TUFTS-CS Technical Report 2003-05

November 2003

Topologically Sweeping the Complete Graph in Optimal Time and Space

by

Eynat Rafalin
Department of Computer Science
Tufts University
Medford, MA 02155
and
Diane L. Souvaine
Department of Computer Science
Tufts University
Medford, MA 02155

ABSTRACT

Reporting all intersections of line segments and their characteristics is one of the early and most fundamental problems in computational geometry. We concentrate on the problem of reporting all intersections in an embedding of a complete graph that may contain degeneracies such as vertical lines, or multiply intersecting lines. A graph presents some difficulties for a sweep line algorithm that most of the existing sweep based algorithms do not handle.

We present a novel approach that sweeps a complete graph of N vertices and k intersection points in optimal $O(k) = O(N^4)$ time and $O(N^2)$ space, that is simple and easy to code. The algorithm sweeps the graph using a topological line, borrowing the concept of horizon trees from the topological sweep method [8] and using ideas from [13] to deal with degeneracies. The novelty in our approach is the use of the *moving wall* that separates at any time the graph into two regions: the region of known structure, in front of the moving wall, and the region that may contain intersections generated by edges that have not yet been registered in the sweep process, behind the wall. The algorithm has applications in computing the simplicial depth median for a set of n data points [1]. The paper includes the algorithm, its analysis and experimental results.

1 Introduction

The problem of reporting all intersections of line segments and their associated characteristics is one of the early and most fundamental ones in computational geometry: Given an arrangement of n objects, report all k intersection points. We concentrate on the problem of reporting all intersection points in an *embedding of the complete graph* on N vertices using a sweep line method. This problem has application to graph drawing and for depth-based statistical analysis, for computing the simplicial depth median for a set of N data points [1].

A graph presents several difficulties for a sweep-line algorithm, that are not present in an arrangement of lines, and that most of the existing sweep techniques do not handle. Clearly, the structure holding the sweep-line status has to be dynamic, since vertices and edges are constantly inserted and deleted. In addition, the event points where the status of the sweep line changes now have two types and include the intersection points (where segments swap order) and the vertices of the graph (where segments are inserted and removed). Any sweep algorithm must process both event-point types, but with different steps. Moreover, since some edges may extend further in the embedding than others, some *short* edges not yet encountered by the sweep line may create intersections that should be processed before intersections created by *long* edges (see Section 3.5). Processing in the wrong order may introduce intersections not in the graph or ignore others, causing the sweep to terminate unsuccessfully.

We present a novel approach that sweeps a complete graph of N vertices and k intersections in optimal $O(k) = O(N^4)$ time and $O(N^2)$ space, that is simple and easy to code. The sweep uses a topological line, borrowing the concept of the horizon trees from the topological sweep method [8], processes each intersection in amortized constant time and keeps the optimal time and space. Degeneracies are handled using ideas from the enhanced topological sweep algorithm [13]. The key innovation in our approach is the use of the *moving wall* that separates at any time the graph into two regions: the region in front of the moving wall with known structure, and the region behind the wall that may contain intersections generated by edges not yet registered in the sweep process.

Section 2 surveys previous work. Section 3 presents an overview of the sweep algorithm and Section 4 its analysis. Sections 5 and 6 contain experimental results and future work.

2 Previous Work

Plane sweep, first introduced by Bentley and Ottmann [4], which sweeps a vertical line from left to right, is a popular technique to report all intersection points in an arrangement of lines, segments or objects. It achieves a time complexity of $O((n+k)\log n)$ and requires $O(n+k)$ space for an arrangement of n segments and k intersections. Brown [5] improved the space requirement to $O(n)$. For the complete graph with N vertices and $k = O(N^4)$ intersections, the performance of $O(N^4 \log N)$ time and $O(N^2)$ space adds a $\log N$ factor to the optimal time complexity.

