Algorithms for Performing Polygonal Map Overlay and Spatial Join on Massive Data Sets

Ludger Becker, André Giesen, Klaus H. Hinrichs, and Jan Vahrenhold

FB 15, Mathematik und Informatik, Westfälische Wilhelms-Universität Münster,
Einsteinstr. 62, D–48151 Münster, Germany
{beckelu,vanessa,khh,jan}@math.uni-muenster.de

Abstract. We consider the problem of performing polygonal map overlay and the refinement step of spatial overlay joins. We show how to adapt algorithms from computational geometry to solve these problems for massive data sets. A performance study with artificial and real-world data sets helps to identify the algorithm that should be used for given input data.

1 Introduction

During the last couple of years Spatial- and Geo-Information Systems (GIS) have been used in various application areas, like environmental monitoring and planning, rural and urban planning, and ecological research. Users of such systems frequently need to combine two sets of spatial objects based on a spatial relationship.

In two-dimensional vector based systems combining two maps m_1 and m_2 consisting of polygonal objects by *map overlay* or *spatial overlay join* is an important operation. The *spatial overlay join* of m_1 and m_2 produces a set of pairs of objects (o_1, o_2) where $o_1 \in m_1, o_2 \in m_2$, and o_1 and o_2 intersect. In contrast, the *map overlay* produces a set of polygonal objects consisting of the following objects:

- All objects of m_1 intersecting no object of m_2
- All objects of m_2 intersecting no object of m_1
- All polygonal objects produced by two intersecting objects of m_1 and m_2

The map overlay operation can be considered a *special outer join*. For simplicity of presentation, we assume that all objects of map m_1 are marked red and that all objects of map m_2 are marked blue. Usually spatial join operations are performed in two steps [25]:

- In the *filter step* an—usually conservative—approximation of each spatial object, e.g., the minimum bounding rectangle, is used to eliminate objects that cannot be part of the result.
- In the *refinement step* each pair of objects passing the filter step is examined according to the spatial join condition.

R.H. Güting, D. Papadias, F. Lochovsky (Eds.): SSD'99, LNCS 1651, pp. 270–285, 1999.

Although there exists a variety of algorithms for realizing the filter step of the join operator for massive data sets [3,7,19,22,28], not much research has been done in realizing the refinement step of the map overlay operation for large sets of polygonal objects, with the exception of Kriegel et al. [21]. Brinkhoff et al. [8] and Huang et al. [20] present methods to speed up the refinement step of the spatial overlay join by either introducing an additional filter step which is based on a progressive approximation or exploiting symbolic intersection detection. Both approaches allow the early identification of some pairs of intersecting objects without investigating the exact shapes. However, they do not solve the map overlay problem.

In this paper we discuss how well known main-memory algorithms from computational geometry can help to perform map overlay and spatial overlap join in a GIS. More specifically, we show how the algorithm by Nievergelt and Preparata [24] and a new modification of Chan's line segment intersection algorithm [12] can be extended to cope with massive real-world data sets. Furthermore, we present an experimental comparison of these two algorithms showing that the algorithm by Nievergelt and Preparata performs better than the modified algorithm by Chan, if the number of intersection points is sublinear, although Chan's algorithm outperforms other line segment intersection algorithms [2]. In this paper we do not consider algorithms using a network oriented representation, e.g., [11,14,16,23], since many GIS support only polygon oriented representations, e.g., ARC/INFO [15].

The remainder of this paper is structured as follows. In Section 2 we review the standard algorithm by Nievergelt and Preparata for overlaying two sets of polygonal objects and show how to modify Chan's algorithm for line intersection in order to perform map overlay. In Section 3 we discuss extensions required to use these algorithms for practical massive data sets. The results of our experimental comparison are presented in Section 4.

2 Overlaying Two Sets of Polygonal Objects

Since the K intersection points between line segments from different input maps (N line segments in total) are part of the output map, line segment intersection has been identified as one of the central tasks of the overlay process [2]. Methods for finding these intersections can be categorized into two classes: algorithms which rely on a partitioning of the underlying space, and algorithms exploiting a spatial order defined on the segments. While representatives of the first group, e.g., the method of *adaptive grids* [17], tend to perform very well in practical situations, they cannot guarantee efficient treatment of all possible configurations. The worst-case behavior of these algorithms may match the complexity of a brute-force approach, whereas the behavior for practical data sets often depends on the distribution of the input data in the underlying space rather than on the parameters N and K. Since we will examine the practical behavior of overlay algorithms under varying parameters N and K (Section 4), we restrict our study to algorithms that do not rely on a partitioning of the underlying space.

Several efficient algorithms for the line segment intersection problem have been proposed [4,6,12,13,23]; most of these are based on the *plane-sweep* paradigm [29], a framework facilitating the establishment of a spatial order.

