Further Comparison of Algorithms for Geometric Intersection Problems

D. S. Andrews J. Snoeyink J. Boritz T. Chan
G. Denham J. Harrison C. Zhu
Department of Computer Science
University of British Columbia
201 – 2366 Main Mall
Vancouver, BC V6T 1Z4 Canada
Email contact: andrews@cs.ubc.ca

Abstract

The usual first step in computing an overlay of two vector maps is to determine which pairs of segments (one from each map) intersect so as to perform line-breaking. We identify two classes of algorithms for the segment intersection problem, spatial partitioning and spatial ordering, and we report on implementations of seven algorithms, some known and some new. Though the spatial ordering algorithms can be made *output sensitive*, our experiments show that the spatial partitioning algorithms are better for segment intersection in a GIS context. We do identify a new Trapezoid Sweep algorithm that is competitive if the segments are stored in sorted order.

1 Introduction

Map overlay processing is at the core of most vector-based Geographic Information Systems (GISs). One of the time-consuming steps of this processing is *line-breaking*, which we can abstract as the *segment intersection problem:* Given a collection of n line segments in the plane, determine which pairs intersect.

For overlaying two maps, we can refine this to the *red/blue segment intersection problem:* Given a set R of red line segments in which no pair intersect except possibly at endpoints, and a set B of blue line segments (again, having disjoint interiors), find the red/blue intersections—all pairs of a red and a blue segment that intersect. This distinction between the general segment intersection problem and the red/blue segment intersection problem has been made in computational geometry, which is a branch of the theoretical analysis of algorithms that studies geometric computation. It has not been made in the more practically-oriented literature on map overlay in GIS. We undertook this research to determine if this distinction could make a practical difference.

We survey results on segment intersection that have been reported in the practical and theoretical literature in the next section. After discussing the basis for algorithm comparison in section 3, we divide the algorithms that we study into two groups and describe them in:

Section 4 Spatial partition algorithms, which partition the plane and its line segments into regions in which the problem is solved by exhaustive checking, and

Section 5 Spatial ordering algorithms, which order the segments—often by a sweep—and use the properties of the red/blue problem to determine intersections.

Section 6 describes the test data sets, lists the results of runs, and interprets the results.

Spatial ordering algorithms can be made *output sensitive*—that is, they can be implemented to perform work proportional to the number of segments plus the number of reported intersections, for any input data set. Such a property is appealing because in GIS data intersections are relatively rare. Partitioning (also called bucketing or tiling) algorithms, however, work especially well in GIS because segments tend to be short, sparse, and evenly distributed in the plane.

2 Algorithm comparisons in the literature

Three authors have argued eloquently for spatial partitioning approaches. Franklin [6, 7, 8] advocates the Uniform Grid. At the fourth spatial data handling conference, Pullar showed that a tiling approach could be far superior to a theoretically efficient sweep [13]. Samet [14, 15] advocates quadtree partitioning. Because our initial emphasis was on novel algorithms specifically for the red/blue intersection problem, we have not explored all known variants of these algorithms. We can, however, partially rehabilitate theoretically efficient algorithms from the unflattering comparisons that Pullar gave. The Trapezoid Sweep algorithm described in section 5.2 is a new algorithm by Timothy Chan [2] that is holds its own against our partitioning algorithms.

The computational geometry literature has studied output-sensitive algorithms for the general and red/blue segment intersection problems. Theoretical (mathematical) analysis of algorithms often studies the worst-case performance of algorithms. (One would like to study the average case, but mathematically defining an "average" instance of, say, map overlay is impossible. We believe that a combination of mathematical analysis and execution on sample data sets is the best way to understand the performance of an algorithm.) For a segment intersection problem instance involving n segments, r red and b blue, there can be $r \cdot b$ pairs of intersecting segments; therefore, in the worst-case, no algorithm can significantly outperform the trivial algorithm that tests each red segment against each blue. If the number of intersections is represented by a parameter K, however, then one can represent the running time of an algorithm as an overhead term plus an output-sensitive term that depends on K. For example, the red/blue intersection algorithms of sections 5.2 and 5.3 are optimal—they run in $\Theta(n \log n + K)$ time and use $O(n)$ space.[1]