Nievergelt and Preparata [11] used plane sweep to merge two *planar* graphs with total of N vertices and k intersection points. They showed that if the convexity property holds, this merge takes $O(N \log N + k)$ time. Their method uses a line that is not necessarily straight and is one of the few that directly deals with the vertices of the graph (as event points

that require special processing). Later Mairson and Stolfi [10] extended the result for a subdivision with not necessarily convex regions. These techniques do not handle non-planar graphs.

Chazelle and Edelsbrunner [6] presented an optimal deterministic algorithm for the intersection reporting problem on an arrangement of n segments in $O(n \log n + k)$ time and $O(n + k)$ space. The algorithm breaks segments into pieces and uses line sweep techniques on vertical strips of the arrangement. Balaban [3] proposed an $O(n \log n + k)$ time and linear space algorithm that also works on vertical strips, but traverses the strip tree instead of sweeping the arrangement. On the complete graph with N vertices, the time complexity of both algorithms will be dominated by $O(k) = O(N^4)$. The problem with both algorithms is their complexity to implement.

The *topological sweep* of Edelsbrunner and Guibas [8] sweeps an arrangement of n planar lines in $O(n^2)$ time and $O(n)$ space with a topological line instead of a vertical one. It reports the intersection points of the lines according to a partial order related to the levels in the arrangement. Edelsbrunner and Souvaine [9] extended the technique to *guided topological sweep*, which forces additional constraints needed by other problems. Useful variants like *topological walk* [2] and *topological peeling* [7], for sweeping only a convex subset of the arrangement were suggested and implemented. Recently, Rafalin *et al.* [13] presented a modification to the algorithm that handles degeneracies, such as parallel lines and multi-intersection points, creating an output-sensitive algorithm, with time complexity dependent on the size of the degenerate arrangement. Topological sweep offers a $\log n$ improvement factor over the vertical line sweep on an arrangement of n infinite lines. However, it does not work for finite line segments.

3 Algorithm Overview

Let $G = (V, E)$ be a complete graph on N vertices, embedded in the plane. Assume the graph vertices are in general position. The complete graph contains exactly $n = \frac{N(N-1)}{2}$ edges and k is $O(N^4)$. The number of edges along the sweep line at any time is $O(N^2)$ but can be $\Theta(N^2)$: a sweep line could have $N/2$ vertices on each side, with $(\frac{N}{2})^2$ edges crossing the line.

We sweep G from left to right to report all intersection points using a topological line (*cut*): a monotonic line in y-direction, intersecting each of the n edges *at most* once (rather than exactly once, as for an arrangement of infinite lines). *Active edges* intersecting the sweep line are exactly the edges that connect a vertex left of the sweep line to a vertex on its right. This set is dynamic and an edge can be added and removed from it only once. Since the edges of the graph are finite the cut can intersect an arbitrary number of edges, from 0 to $(\frac{N}{2})^2$.

3.1 Active Segments and the Cut

The cut is specified by the sequence of *active segments*, one per edge intersected by the topological line. A segment of an edge is delimited by two adjacent intersection points or by the rightmost/leftmost intersection point and the vertex of the graph that is right/left end-point of the edge. An *active segment* for edge e is defined by extending the segment

of e currently crossed by the cut to the right until it intersects another active edge or a vertex of the graph. The active segment is therefore delimited from left and right by another intersection point with an active edge or by a vertex of the graph. Since at any position of the cut there is a one-to-one correspondence between the active segments and the active edges the terms will be used interchangeably.

A sweep is implemented by starting with the leftmost cut, which intersects no edges of the graph (for comparison, the leftmost cut of the topological sweep algorithm includes all semi-infinite edges starting at -∞) and pushing it to the right in a series of elementary steps until it becomes the rightmost cut. An elementary step consists of the topological line sweeping past a vertex of the graph all of whose incoming edges are currently intersected by the cut or past an intersection of two or more edges that are consecutive in the current cut (a *ready* intersection), see Lemma 3.