The characterizing feature of the plane-sweep technique is an (imaginary) line that is swept over the entire data set. For sake of simplicity, this *sweep-line* is usually assumed to be perpendicular to the x-axis of the coordinate system and to move from left to right. Any object intersected by the sweep-line at $x = t$ is called *active at time* t, and only active objects are involved in geometric computations at that time. These active objects are usually stored in a dictionary called *y-table*. The status of the y-table is updated as soon as the sweep-line moves to a point where the topology of the active objects changes discontinuously: for example, an object must be inserted in the y-table as soon as the sweep-line hits its leftmost point, and it must be removed after the sweep-line has passed its rightmost point. For a finite set of objects there are only finitely many points where the topology of the active objects changes discontinuously; these points are called events and are stored in increasing order of their x-coordinates as an ordered sequence, e.g, in a priority queue. This data structure is also called *x-queue*. Depending on the problem to be solved there may exist additional event types.

In the following two sections we recall the plane-sweep algorithm by Bentley and Ottmann [6] and its extension by Nievergelt and Preparata [24] to compute the overlay of polygonal objects. These algorithms exploit an order established during the plane-sweep: for a set of line segments we can define a total order on all segments active at a given time by means of the aboveness relation. An active segment a lying above an active segment b is considered to be "greater" than b. This order allows for efficient storage of all active segments in an ordered dictionary (y-table), e.g., in a balanced binary tree which in turn provides for fast access to the segments organized by it.

2.1 The Line Segment Intersection Algorithm by Bentley and Ottmann

The key observation for finding the intersections is that two segments intersecting at $x = t_0$ must be direct neighbors at $x = t_0 - \varepsilon$ for some $\varepsilon > 0$. To find all intersections it is therefore sufficient to examine all pairs of segments that become direct neighbors during the sweep. The neighborhood relation changes only at event points and only for those segments directly above or below the segments inserted, deleted, or exchanged at that point. Thus, we have to check two pairs of segments at each *insert* event and one pair of segments at each *delete* event. If we detect an intersection between two segments we do not report it immediately but insert a new *intersection* event in the x-queue. If we encounter an intersection event we report the two intersecting segments and exchange them in the y-table to reflect the change in topology. The neighbors of these segments change, and two new neighboring pairs need to be examined. Since there is one insert and one delete event for each of the N input segments and one event for each of the K intersection points the number of events stored in the x-queue is bounded

by $\mathcal{O}(N + K)$. At most N segments can be present in the y-table at any time. If we realize both the x-queue and the y-table so that they allow updates in logarithmic time, the overall running time of the algorithm can be bounded by $\mathcal{O}((N + K) \log N)$. Brown [10] showed how to reduce the space requirement to $\mathcal{O}(N)$ by storing at most one intersection event per active segment. Pach and Sharir [27] achieve the same reduction by storing only intersection points of pairs of segments that are currently adjacent on the sweep-line.

2.2 The Region-Finding Algorithm by Nievergelt and Preparata

Two neighboring segments in the y-table can also be considered as bounding edges of the region between them. This viewpoint leads to an algorithm for computing the regions formed by a self-intersecting polygon as proposed by Nievergelt and Preparata [24]. Each active segment s maintains information about the regions directly above and directly below in two lists $A(s)$ and $B(s)$. These lists store the vertices of the regions swept so far and are updated at the event points. Nievergelt and Preparata define four event types as illustrated in Figure 1.

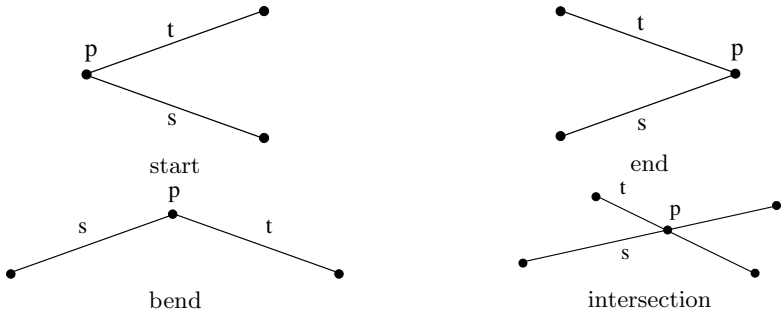


Fig. 1. Four event types in Nievergelt and Preparata's algorithm [24].

At each event point p the segments s and t are treated as in the line segment intersection algorithm described in Section 2.1. In addition, new regions are created at start and intersection events, while existing regions are closed or merged at intersection and end events. Bend events cause no topological modifications but only the replacement of s by t and the augmentation of the involved vertex lists by p. With a careful implementation the region lists can be maintained without asymptotic overhead so that the algorithm detects all regions induced by N line segments and K intersection points in $\mathcal{O}((N + K) \log N)$ time and $\mathcal{O}(N + K)$ space.

Although there are only $\mathcal{O}(N)$ active *segments* at any given time, the combinatorial complexity of all active *regions*, i.e., the total number of points on their boundaries, is slightly higher.