Bently and Ottmann [1] developed the plane-sweep algorithm described in section 5.1 to solve the general segment intersection problem in $O((n + K)\log n)$ time and linear space. Mairson and Stolfi [10] extended a sweep to incorporate properties of the red/blue intersection problems and obtained $O(n \log n + K)$ time. Chazelle and Edelsbrunner [3] gave an intricate output-sensitive algorithm for the general intersection problem that achieves this time, but may require $\Theta(n + K)$ space. Others [4, 11] have determined how to count the intersections without computing all of them—the algorithm of section 5.3 does this.

[1]The big-O notation is often used to hide implementation-dependent constants in the running time. A function $f(n) = O(g(n))$ if there are constants N and c such that $f(n) < cg(n)$ for all $n > N$. Big-Θ notation means that there is both an upper and a lower bound of the given form.

3 Comparing algorithms

The task of comparing algorithms is a difficult one. Running two algorithms on particular instances of a problem gives little information unless one can extrapolate to predict behavior on other instances. A further complication is the fact that data sets for red/blue segment intersection problems vary widely; they can contain long or short segments that are clustered or spread over the plane and have many or few intersections.

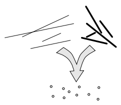
In the next sections we give the worst-case asymptotic times of algorithms running on r red and b blue segments, for a total of $n = r + b$ line segments. Times for spatial ordering algorithms in section 5 also depend on the number of intersection points, K. As is common, these running times are expressed asymptotically using big-O notation. The values of hidden constants are crucial for the practical application of an algorithm; thus, the analysis of running time is most useful as a way to extrapolate and explain the performance results listed in section 6.

We also comment on other factors:

- Memory requirements and locality of reference. The amount of storage needed per segment varies slightly from algorithm to algorithm. For example, some partitioning algorithms increase the number of segments by storing segments with each region that they intersect. Our algorithms have been implemented and tested on workstations with large virtual memory. Even here the amount and access patterns for memory are important because an algorithm's interaction with caching and paging schemes depends on its locality of reference.
- Dependence on the data sets. Some algorithms (e.g., the standard uniform grid) do not adapt to the data set, but assume certain statistical behavior (perhaps uniformly-distributed short segments) in order to deliver good performance. Others (e.g. quadtree and BSP tree) adapt their data structures to handle long segments or clusters. Still others use the individual segments of the data to determine the tests to perform.
- Ease of correct implementation. Some algorithms are easier than others to implement so that they correctly handle special cases and are not prone to errors due to the inaccuracies of floating point computations. The specification of correct behavior in special cases may not be even be clear. For example, if a red segment is a subset of a blue segment, should they be reported as intersecting zero, one, or infinitely many times, or be flagged for special treatment? (Algorithms that duplicate segments are apt to find these segments intersecting multiple times.) Since our algorithms were implemented by different programmers, we found that this case in particular was treated in different ways.
- Preprocessing. Several of the algorithms begin by sorting the endpoints of each color by x-coordinate or putting red or blue segments into appropriate data structures. These steps can be treated as preprocessing—they can be performed once and the results can be stored with the data so that any future overlays involving the same map layer can be performed more efficiently.

We can use the brute force algorithm as an example of how we list an algorithm's characteristics and describe its operation.

3.1 Brute Force



Characteristics: $\Theta(n^2)$ best, worst, and average-case time. $O(n)$ space, can have good locality. Small constant factors. Ignores data characteristics. No preprocessing.

The *brute force* approach to the line segment intersection problem simply tests all pairs of segments for intersection. For the red/blue problem, one tests only pairs consisting of a red and a blue segment—we implemented this algorithm for comparison purposes.

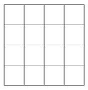
When all intersections actually occur, or when the number of segments of one of the colors is small, the brute force solution can be effective because of its low overhead. For most GIS data sets, however, it is better to spend some effort reducing the number of segments to be tested against a given segment.