The set of active edges crossed by the sweep line at each step is not static (as in the original topological sweep algorithm) but dynamic, since edges are constantly added to and deleted from it. A dynamic data structure, like a linked list, needs to be used to store the active segments currently crossed by the cut (whereas the original algorithm uses a static array of edges).

In our data structure, each active segment is defined by the edge that contains it, represented by its two end-points, and by the segments that form its left and right end-points. Some active segments may be delimited not by other active segments but by a vertex of the graph or a segment that is no longer active, complicating the data structure. Pointers to vertices are implemented by using another array of n degenerate segments, representing the vertices. If an active segment points to a deleted one, a dummy segment is declared and then deleted, when the segment itself is deleted.

3.2 Horizon Forests

To achieve optimal time complexity, linear in the size of the arrangement, while maintaining space complexity that is linear in the maximal size of the cut, we build upon the concept of *horizon trees* introduced by Edelsbrunner and Guibas [8]. Our horizon graphs are often not trees but *horizon forests*. The upper (respectively lower) *horizon forest* of the cut is constructed by extending the cut edges to the right. When two edges intersect, only the one of higher (respectively lower) slope continues to the right (see Fig. 1). Given the lower and upper *horizon forests*, the cut is identified by the intersection of the two right delimiters of the forests, in constant time per active edge.

The set of edges of the *horizon forests* corresponds to the set of active edges/segments. The initial horizon forests are empty, and are updated in each event point (see below). Keeping an amortized constant update time per event point is the key in achieving the optimal time complexity.

3.3 Intersection Event Points

Edge Changes: In an intersection event point, the active segments switch position, carrying with them all the information associated with each active segment. In addition changes are needed to the *horizon forests'* edges. Once the updated *horizon forests'* edges are computed, the new cut can be computed in constant time and new ready intersection points discovered.

To update the upper (resp. lower) *horizon forest* after processing the intersection point of segments $s_i, \ldots s_{i+k}$, we need, for each segments s of the intersection apart from the first (resp. last) one, to traverse the *bay* formed by the segments above (below) s, starting from the segment immediately above (below) s until we encounter the segment that intersects s. The traversal is done by walking from one segment to the segment that delimits its *horizon forest's* edge to the right, until the intersection with s. If s's right endpoint precedes the intersection, the right delimiter of the *horizon forest* is set to be the right endpoint. If the forest edge associated with segment l, along the bay of s, terminates in a vertex v_l, then if the intersection of l and s along the bay precedes v_l, l is the right delimiter of s. Otherwise, l terminates before it encounters s and the right delimiter of the horizon forest edge of s is the vertex that is s's right end-point (see Figure 1 and [8]).

Degeneracies: Dealing with degeneracies is achieved by borrowing ideas from the enhanced topological sweep algorithm of Rafalin *at al.* [13]. First, an additional modification to the cut is needed, by representing two right endpoints of each active segment l_i, $r_{up,i}, r_{down,i}$. If the right endpoint of an active segment is generated by its intersection with a segment from above (resp. from below), then $r_{up,i}$ (resp. $r_{down,i}$) is pointing to that segment; otherwise $r_{up,i}$ (resp. $r_{down,i}$) is `null`. At least one is non-null, but both may be in instances where the right endpoint of the active segment is the intersection point of three or more edges. Given the lower and upper horizon forests, these delimiters are computed in constant time.

Discovering Ready Intersections: Define a *matching pair* as a pair of consecutive active segments l_i, l_j in the cut where $r_{up,i}$ is l_j and $r_{down,j}$ is l_i. When at most two segments participate in an intersection, a *matching pair* implies a *ready intersection.* For the more general case, we define a *matching sequence* of consecutive active segments $l_i, ..., l_j$ in the cut where every adjacent pair of segments forms a *matching pair.* A *ready intersection* is generated by a *complete matching sequence* where the bottom segment is l_i and the top is l_j and in which $r_{down,i}$ and $r_{up,j}$ are both *null.* The set of segments along any cut that contain the same intersection point as their right endpoint form at most one connected component (*the connected component property*), see [13].