Lemma 1. *The combinatorial complexity of active regions at any given time during the region-finding algorithm is bounded by* $\mathcal{O}(N\alpha(N))$*, where* $\alpha(N)$ *is the inverse Ackermann function.*

Proof. [1] Consider the sweep-line at time t, cut all segments crossing the sweep-line at $x = t$, and discard all segments and fragments to the right of the sweepline. This results in an arrangement of at most N segments. All active regions now belong to one single region of this arrangement, namely to the outer region. Applying a result by Sharir and Agarwal [30, Remark 5.6] we immediately obtain an upper bound of $\mathcal{O}(N\alpha(N))$ for the combinatorial complexity of the active regions to the left of the sweep-line. The same argument obviously holds for the combinatorial complexity of the active regions to the right of the sweep-line.

Taking into account Lemma 1 we can bound the space requirement for the region-finding algorithm by $\mathcal{O}(N\alpha(N))$. To do so we modify the Bentley-Ottmann algorithm according to Brown's comments [10] and report all regions as soon as they have been swept completely.

To use this algorithm for overlaying a set of red polygonal objects and a set of blue polygonal objects we store the endpoints of the corresponding blue and red segments in the x-queue in sorted order. During the sweep we can determine for each computed region whether it is an original region, i.e., a red or blue region, or resulting from the intersection of a red and a blue region. In the latter case, we can combine the attributes of two overlapping regions by an application-specific function to obtain the attribute of each new region if we store the attribute of each region for each of its bounding segments.

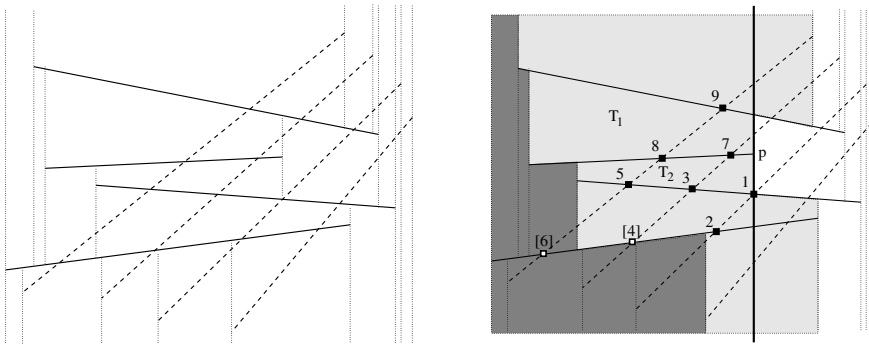
2.3 The Red/Blue Line Segment Intersection Algorithm by Chan

The algorithm described in the previous section can be used to determine the intersection points induced by any set of line segments. In many GIS applications, however, we know that the line segments in question belong to two distinct sets, each of which is intersection-free. Chan [12] proposed an algorithm that computes all K intersection points between a red intersection-free set of line segments and a blue intersection-free set of line segments with a total number of N segments in optimal $\mathcal{O}(N \log N + K)$ time and $\mathcal{O}(N)$ space. Since this algorithm forms the basic building block of an overlay algorithm which we present in the following sections, we explain it in more detail than the previous algorithms.

The asymptotic improvements are achieved by computing intersection points by looking backwards at each event instead of computing intersection points in advance and storing them in the x-queue. Since intersection events are used to determine intersection points and to update the status of the sweep-line (via exchanging the intersecting segments) it is not possible to maintain both red and blue segments in a single y-table without storing intersection events. Chan solves this conflict by introducing two y-tables, one for each set of line segments. Since segments within each set do not intersect both y-tables can be maintained without exchanging segments. The events, i.e., the endpoints of the segments,

are maintained in one x-queue, and depending on the kind and color of the event point the segment starting or ending at that point is inserted in or deleted from the corresponding y-table. For the analysis of the time complexity we assume that the table operations *insert*, *delete*, and *find* can be handled in $\mathcal{O}(\log N)$ time, and that the successor and predecessor of a given table item can be determined in constant time.

The algorithm can be explained best by looking at the trapezoidal decomposition of the plane induced by the blue segments and the endpoints of the blue and the red segments. Each endpoint produces vertical extensions ending at the blue segments directly above and below this point. These extensions are only imaginary and are neither stored nor involved in any geometric computation during the algorithm. Figure 2(a) shows the trapezoidal decomposition induced by a set of blue segments (solid lines) and the endpoints of both these and the red segments (dashed lines).



(a) Trapezoidal decomposition. (b) Sweep-line and sweep-front.

Fig. 2. Trapezoid sweep as proposed by Chan [12].