4 Spatial Partitioning Algorithms

Spatial partitioning algorithms divide the plane into disjoint, non-overlapping regions, distribute the segments among the regions that they intersect, and then determine the intersections that lie in each region—often by a brute force algorithm. Partitioning algorithms based on a uniform grid, on a quadtree, and on a binary space partitions have been implemented and tested.

There are some caveats for implementing partitioning algorithms. To correctly count intersections on the boundaries of partition regions, these boundaries must be defined so that the regions cover the plane but do not overlap. It is tempting to clip line segments to the regions that contain them, but for computational accuracy one should always compute locations of intersection points using original data points. One would therefore like to keep the region descriptions simple so that one can determine whether an intersection point lies in the region or not.

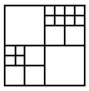
4.1 Uniform Grid/Tiling



Characteristics: With a $g \times g$ grid, $O(n^2)$ worst and about $O(n \log g + (n/g^2)^2)$ best-case time. $O(ng)$ worst and $O(n)$ best-case space. Small constant factors. Will be closer to best cases except for long and unevenly distributed segments. Typically no preprocessing, although one can assign segments to grid cells. (We did so in our implementation.)

The simplest way to do spatial partitioning is to distribute segments to cells in a uniform grid as advocated by Franklin [6, 8, 7]. When the segments are short and sparse in the plane, this approach can lead to significant savings.

4.2 Quadtree



Characteristics: $O(n^2)$ worst-case time, and grid-like performance in the best case. $O(n)$ space. Fairly small constant factors. Long segments are stored once (in our implementation), but still increase computational complexity. Memory and, to a lesser extent, time adapt to data distribution. Typically no preprocessing, but segments of each color could be stored in a quadtree.

A quadtree is a hierarchical decomposition of the plane into rectangles. In our application, we took the bounding box of the red line segments and cut it into four equal-sized rectangles using two lines. These were made children of the original rectangle. Red segments completely contained in one of the four rectangles were stored in quadtrees defined recursively for the children. Those segments that overlapped two or more rectangles were stored in the original. If there are few segments or the depth of a rectangle is large, then the rectangle is not subdivided further. There are other ways to store segments in a quadtree [14, 15, 16]; we chose this one because it saves memory by storing each segment only once.

To compute intersections of a blue segment b with the red segments stored in a quadtree, one simply starts from the root of the tree, determines all rectangles that intersect b, and intersects b with all red segments stored in these rectangles.

Long segments that cross rectangle boundaries do not increase the memory requirements under this type of quadtree storage, whereas they do in the uniform grid. Long segments do, however, increase the computation time. If they are red, then they are stored near the root of the tree and many blue segments are compared against them; if they are blue, then they cross many rectangles and are compared against many segments. The short segments of in most GIS applications, however, can be handled well and the partitioning done by the data structure adapts to clustering of the data.

4.3 Binary Space Partition

Characteristics: Time and space depend on exact partitioning strategy. $O(n^2)$ worst case time and space, but expect better. Reasonable constant factors. Long segments increase storage. Adapts to data clustering. No preprocessing.

Binary space partition (BSP) trees were developed in computer graphics [9, 17] and have been applied in GIS by van Oosterom [18]. A BSP tree represents a partition of a region of the plane by storing a line in a root node and having two children that recursively store partitions of the regions to the right and left of the line. A leaf of the BSP tree, therefore, represent convex regions that are the intersections of halfplanes bounded by lines stored in the ancestors of the leaf.

Red and blue line segments are stored at the leaves of a BSP tree. Segments are initially compared against the line that is stored at the root node. Segments that lie strictly to the right or left of this line are stored in the right or left subtree; segments that intersect the line are stored in both.

Our goal is to partition the lines until one color has few segments and we can efficiently run the brute force algorithm. To achieve this, we can choose the partitioning lines in the BSP based on the data sets: one method that we used successfully was to compute the centroids of the red and the blue segments at a tree node and then take the bisector of the centroids as our partitioning line. This partitioning strategy works especially well when red segments and blue segments form disjoint clusters; it tends to separate these clusters and avoid testing many possible pairs.