After each update to the cut, we test whether two newly adjacent active segments form a matching pair. If so, this new pair either augments a *matching sequence* that was already discovered or initializes a new one. Since a sequence can be discovered a few times, for each active edge s we hold pointers to the uppermost and lowermost active edges that currently share the same right endpoint as s, in a pointer called $MATCH(s)$. $MATCH(s)$ is initialized when s first becomes active, and is reset to itself every time it participates in an elementary step. $MATCH$ is updated at the conclusion of each elementary step to detect new alignment of right endpoints, already represented in $MATCH$ or initializes a new one. Updating $MATCH$ and checking whether the *matching sequence* is not *complete* takes constant time (see Lemma 4 and [13]).

3.4 Vertex Event Points

The sweep processes the vertices according to the order in which they are encountered, from left to right, requiring presorting at a cost of $O(N \log N)$. A vertex is processed only after all its incoming segments are active. In the algorithm, one more constraint is added: vertices are processed only after all ready intersection points are swept and the stack of ready intersection

points is empty.

When a vertex v is encountered, all active edges $\{\overline{uv} | u_x < v_x\}$ are deleted from the set of active segments and edges $\{\overline{vu} | u_x > v_x\}$ are added to it. This is done in constant time per active edge, if an initial pointer to the top-most and bottom-most deleted segments is given.

The horizon forests are updated as follows: The deleted edges of the forests are the in-edges of the event vertex v, and terminate in v. They therefore cannot abut any other active segment from the right, allowing their safe deletion. The inserted edges are the out-edges of v. To update the upper (resp. lower) horizon forest they are inserted in decreasing (increasing) order of slope into the structure. To insert an edge we begin at its endpoint on the left boundary, which is v. We walk in counterclockwise order around the bay formed by the previous edges to find the intersection point with an active edge (similar to the update for an intersection point, see Section 3.3).

The edges emanating from the new vertex to the right, added to the set of active edges, might cut some of the existing active segments, changing the horizon forests and the cut. It is crucial that these changes will be recorded by the algorithm, while not changing the time complexity. For example, the graph in Figure 1 contains 5 vertices. After vertex 0 is processed both the upper and lower horizon forests contain edges emanating from 0 to the right. Next vertex 1 is processed, deleting edge $\overline{01}$ and adding edges $\overline{12}, \overline{13}, \overline{14}$. The edges of the horizon forests that correspond to these 3 new edges can be updated, by traversing the *bay*. However, the addition of edge $\overline{13}$ changed the upper horizon forest edges also for edges $\overline{02}, \overline{04}$, since they are now cut by $\overline{13}$.

In general all active segments can be affected by this change: the segments below the new segments can be cut in the upper horizon forest (cut by the bottom-most newly added segment) and the segments above the newly added segments can be changed in the lower horizon forest (cut by the top-most newly added segment). Updating the entire horizon forests costs a constant time per forest edge (or active segment), since only a single test is needed, checking if the forest edge is cut by the new segment to the right or to the left of its current right endpoint. After the horizon forests are updated another update is needed to the cut, again charging a constant time per updated edge, and at most a cost that is linear in the number of active segments.

3.5 A Moving Wall

The set of active edges does not span the whole arrangement. As a consequence, the sweep can encounter intersection points created by active edges that should not yet be processed, since they are located *behind* hidden edges, not yet encountered by the algorithm.