At each event p the algorithm computes all intersections between the *active* red segments and the boundaries of the active blue trapezoids that lie to the left or on the sweep-line and have not been reported yet. The trapezoids whose right boundaries correspond to the event p are then added to the imaginary sweep-front, the collection of trapezoids that have been swept completely so far. An example is depicted in Figure 2(b): all active trapezoids are shaded light grey, and the sweep-front is shaded dark grey. After having processed the event p the trapezoids T_1 and T_2 will be added to the sweep-front.

The blue boundary segments of the trapezoid(s) ending at p are processed in ascending order: starting from the red segment directly above the current blue segment we check the active red segments in ascending order for intersections to the left of the sweep-line with the current blue segment and its predecessors in the blue y-table. The algorithm locates each red segment in the blue y-table and traces it to its left endpoint while reporting all relevant intersections with active

blue segments. Note that active red segments and active blue segments might intersect outside the active trapezoids, i.e., on the boundary of trapezoids already in the sweep-front (for example, the intersection points [4] and [6] in Figure 2(b)).

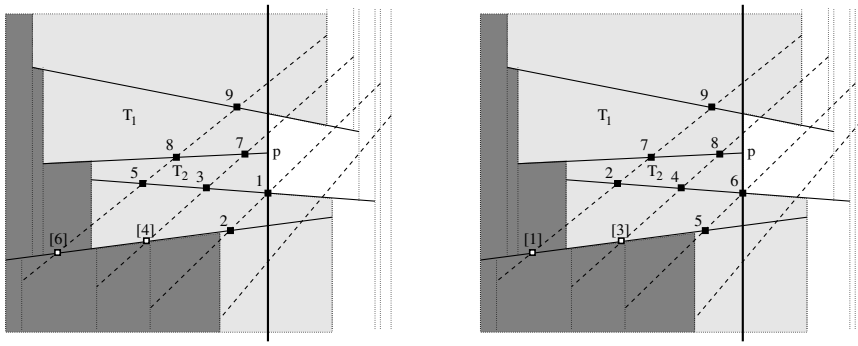
To avoid multiple computation of such points, each active red segment stores its intersection point with the largest x-coordinate (if any) and stops finding intersections for a given red segment as soon as this most recent intersection point is detected again, or if no intersection between a red segment and the current blue segment has been found. In the first situation the algorithm starts examining the next active red segment, whereas in the latter situation we know that no red segment above the current segment can have an undetected intersection with this blue segment. Therefore the red segment directly below the current blue and its predecessors in the red y-table are tested for intersection with the current blue segment. If these tests are finished we process the next blue boundary segment of a trapezoid ending at p. In Figure 2(b) the intersection points are labeled according to the order in which they are detected: black squares indicate new intersection points, and white squares indicate intersection points found during the processing of earlier events.

Since each of the N segments requires one start event and one end event and is stored in exactly one y-table, the space requirement for this algorithm is $\mathcal{O}(N)$. Initializing the x-queue and maintaining both y-tables under N insertions and N deletions can be done in $\mathcal{O}(N \log N)$ time. To find the red segment directly above each event point and to locate this segment in the blue y-table we need additional $\mathcal{O}(\log N)$ time per event. Tracing a red segment through the blue table takes $\mathcal{O}(1)$ time per step. Except for the last step per segment each trace step corresponds to a reported intersection, and there are at most two segments examined per event that do not contribute to the intersections reported at that event. Thus, the number of trace steps performed at one event point is of the same order as the number of intersections reported at that event point resulting in a total number of $\mathcal{O}(K)$ trace steps during the algorithm. The time complexity of this line segment intersection algorithm is $\mathcal{O}(N \log N + K)$, which is optimal [23]. For a complete description we refer the reader to the original paper [12].

2.4 Region-Finding Based upon Red/Blue Line Segment Intersection

The key to an efficient region-finding algorithm based upon Chan's algorithm is to use the lists $A(s)$ and $B(s)$ introduced by Nievergelt and Preparata. Recall that these lists store the vertices of the regions above and below each active segment s and are updated at the event points. Updating these lists at endpoints of segments or intersection points is done analogously to the algorithm of Nievergelt and Preparata. Since there are no explicit intersection events in Chan's algorithm, we have to ensure that these updates are done in the proper order. Consider the situation in Figure 3(a): if we try to process the intersection points in the standard order we are not able to describe the region ending at intersection point 1 since we do not know the intersection points 2 and 3 also describing the region (intersection point [4] has been detected at a previous event). If we

had detected the points in the order shown in Figure 3(b) we would have found the intersection points 4 and 5 prior to intersection point 6. To do so, we only need to report the intersection points in reverse order. It is not hard to show that intersection points are then reported in lexicographical order along each blue and red segment, and thus that all events describing a region are reported in the proper order. We omit this proof due to space constraints.



(a) Standard order. (b) Modified order.

Fig. 3. Intersection points detected during region-finding.

As with the algorithm by Nievergelt and Preparata, maintaining the region list does not asymptotically increase the time spent per event, and thus we can compute the overlay of two polygonal layers in optimal $\mathcal{O}(N \log N + K)$ time. According to Lemma 1 the space requirement can be bounded by $\mathcal{O}(N\alpha(N))$.