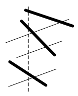
5 Spatial Ordering Algorithms

Because partitioning algorithms partition with more or less disregard for the data, one can devise data sets for which partitioning algorithms take quadratic time even when there are no intersections. Long segments and/or uneven clustering are frequently the source of the problem.

As an alternative, one can consider spatial ordering algorithms that use an *aboveness* ordering to reduce the number of segment comparisons that need to be made: segment A is *above* B if some vertical line intersects A at a greater y coordinate than B. For disjoint line segments in the plane this is a partial order.

Our spatial ordering algorithms use aboveness in different ways. The Bentley-Ottmann algorithm (section 5.1) uses direct comparisons by aboveness and maintains an ordered list of segments that intersects a vertical sweepline. The Trapezoid Sweep algorithm (section 5.2) disguises its use of aboveness; it forms a trapezoidation that guides the search for intersections. The Hereditary Segment Tree algorithm (section 5.3) extends the aboveness relation to a total order and handles segments in that order.

5.1 The Bentley-Ottmann Sweep



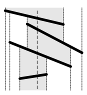
Characteristics: $O(n \log n + K \log n)$ worst, and average-case time. $O(n)$ space. Moderately large constant factors. No data dependence. Adapts somewhat if few segments intersect the same vertical line. Can sort endpoints by x-coordinate in preprocessing.

A conceptually simple example of the use of an aboveness order is Bentley and Ottmann's plane-sweep algorithm for line segment intersections [1]. This algorithm moves a sweepline ℓ from left to right across the plane, maintaining a list of the segments that cross ℓ in aboveness order. When the leftmost or rightmost endpoint of a new segment is encountered, that segment is inserted into or deleted from the list, respectively. Intersections are found when two adjacent segments swap order. The next intersection the sweepline is to encounter always occurs between adjacent segments in the list.

By using balanced search trees, one can keep track of the list as it changes. Thus, the plane-sweep changes a static two-dimensional problem to a dynamic one-dimensional problem. The endpoints of the segments need to be sorted so that they can be processed in x-coordinate order by the sweep. Unfortunately, the intersections are also reported in x-sorted order. Thus, this algorithm takes $O((n+K) \log n)$ time to report all K intersections. The overhead associated with maintaining balanced binary search trees means that the hidden constant is not negligible in practice.

Mairson and Stolfi [10] were the first to use the aboveness properties of the red/blue intersection problem to avoid sorting all the intersections by x-coordinate. They use a recursive "cone-breaking" technique to obtain an algorithm that runs in $O(n \log n + K)$ time. Because our Trapezoid Sweep has the same running time with smaller constants, we describe it instead.

5.2 The Trapezoid Sweep



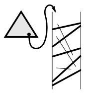
Characteristics: $O(n \log n+K)$ worst and average-case time. $O(n)$ space. Fairly small constant factors. No data dependence. Adapts somewhat if few segments intersect the same vertical line. Can sort endpoints or even cut plane into trapezoids in preprocessing.

Because the Bentley-Ottmann algorithm sorts all intersections by x-coordinates, is pays a logarithmic overhead for each intersection that it finds. The trapezoid sweep finds intersections that occur left of a zig-zag *sweep front* and, therefore, does not sort all intersections.

Imagine a *blue trapezoidation*—a decomposition of the plane into trapezoids that are bounded by the blue line segments—by cutting vertically from the endpoints of both red and blue segments to the blue segments above and below. We then sweep the blue trapezoids with a vertical line and insert the red segments as follows. We maintain a list $\mathcal{T}$ of trapezoids that the sweepline intersects. The *sweep front* consists of the left boundary of the union of trapezoids in $\mathcal{T}$. We separately maintain the list of red segments that the sweep intersects.

For each red segment r, we maintain the invariant that the intersection points on r that are left of points where r intersects the sweep front have been reported. If we begin the sweep with a vertical line to the left of all of the segments, then this invariant is trivially true. When we encounter the end of a blue trapezoid we advance the sweep front, then report the red intersections by a walk in the red tree until the invariant is re-established. Details are omitted due to space constraints, see [2].