For example, Figure 2(a) contains a graph of 7 vertices. Assume the sweep line is positioned on the dotted green line. The bold blue segments are the active segments. Intersection B of $\overline{06}$ and $\overline{25}$ is ready to be processed, as it is the right endpoint of both active segments associated with these edges, that are adjacent along the cut. However, B cannot be processed yet, as $\overline{06}$ and $\overline{25}$ intersect $\overline{34}$ before point B. In addition, if intersection B will be processed, $\overline{06}$ and $\overline{25}$ will switch position in the cut, causing $\overline{34}$ to be inserted incorrectly. **Intersection points that are to the right of any edge that is not yet active cannot**

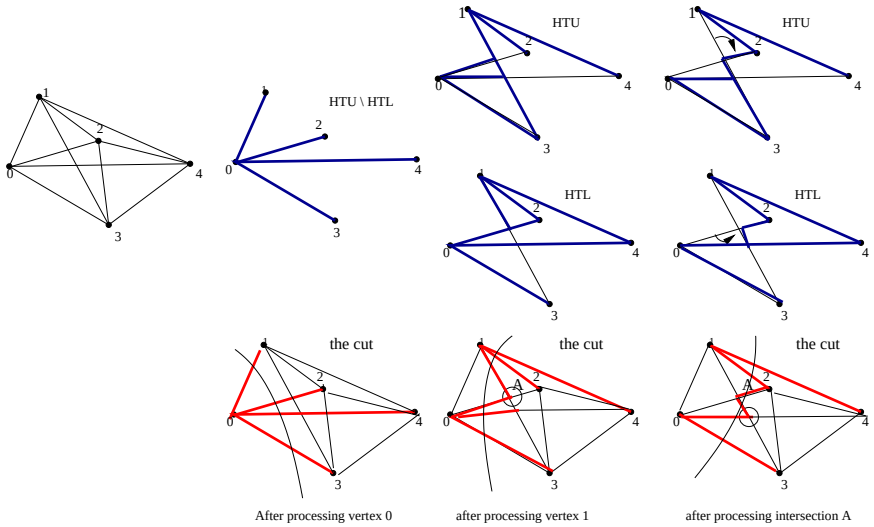


Figure 1: An arrangement of 5 vertices demonstrates the structure of the upper and lower horizon forests and the cut. (0) The arrangement; (1) Processing vertex 0, no intersection is ready; (2) Processing vertex 1. Edges $\overline{12}, \overline{13}, \overline{14}$ are added, The upper horizon forest segments for $\overline{02}$ and $\overline{04}$ below the bottom-most newly added edge $\overline{13}$ need updating, since they are cut by this edge. Since $\overline{13}$ and $\overline{02}$ are adjacent, intersection A is ready. (3) Processing intersection A. Intersection B is discovered to be ready.

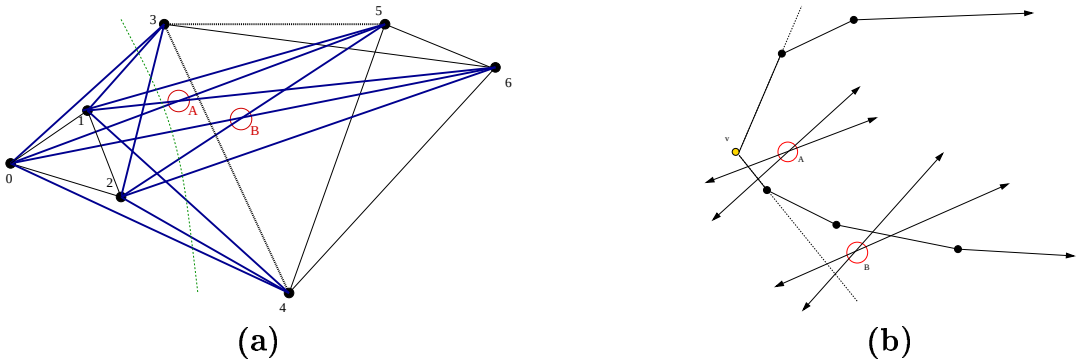
be ready [1].