3 Extensions to the Overlay Algorithms to Handle Real-World Data

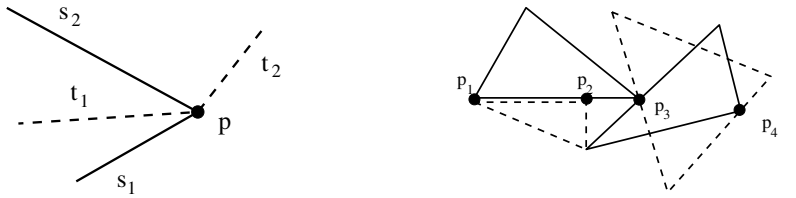
In this section we show how the region-finding algorithms of Section 2 can be extended to handle real-world configurations. In general, these data sets are "degenerate" in a geometric sense, i.e., there are many segments starting or ending at a given node, endpoints may have identical x-coordinates, or segments from different layers overlap. Section 3.1 describes how to handle such situations, and in Section 3.2 we discuss how to deal with massive data sets that are too large to fit completely into main memory.

3.1 Handling Non-general Configurations

The most frequently occurring non-general configuration consists of multiple endpoints having the same x-coordinate. Such situations can be handled easily if we implicitly rotate the data set using a lexicographical order of the endpoints. Other non-general configurations include multiple segments meeting at

one point, overlapping segments, or endpoints lying on the interior of other segments. In the following discussion we assume that all geometric computations in the algorithms can be carried out in a numerically robust way (as for example described by Brinkmann and Hinrichs [9] or by Bartuschka, Mehlhorn, and Näher [5]) and thus, that all non-general configurations can be detected consistently. A recent approach to handle degenerate cases in map-overlay has been presented by Chan and Ng [11]. Their algorithm requires two phases: first all intersections are computed by some line segment intersection algorithm (thereby passing the main part of the care for degeneracies to that algorithm), and then all sorted event points (including intersection points) are processed in a plane-sweep. Since their algorithm does not improve the $\mathcal{O}((N+K)\log N)$ time bound of Nievergelt and Preparata's algorithm and requires a separate intersection algorithm we do not consider it in this paper.

Figure 4(a) depicts a situation where an end event and an intersection event coincide at some point p. No matter which of these events is handled first the algorithm by Nievergelt and Preparata (Section 2.2) does not construct the region bounded by s_1 and t_1 and the region bounded by t_1 and s_2. Kriegel, Brinkhoff, and Schneider [21] solved this problem by replacing the four event types by two more general situations: first, all segments ending at a given point are processed in clockwise order (end event) and then all segments starting at that point are processed in counterclockwise order (start event). Bend events and intersections events are handled as a pair of one end and one start event.



(a) Segments meeting in one point. (b) Coincident points and segments.

Fig. 4. Non-general configurations.

However, for the map overlay operation there are two more details that deserve special attention: in the situation of Figure 4(b) we have two overlapping segments $\overline{p_1p_2}$ and $\overline{p_1p_3}$ of different colors and a red segment passing through a blue endpoint p_4. To avoid topological inconsistencies in the construction and attribute propagation of the empty region between $\overline{p_1p_2}$ and $\overline{p_1p_3}$ we modify these two segments such that there are two segments $\overline{p_1p_2}$ and $\overline{p_2p_3}$ by adjusting the left endpoint of the longer segment $\overline{p_1p_3}$. The resulting unique segment $\overline{p_1p_2}$ needs to store the names of both the blue region above and the red region below. To detect segments passing through an endpoint, at each event p we locate the current endpoint in the y-table of the other color and check whether it lies on an open segment. In this case we split that segment into two new segments

(one ending at p and one starting at p) and proceed as usual. It is easy to see that these modifications guarantee topological consistency and require only a constant number of operations per event, thereby not affecting the asymptotic time complexity.

3.2 Handling of Massive Data Sets

The handling of massive data sets is a well-known problem in GIS and the primary focus of investigation in the field of external memory algorithms (see Vitter [31] for a recent survey). In this section we explain how the algorithms for the overlay of polygonal maps presented above can be used for the map overlay even when dealing with massive data sets.

As noticed by several authors [3,19,26] real-world data sets from several application domains seem to obey the so-called "$\sqrt{N}$-rule": for a given set of N line segments and a fixed vertical line L there are at most $\mathcal{O}(\sqrt{N})$ intersections between the line segments and the line L. As a result, data sets obeying this rule have at most $\mathcal{O}(\sqrt{N})$ active segments that need to be stored in an internal memory y-table. The TIGER/Line data set for the entire United States is estimated to contain no more than 50 million segments [3], and we have $\sqrt{50,000,000} \approx 7,000$.