5.3 Hereditary Segment Tree



Characteristics: $O(n \log n)$ worst and average-case time because intersections can be reported in batches. $O(n)$ space. Somewhat large constant factors. Adapts somewhat if few segments intersect the same vertical line. Can sort segments by aboveness in preprocessing.

Chazelle et al. [4] suggested and Palazzi and Snoeyink [11] simplified a red/blue intersection algorithm based on the *hereditary segment tree* data structure that uses the spatial ordering directly.

This algorithm begins by sorting the red segments and the blue segments by aboveness. Since the colors are independent, the files of segments need be sorted only once as preprocessing. Next, the *hereditary segment tree*, a modification of the *segment tree* [12], is used to find $2n$ *slabs* and special segments that intersect them. (The tree can act as a a guide and need not actually be constructed [11].) Each slab has a group of red (or blue) *long* segments that cut completely through the slab and a group of blue (or red) *short* segments that are clipped to the slab and may end inside the slab. These slabs and segments have the property that every red/blue intersection can be found in exactly one slab as the intersection of a long segment and a short segment that are grouped with the slab.

To find the intersections in the slabs is now an easy task. If the long segments are listed by aboveness, then one can locate the endpoints of each short segment s in the list and report intersections of s with every long segment that lies between the endpoints of s. $O(n \log n + K)$ time is required to find the K red/blue intersections of n segments; $O(n)$ space is used for data structures.

This algorithm has one novel feature. It can count intersections in $O(n \log n)$ time—that is, without looking at all of them. The number of intersections for a short segment s in a slab is simply the difference in the numbers of long segments above the two endpoints of s.

6 Data Sets

To test the algorithms when many intersections occurred we generated long near-horizontal and near-vertical algorithms that form a grid pattern. To give more realistic tests of overlay, we used 1:24,000 Digital Line Graph (DLG) data on Littleton, CO, obtainable by anonymous ftp from xerox.spectrum.com and data on the Malcolm Knapp Research Forest from Jerry Maedel in the UBC Department of Forestry. See table 1.

The algorithms are denoted by abbreviations. bforce: brute force (3.1), quad: quad tree partitioning (4.2), bsp: binary space partition tree (4.3), BO: Bentley-Ottmann sweep (5.1), trap: trapezoid sweep (5.2), and segT: hereditary segment tree (5.3). A plus sign (+) after trap or segT

Data Set	# Segs	Description
Hn	n	Long, near-horizontal segments to create n^2 intersections
Vn	n	Long, near-vertical segments to create n^2 intersections
surv	239	Survey grid for Littleton, CO 1:24K DLG from spectrum.xerox.com
rail	447	Railroads for Littleton. N to S in W half of map.
feat	564	Geographical features. A small number of small clusters.
bound	1176	Boundaries. Follows roads in town and meanders outside.
vegit	5562	Vegitation. Small clusters away from town.
roads	11074	Roads and trails. Mostly in town (NW corner) and suburbs (N of center).
hydro	13972	Hydrography. Evenly distributed.
comp	8053	Research compartments in Malcolm Knapp Forest, B.C.. Clustered in center.
froads	19532	Forest roads. Somewhat evenly distributed.
fcover	116359	Forest cover. Evenly distributed.
biogeo	235635	Biological and geographic features. Evenly distributed.

Table 1: Characteristics of test data sets

indicates that preprocessing is included in the time. If the plus is omitted, then the algorithm assumes that the segments are stored in a sorted order that the algorithm can use.

Running times on the artificial and Littleton data were measured on a Silicon Graphics Personal Iris with a 12 Mhz processor and 16M of memory. The Research Forest data sets were run on a SGI Indigo Elan with a 33 Mhz processor and 32M of memory. The average times for 10 runs are given in seconds.

We should emphasize that the algorithms were implemented by different programmers and that factors such as programming skill, attention paid to correctness and accuracy, and engineering decisions made by the programmer have a significant affect on running times and memory usage.