On the other hand, consider intersection A in Figure 2(a). Considering the status of the cut, both this intersection and vertex 3 are ready to be processed (A is even located to the right of vertex 3, which means that in a vertical line sweep algorithm vertex 3 would have definitely been processed prior to A). However, the order of the cut edges currently places the edge $\overline{16}$ over edge $\overline{05}$. If vertex 3 will be processed at this time, the update of the horizon forests and the cut will be incorrect. **All intersection points that are to the left of all segments emanating from vertex v must be processed before vertex v is processed**.

Define a *moving wall* of a position of the sweep line as the semi infinite lines corresponding with the two extreme edges emanating to the right from the next vertex to be swept v_{next}. The moving walls for all vertices can be computed as an initialization step in $O(N \log N)$ time using the incremental convex hull algorithm [12]. At any time, the sweep line has to be forced to the current position of the moving wall, as discussed above [2]. This is done by ensuring that no intersection that is inside the moving wall can be ready (constant time per check) and that ready intersections that affect the moving wall are swept before the next vertex is processed.

[1] An intersection i is to the *left* of edge e if the x-coordinate of i is smaller than or equal to the x-coordinate of the infinite line defined by e with the horizontal line passing through i. The intersection is to the *right* otherwise.

[2] The idea of aligning the sweep line to the arrangement was mentioned by Edelsbrunner and Souvaine [9], referring to vertical alignments of pairs of vertices. In their paper the alignment is done using an *alignment graph*, a graph whose nodes are the cut edges and whose arcs connect pairs of vertices that need to be aligned. If the alignment graph is not *too big* the topological sweep can be performed with no increased cost.



(a) (b)

Figure 2: (a) A graph of 7 vertices with the sweep line positioned on the dotted green line. The bold blue segments are the set of active segments. The next vertex to be processed is 3. The associated moving wall is drawn as a bold line. (b) Moving wall and moving chain: A subset of a graph, with the moving wall (dashed) and moving chain associated with vertex v. For clarity not all vertices and edges are drawn.

Construct the *lower (resp. upper) active chain* as follows: start with v_{next} and pick the lowermost (resp. uppermost) edge emanating from it to the right. As long as the endpoint of the chain is not the right-most vertex, add the lowermost (uppermost) edge emanating from the current endpoint of the chain to the right. Define the *active chain* to be the union of both upper and lower chains (see Figure 2(b)). There might be intersections that are inside the moving wall associated with v_{next}, but can still be *legally* swept, since they are outside the moving chain (for example intersection B in Figure 2(b)). We could have forced the sweep line to advance all the way up to the moving *chain* corresponding to v_{next}, instead of only to the moving *wall*. However, an intersection that is ready relative to the set of active segments, and that is located to the right of most of the segments in the active chain, will have to be compared to every one of these segments before we could conclude if it is ready or not, raising the time complexity of the step. Instead, we choose to *force* the sweep line only to the wall created from the two extreme segments emanating to the right from v_{next}, and their extension to infinity. In this case, intersections that are outside the moving chain but inside the moving wall will be discarded (see Figure 2(b)). They will be rediscovered when v_{next} will be swept, by adding a step of checking every pair of adjacent active segments in the cut for ready intersections. The cost of this step is linear in the size of the active set, however, it is performed only a linear number of times in N (once for each vertex), keeping the total time complexity unaffected.

4 Algorithm Analysis

4.1 Data Structures

The algorithm uses 3 main data structures: $V[]$, holding the vertices of the graph, E representing the sweep line, and $I[]$, holding the ready intersection event points.

$V[]$ is a static array holding the vertices of the graph, represented with their x and y coordinates and sorted according to their x coordinate. In addition, every vertex holds the moving wall associated with it (computed at the initialization step).

E, the cut, is a linked list of `segments`, ordered according to their y intercept with the sweep line, allowing insertions, deletions and swaps to be done in constant update time [3]. Each `segment` is specified by the index of its begin and end vertices (`begin_point and end_point`) in $V[]$ and includes pointers to the segments above and below it and to the following data:
`Cut_right_up, Cut_right_down, Cut_left` - pointers to `segments` delimiting the cut associated with this segment, from the left and right.
`HFL_right, HFL_left, HFU_right, HFU_left` - pointers to the `segments` delimiting the edges of the upper and lower horizon forests associated with this segment from the left and the right.
`MATCH_up, MATCH_down` - pointers to the uppermost and lowermost active `segments` that currently share the same right endpoint as the current segment.