Both overlay algorithms presented in Section 2 allow for space-efficient storage of the computed overlay. As soon as a region of the resulting layer has been swept completely it does not need to be present in internal memory anymore and can be transferred to disk. As long as no resulting region contains too many boundary segments, there is a fair chance that even for massive data sets both active segments and active regions fit into main memory. In contrast, if we aim at constructing a network oriented layer for the resulting partition [14,23] we are forced to keep the complete data structure in main memory.

Conceptually, the x-queue can be thought of as an input stream that feeds the algorithm, and due to the size of the input data set this stream will be resident on secondary storage. To optimize access to this x-queue its elements should be accessed sequentially, i.e., one should avoid random access as needed for update operations. The basic algorithm by Nievergelt and Preparata does not fulfil this property since intersection events are detected while sweeping and need to be inserted in the x-queue. However, by applying Brown's modification and thus storing at most one intersection event per active segment we can avoid such updates during the sweep. Instead, we store all detected intersection events in an internal memory priority queue. Whenever we find a new intersection event p, say an intersection between the segments s and t, we check whether the events possibly associated with s and t are closer to the sweep-line than p. If any of these event lies farther away from the sweep-line than p, we remove this event from the internal memory priority queue and insert p instead. At each transition the sweep algorithm has to choose the next event to be processed by comparing the first element in the x-queue and the first element in the priority queue storing the intersection events. This internal memory priority queue can be maintained in $\mathcal{O}((N+K)\log N)$ time during the complete sweep and occupies $\mathcal{O}(\sqrt{N})$ space under the assumption that the data sets obey the "$\sqrt{N}$-rule". Our

modification of Chan's algorithm does not require such an additional internal memory structure since there is no need for updating the x-queue during the sweep.

If we use the extension proposed in Section 3.1 for handling non-general configurations, we are forced to modify endpoints of active segments or to split segments which triggers the need for updating the x-queue. This situation can be handled by using an internal memory priority queue similar to the one proposed above. If we encounter two overlapping segments, we adjust the left endpoint of the longer one and insert it in the internal memory event queue. Since both original segments are active we simply move this modified segment from the y-table to the priority queue without increasing internal memory space requirements. For each segment split due to handling non-general configurations, we can charge one endpoint of the active segments coincident with the split point for the new segment inserted into the priority queue. Since each active segment can contribute to at most two splits the internal memory space requirement remains asymptotically the same. Synchronizing the x-queue and the priority queue is done as described above.

4 Empirical Results

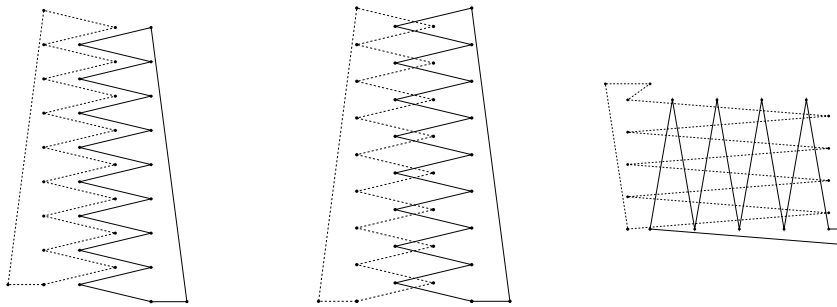
In this section we discuss our implementation of the algorithms described in Section 2.2 and Section 2.4 and the results of experiments with artificial and real-world data sets. Our focus is on the comparison of algorithms having different asymptotical complexities. The comparison is based on real-world situations and artificial situations simulating special cases. According to Section 3.2 we expect identical I/O cost for both algorithms. Hence, we ran the algorithms on data sets ranging from small to quite large comparing the absolute internal memory time complexities of both algorithms.

The algorithms have been implemented in C++, and all experiments were conducted on a Sun$^{\text{TM}}$ Ultra 2 workstation running Solaris 2.5.1. The workstation was equipped with 640 MB main memory, to ensure that even without implementing the concepts from Section 3.2 all operations are performed without causing page faults or involving I/O operations. For a more detailed description of the implementation we refer the reader to Giesen [18].

4.1 Artificial Data Sets

The first series of experiments concentrated on the behavior of both algorithms for characteristic values of K. To this end we generated three classes of input data with $K = 0$, $K \in \Theta(N)$, and $K \in \Theta(N^2)$ intersection points, respectively, as shown in Figure 5. The results of the experiments are presented in Table 1.

As a careful analysis shows the algorithm from Section 2.4 performs three times as many intersection tests as the algorithm by Nievergelt and Preparata resulting in a slow-down by a factor of three.



(a) $K = 0$. (b) $K \in \Theta(N)$. (c) $K \in \Theta(N^2)$.

Fig. 5. Artificial data sets.