Artificial Data	$\bigcap$s	bforce	grid	quad	bsp	BO	trap+	segT+	segT
H10/V10	100	0.008	0.008	0.005	0.0	0.041	0.009	0.008	0.002
H50/V50	2,500	0.130	0.132	0.110	0.3	1.517	0.103	0.040	0.012
H100/V100	10K	0.510	0.512	0.410	1.3	7.015	0.357	0.093	0.023
H500/V500	250K	12.833	12.560	10.438	36.7		8.454	0.587	0.128
H1000/V1000	1M	51.632	50.137	49.556	145.4		34.167	1.283	0.278
H5000/V5000	25M							7.927	1.500
H10000/V10000	100M							17.252	3.098
H10/H10	0	0.005	0.005	0.003	0.000	0.016	0.006	0.017	0.012
H50/H50	0	0.103	0.063	0.077	0.100	0.071	0.032	0.103	0.062
H100/H100	0	0.408	0.232	0.288	0.300	0.153	0.070	0.248	0.147
H500/H500	0	10.205	8.305	7.363	9.800	0.939	0.440	1.815	1.070
H1000/H1000	0	40.950	21.963	36.710		2.111	0.955	4.193	2.557

Table 2: Performance for long segments with many or no intersections

We begin with artificial data sets that show the output-sensitive algorithms in the best light. These consist of long segments that either have a quadratic number of intersections (K and M denote thousands and millions) or no intersections. As table 2 shows, the spatial partitioning algorithms run in quadratic time whether or not there are many intersections. The spatial ordering algorithms, being output sensitive, are much faster when there are no intersections. The Bentley-Ottmann sweep (BO) is especially expensive when there are many intersections because it performs balanced-tree operations for every intersection it finds.

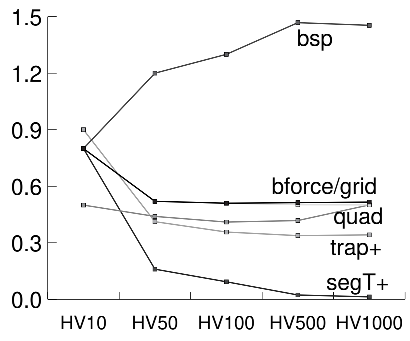


Figure 1: Cost (μsec) per *intersection:* artificial data

Figure 1 displays the cost in milliseconds per *intersection* for the grid of horizontal and vertical segments. The hereditary segment tree runs in subquadratic time—even when there are a quadratic number of intersections—because reports intersections in batches. It can report, for example, that a red segment r intersects blue segments in a list from index i to index j. It also appears to be taking advantage of the clustering of the segment endpoints by x-coordinate, which is why it is faster for many intersections than for no intersections.

For a more realistic comparisons, we run the same algorithms on the surv (survey grid) and one other GIS data set in table 3. Figure 2 graphs the time taken per line segment in milliseconds.

Unlike the other algorithms, the performance of the quad tree algorithm depends on the order in which the files are presented. Column quad1 shows the results for building the quadtree on the surv segments and quad2 shows the other order. It is better to do the latter because a large data set induces a finer partition and avoids more comparisons. The exploitation of clustering can be seen in the times for rail vs. feat and bound vs. vegit.

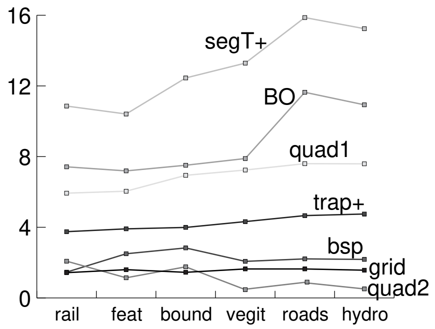


Figure 2: Cost per *segment:* $\bigcap$ with surv

Here the spatial partitioning algorithms are the best. They take advantage of the characteristics of GIS data (short segments, evenly distributed) and the fact that the number of intersections per segment is small. They also benefit most from the fact that the surv data set is small; the spatial ordering algorithms have an overhead associated with looking at the structure of the larger data set. The trapezoid sweep is somewhat competitive; it would be more so if we assumed that the initial sorting had been done in preprocessing.

If we concentrate on the algorithms that have performed well above, we can attempt larger data sets. In table 4 we report the times for trap and segT including and excluding preprocessing.

