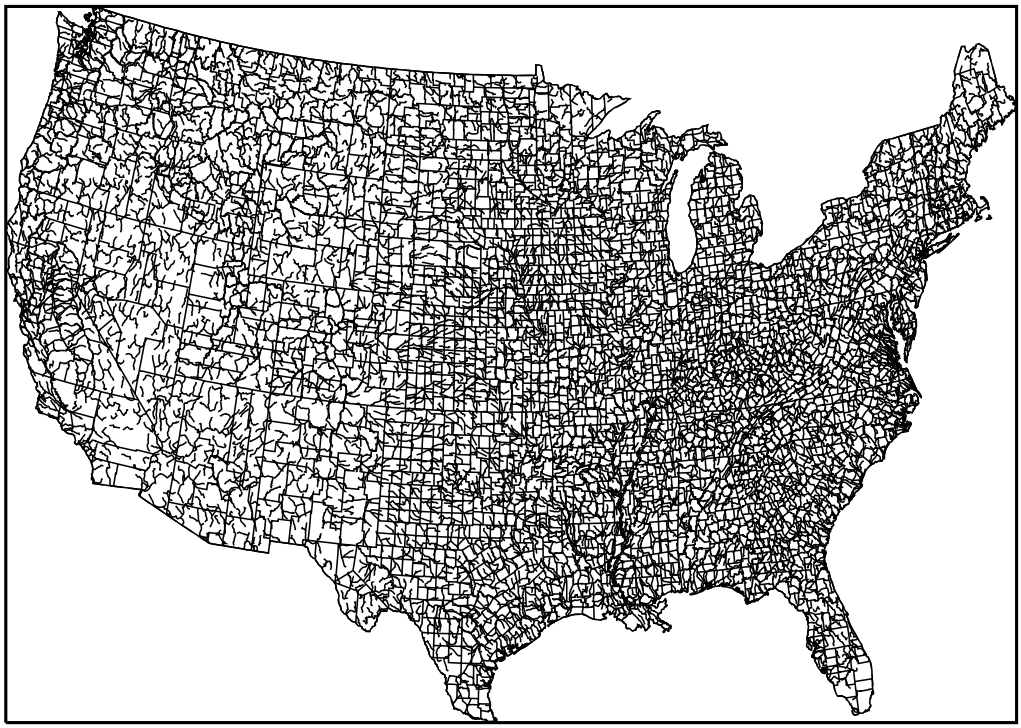

Figure 7: US Counties Versus Hydrography Polygons

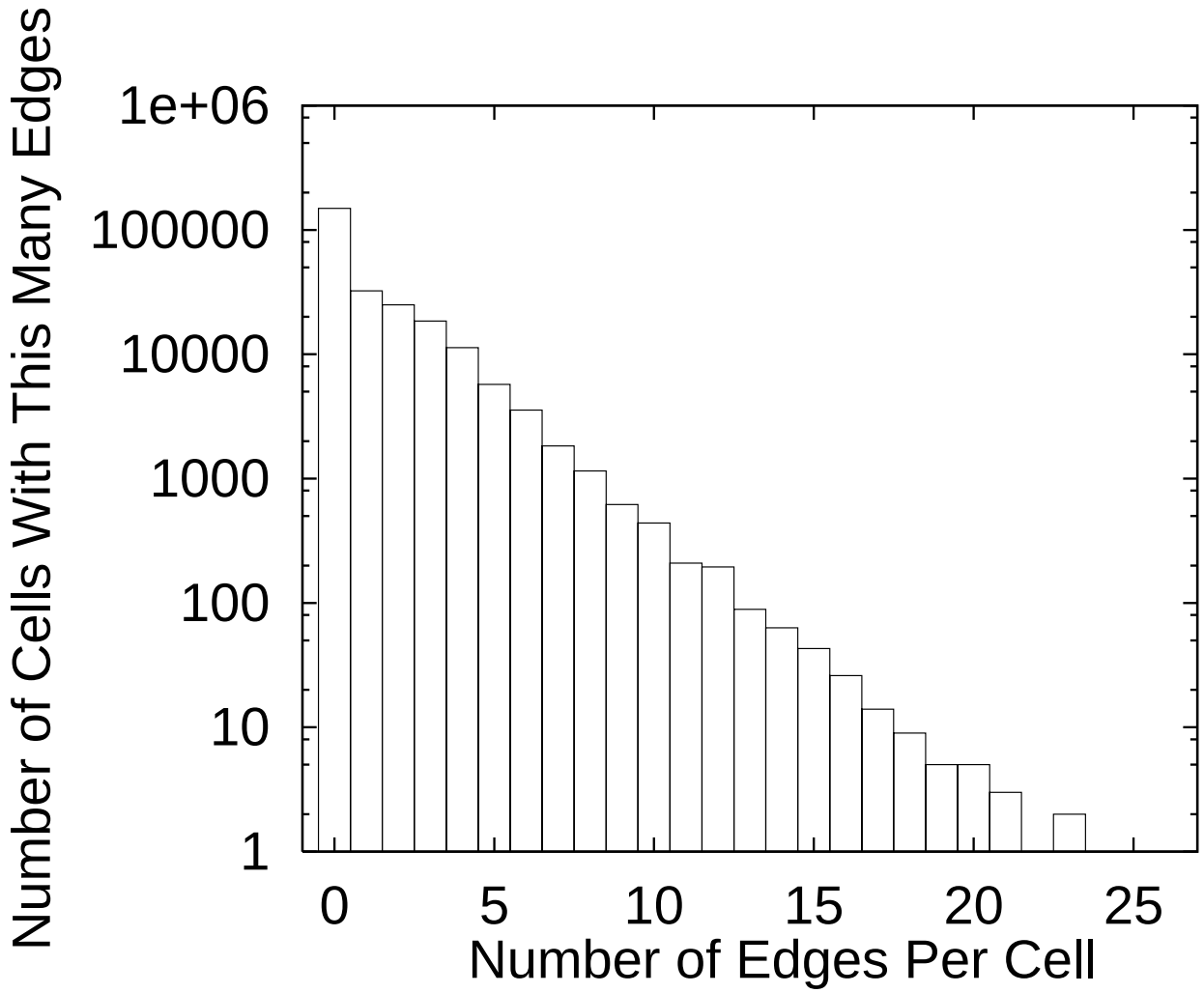

Figure 8: Distribution of Edges per Cell