For $K \in \Theta(N)$ the more efficient reporting of our algorithm cancels out that drawback, and for $K \in \Theta(N^2)$ we see a clear superiority of our algorithm reflecting the theoretical advantage of $\mathcal{O}(N \log N + N^2) = \mathcal{O}(N^2)$ versus $\mathcal{O}((N + N^2) \log N) = \mathcal{O}(N^2 \log N)$. The speed-up varies between factors of 10.4 and 17.8, and the more segments (and thus intersection points) are involved the more our algorithm improves over Nievergelt and Preparata's algorithm.

N	$K = 0$			$K \in \Theta(N)$			$K \in \Theta(N^2)$		
	[24]	Section 2.4	Ratio	[24]	Section 2.4	Ratio	[24]	Section 2.4	Ratio
256	0.3	0.7	0.4	0.7	0.7	1	51	5	10.4
512	0.6	1.6	0.4	1.6	1.7	1	228	18	12.4
1,024	1.2	3.6	0.3	3.8	3.6	1	1,033	72	14.4
2,048	2.7	8.0	0.3	8.3	8.0	1	4,401	282	15.6
4,096	5.8	17.8	0.3	18.1	17.8	1	19,939	1195	16.7
8,192	12.6	38.5	0.3	39.4	38.7	1	89,726	5,019	17.8

Table 1. Overlaying artificial maps (time in seconds).

4.2 Real-World Data Sets

After having examined the behavior of the two algorithms under varying parameters N and K, we tested the described algorithms for real-world data sets. We ran a series of experiments with polygonal maps of cities and counties from the state of North-Rhine Westphalia (NRW) and Germany (GER). Vectorized maps obtained from different distributors are likely to unveil different digitizing accuracies, and as can be seen from Figure 6 such differences bring about so-called "sliver polygons".

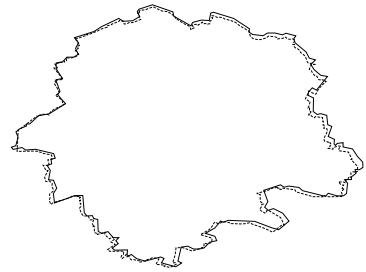


Fig. 6. Real-world data set: City of Münster, Germany.

Detecting and post-processing sliver polygons is an important issue in map overlay, and there are specialized methods for this task depending on the application at hand. At this point we only mention that the region-finding algorithms can be extended in a straightforward way to serve as a filter for sliver polygons because they produce all regions while sweeping.

To simulate additional sliver polygons we performed experiments where we overlayed a data set with a slightly translated second data set. The running times for overlaying real-world maps are summarized in Table 2.

Map 1	Map 2	N	[24]	Section 2.4	Ratio
NRW (counties)	NRW (counties) +	23,732	22	51	0.43
NRW (cities)	NRW (counties)	66,436	62	142	0.45
NRW (cities)	NRW (cities) +	109,140	113	256	0.45
GER (cities)	GER (counties) +	745,544	945	2036	0.48
GER (cities)	GER (counties)	745,544	657	1026	0.67
GER (cities)	GER (cities) +	1,238,538	2216	3559	0.63
GER (cities)	GER (cities)	1,238,538	988	1151	0.90

Table 2. Overlaying real-world maps (time in seconds, "+" indicates translated data set).

These figures show an advantage for Nievergelt and Preparata's algorithm by a factor of up to 2.3. This result could have been preempted by the results from Section 4.1, however, since there are less than linearly many intersection points in all of these experiments. In analogy to the above observations, our proposed algorithm improves with increasing number of segments involved.

Summarizing the experiments from both this section and Section 4.1 we see an advantage for the conventional method by Nievergelt and Preparata as long as there are at most linearly many intersection points. For large data sets this

advantage becomes smaller, and there is a clear superiority for our proposed algorithm when dealing with data sets producing more than linearly many intersection points.

5 Conclusions

In this paper we have examined algorithms for performing map overlay operations with massive real world-data sets. We have proposed to modify plane-sweep algorithms to cope with such data sets. It is obvious that such algorithms may also be used to implement the refinement step of a spatial join efficiently. However, this approach is not useful if the filter step produces only few candidates which must be checked in the refinement step, since the plane-sweep algorithms may require to store theses candidates on disk for sorting. We have not yet addressed this problem in our research, but obviously it is not efficient to use a plane-sweep algorithm if each polygon of the two input maps only participates in at most one candidate pair.

Our experiments have shown that the modified algorithm of Chan performs better than the algorithm of Nievergelt and Preparata, if there are more than linearly many intersections points. On the other hand the algorithm of Nievergelt and Preparata performs better than the algorithm of Chan, if there are sublinearly many intersection points.

In our future work we will focus on the realization of the methods proposed in Section 3.2 and on the combination with existing algorithms for the filter step of the spatial join operation. This also includes a comparison of using plane-sweep algorithms in the refinement step with a simple checking of each pair reported by the filter step for an intersection.