Survey Data	# Segs	$\bigcap$s	bforce	grid	quad1	quad2	bsp	BO	trap+	segT+
surv/rail	239/ 447	82	4.365	0.098	0.405	0.142	0.10	0.507	0.256	0.742
surv/feat	239/ 564	40	5.547	0.128	0.483	0.091	0.20	0.575	0.313	0.833
surv/bound	239/ 1176	128	11.515	0.205	0.980	0.248	0.40	1.060	0.564	1.758
surv/vegit	239/ 5562	108	58.068	0.948	4.200	0.271	1.20	4.573	2.502	7.703
surv/roads	239/11074	499	—	1.858	8.590	0.968	2.50	13.170	5.265	17.947
surv/hydro	239/13972	367	—	2.237	10.778	0.723	3.10	15.531	6.747	21.660

Table 3: Intersections with the survey grid

Large Sets	# Segs	$\bigcap$s	grid	quad1	quad2	bsp	trap+	trap	segT+	segT
feat/hydro	564/13972	100	4.507	3.663	1.342	2.20	6.607	2.970	22.905	10.320
bound/vegit	1176/ 5562	90	3.618	3.873	1.003	1.30	2.903	1.422	10.657	5.480
bound/roads	1176/11074	225	5.160	7.166	3.354	2.60	5.620	2.636	21.883	10.655
bound/hydro	1176/13972	249	9.045	9.622	2.667	3.30	7.150	3.313	26.135	12.867
vegit/roads	5562/11074	85	20.022	6.839	13.648	3.40	7.919	3.607	32.320	17.543
vegit/hydro	5562/13972	251	39.208	10.281	11.570	4.50	9.517	4.287	37.935	21.045
roads/hydro	11074/13972	569	63.645	38.021	21.904	7.30	13.430	6.200	55.167	31.845
comp/froads	8053/19532	12670	52.02	6.04	9.96	—		2.49		12.40
comp/fcover	8053/ 116K	5119	295.98	53.45	23.68	—		11.94		57.11
comp/biogeo	8053/ 235K	4482	607.58	131.32	44.41	20.0		27.45		110.67
roads/fcover	19532/ 116K	861	702.12	76.16	53.21	12.0		12.75		71.35
roads/biogeo	19532/ 235K	1500	1369.87	126.08	100.80	22.0		27.74		133.13
fcover/biogeo	116K/ 235K	35537	—	652.81	712.41	—		49.35		244.75

Table 4: Running times for larger data sets.

Again, the spatial partitioning algorithms perform the best, but the trapezoid sweep is competitive—especially if the segments are already stored in sorted order. The memory requirements of the quadtree algorithm and the bsp tree algorithm did not allow us to run many tests on the research forest data.

To produce instances with a single data distribution and number of segments, we intersected two copies of a map—one displaced slightly relative to the other. Table 5 shows that here the trapezoid algorithm without counting preprocessing is best, followed by the binary space partition, and the trapezoid algorithm with preprocessing. Figure 4 again shows running time in milliseconds per segment.

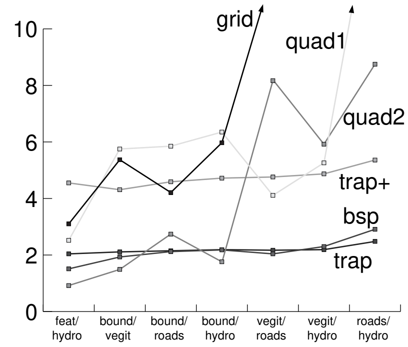


Figure 3: Large data sets (μsecs/segment)

Bumped	# Segs	$\bigcap$s	bforce	grid	quad	bsp	trap+	trap	segT+	segT
surv	239	140	2.397	0.077	0.283	0.20	0.181	0.107	0.560	0.300
rail	447	62	8.165	0.497	0.497	0.30	0.338	0.184	1.228	0.653
feat	564	56	13.115	1.968	0.521	0.30	0.452	0.247	1.527	0.748
bound	1176	220	57.137	1.445	1.245	0.70	0.956	0.502	3.725	2.037
vegit	5562	585	1362.072	43.788	7.370	3.50	5.118	2.467	21.152	12.350
roads	11074	2871	—	76.752	36.986	8.30	11.680	5.519	49.740	28.395
hydro	13972	1817	—	98.978	31.796	9.80	15.097	6.868	61.585	35.938

Table 5: Overlaying a map with a bumped version of itself.