References

1. P. Agarwal. Private communication. 1999.
2. D. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. Further comparison of algorithms for geometric intersection problems. In T. Waugh and R. Healey, editors, *Advances in GIS Research – Proceedings of the 6th International Symposium on Spatial Data Handling (SDH '94)*, volume 2, 709–724, 1994.
3. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. Vitter. Scalable sweeping-based spatial join. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98: Proceedings of the 24th International Conference on Very Large Data Bases*, 570–581. Morgan Kaufmann, 1998.
4. I. Balaban. An optimal algorithm for finding segment intersections. In *Proceedings of the 11th Annual ACM Symposium on Computational Geometry*, 211–219, 1995.
5. U. Bartuschka, K. Mehlhorn, and S. Näher. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. Online-Proccedings of the First Workshop on Algorithm Engineering <http://www.dsi.unive.it/~wae97/proceeedings/ONLY_PAPERS/pap13.ps.gz>, accessed 7 Jul. 1998, 1997.

6. J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979.
7. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In P. Buneman and S. Jajoda, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22.2 of *SIGMOD Record*, 237–246. ACM Press, June 1993.
8. T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In R. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23.2 of *SIGMOD Record*, 197–208. ACM Press, June 1994.
9. A. Brinkmann and K. Hinrichs. Implementing exact line segment intersection in map overlay. In T. Poiker and N. Chrisman, editors, *Proceedings of the Eighth International Symposium on Spatial Data Handling*, pages 569–579. International Geographic Union, Geographic Information Science Study Group, 1998.
10. K. Brown. Comments on "Algorithms for reporting and counting geometric intersections". *IEEE Transactions on Computers*, C-30(2):147–148, 1981.
11. E. Chan and J. Ng. A general and efficient implementation of geometric operators and predicates. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases — Proceedings of the Fifth International Symposium on Spatial Databases (SSD '97)*, volume 1262 of *Lecture Notes in Computer Science*, 69–93. Springer, 1997.
12. T. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proceedings of the 6th Canadian Conference on Computational Geometry*, 263–268, 1994.
13. B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.
14. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 1997.
15. Environmental Systems Research Institute, Inc. ARC/INFO, the world's GIS. ESRI White Paper Series, Redlands, CA, March 1995.
16. U. Finke and K. Hinrichs. The quad view data structure: a representation for planar subdivisions. In M. Egenhofer and J. Herring, editors, *Advances in Spatial Databases — Proceedings of the Fourth International Symposium on Spatial Databases (SSD '95)*, volume 951 of *Lecture Notes in Computer Science*, 29–46, 1995.
17. W. Franklin. Adaptive grids for geometric operations. In *Proceedings of the Sixth International Symposium on Automated Cartography (Auto-Carto Six)*, volume 2, 230–239, 1983.
18. A. Giesen. Verschneidung von Regionen in Geographischen Informationssystemen (Overlaying polygonal regions in geographic information systems). Master's thesis, University of Münster, Dept. of Computer Science, November 1998. (in German).
19. R. Güting and W. Schilling. A practical divide-and conquer algorithm for the rectangle intersection problem. *Information Sciences*, 42:95–112, 1987.
20. Y.-W. Huang, M. Jones, and E. Rundensteiner. Improving spatial intersect using symbolic intersect detection. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases — Proceedings of the Fifth International Symposium on Spatial Databases (SSD '97)*, volume 1262 of *Lecture Notes in Computer Science*, 165–177. Springer, 1997.
21. H.-P. Kriegel, T. Brinkhoff, and R. Schneider. An efficent map overlay algorithm based on spatial access methods and computational geometry. In *Proceedings of the International Workshop on DBMS's for Geographic Applications*, 194–211, Capri, May 12–17 1991.

22. M.-L. Lo and C. Ravishankar. Spatial joins using seeded trees. In R. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23.2 of *SIGMOD Record*, 209–220. ACM Press, June 1994.
23. H. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 of *NATO ASI*, 307–325. Springer-Verlag, 1988.
24. J. Nievergelt and F. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, 1982.
25. J. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In H. Garcia-Molina and H. Jagadish, editors. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, volume 19.2 of *SIGMOD Record*, pages 343–352. ACM Press, June 1990.
26. T. Ottmann and D. Wood. Space-economical plane-sweep algorithms. *Computer Vision, Graphics and Image Processing*, 34:35–51, 1986.
27. J. Pach and M. Sharir. On vertical visibility in arrangements of segments and the queue size in the Bentley-Ottmann line sweeping algorithm. *SIAM Journal on Computing*, 20(3):460–470, 1991.
28. J. Patel and D. DeWitt. Partition based spatial-merge join. In H. Jagadish and I. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25.2 of *SIGMOD Record*, 259–270. ACM Press, June 1996.
29. F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, Berlin, 2nd edition, 1988.
30. M. Sharir and P. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, Cambridge, 1995.
31. J. Vitter. External memory algorithms. In *Proceedings of the 17th Annual ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '98)*, 119–128, 1998. invited tutorial.