7 Conclusions and Future Research

We grouped seven algorithms for segment intersection into two categories: spatial partitioning algorithms that divide space into regions and compute intersections among the segments that pass through each region, and spatial ordering algorithms that use the structure given by a geometric order to reduce the number of pairs of segments that must be tested for intersection. Despite our hope to the contrary, our experiments have shown that spatial partitioning algorithms give better performance on GIS data sets than spatial ordering algorithms, corroborating Pullar's work [13].

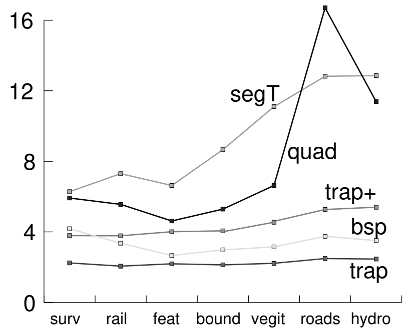


Figure 4: Bumped data (μsecs/segment)

We have identified a new spatial ordering algorithm, the trapezoid sweep of section 5.2, that is competitive with the partitioning algorithms—especially if the data is maintained in sorted order by x-coordinate. The fact that this algorithm uses geometric structure to find intersections may allow us to combine the line-breaking and topology-building phases of map overlay computation and avoid the more global break-and-rebuild operation that is necessary after finding intersections by brute force means. We plan to investigate this approach in future research.

Acknowledgements

This research was supported in part by Canada's National Science and Engineering Research Council and the B.C. Advanced Systems Institute. We thank Tom Poiker for encouragement to investigate the GIS overlay problem and the referees for their encouraging and helpful comments.

References

[1] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C–28(9):643–647, 1979.

[2] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Submitted to Proc. 6th Can. Conf. Comp. Geom.*, Saskatoon, Saskatchewan, 1994.

[3] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *JACM*, 39:1–54, 1992.

[4] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Technical Report UIUC DCS–R–90–1578, Dept. Comp. Sci., Univ. Ill. Urbana, 1990.

[5] N. R. Chrisman, J. A. Dougenik, and D. White. Lessons for the design of polygon overlay processing from the Odyssey WHIRLPOOL algorithm. In *Proc. 5th Intl. Symp. Spatial Data Handling*, pages 401–410. IGU Commission on GIS, 1992.

[6] W. R. Franklin. Efficient intersection calculations in large databases. In *International Cartographic Association 14th World Conference*, pages A62 – A63, Budapest, Aug. 1989.

[7] W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. *Comput. Aided Design*, 21(7):410–420, 1989.

[8] W. R. Franklin, C. Narayanaswami, M. Kankanhalli, D. Sun, M.-C. Zhou, and P. Y. F. Wu. 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, 2-7 April 1989.

[9] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Proc. SIGGRAPH '80*, pages 124–133, 1980.

[10] H. G. Mairson and J. Stolfi. Reporting line segment intersections. In R. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, number F40 in NATO ASI Series, pages 307–326. Springer-Verlag, 1988.

[11] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP: Graph. Mod. Image Proc.*, 1994.

[12] F. P. Preparata and M. I. Shamos. *Computational Geometry—An Introduction*. Springer-Verlag, New York, 1985.

[13] D. Pullar. Comparative study of algorithms for reporting geometrical intersections. In K. Brassel and H. Kishimoto, editors, *Proceedings of the 4th International Symposium on Spatial Data Handling*, pages 66–76, 1990.

[14] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1989.

[15] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[16] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *Int. J. GIS*, 4(2):103–131, 1990.

[17] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proc. SIGGRAPH'87*, pages 153–162, 1987.

[18] P. van Oosterom. A modified binary space partition for geographic information systems. *Int. J. GIS*, 4(2):133–146, 1990.