Some of the active segments may be delimited not by an active segments, but instead by a vertex. Pointers to vertices are implemented by using an array of n degenerate `segments`, called `point_segments[]`, where the `begin_point` and `end_point` are the same index. For the right endpoints of the cut edges `Cut_rights`, an additional pointer to an `empty` or `NULL` segment is used.

When the sweep advances, the left delimiters of some of the active segments might be deleted from the active list, causing a pointing error. The pointing error is avoided by creating a dummy segment for any active segment pointing to a deleted one, and is deleted when the segment itself is deleted. This can be achieved since the algorithm uses only the immediate information stored by the segment, and not its location in the linked list of segments.

$I[]$ is a stack (or a linked list) of pairs of pointers to segments that correspond to intersection points that are currently *ready* to be processed. If (s_i, s_j) is in I then the segments between s_i and s_j share a common right endpoint and represent a legal next move for the topological line. $I[]$ holds only the intersection event points. Vertex event points are stored in an additional structures as the index to the next vertex to be swept v_{next}. Vertex event points are processed only after all possible intersection events points (outside the moving wall) have been processed.

4.2 Analysis

The appendix contains a pseudo code of the detailed algorithm. This section presents its analysis.
Progress of the Sweep Line: Lemmas 1, 2, 3 explain the progression of the sweep line, proving that every intersection point will be traversed exactly once. The first two lemmas are proved (using an example) in Section 3.5.

Lemma 1. Intersections that are to the right of any edge that is not yet active cannot be ready.

[3]The topological sweep algorithm uses several static arrays instead of one linked list that stores all the relevant data. In that implementation the array $E[]$ stores the lines of the arrangement and the order of the edges along the cut is maintained using an array holding the current sequence of indices from $E[]$ that form the lines of the cut.

Lemma 2. All intersections that are to the left of all segments emanating from vertex v must be processed before vertex v is processed

Lemma 3. There always exists a ready intersection, unless the sweep line is aligned with the moving wall or unless we are considering the rightmost cut.

Proof. An intersection is ready if all edges that participate in it have active segments that are consecutive in the cut, and if it is outside the moving wall associated with the next vertex to be swept, v_{next}. Assume that some position of the sweep line has no consecutive segments of the cut with a common right endpoint that is outside the moving wall and where all segments that share the endpoint are active (otherwise we are done). We show that the next event point, the next vertex to be swept, v_{next} is a legal move. Assume that v_{next} is not a legal move. Then there is an in-edge e of v_{next} whose active segment does not have v_{next} as its right endpoint. Since all vertices to the left of v_{next} have been swept and since the graph is complete, e must be cut by the sweep line and contain some active segment s_e. Let the right endpoint v_e of s_e be an intersection with edge f and let f's active segment be s_f. Let v_f be s_f's right endpoint. Since v_e was not swept yet, either $v_f = v_e$ or $v_f < v_e$. Let s_e be the segment with the leftmost right endpoint. Such an endpoint exists, because the cut is not the rightmost, thus $v_f = v_e$. By the *connected component property* either the intersection v_e is ready or there are segments associated with it that are not yet active: a contradiction, since s_e is the leftmost right endpoint. □

Lemma 4. The total cost of comparing adjacent active segments (computing ready intersection points) through all steps is $O(k) = O(N^4)$.

Proof. Each segment c is first matched with the segments above and below. If a matching segment m is found, the $MATCH(m)$ will also be tested. By the *connected component property*, at most one of the edges that delimits c above or below can have a MATCH, or more than one connected component would exist. Thus, at most 3 tests will be generated. No additional tests are needed since if another match exists, it would have been found, when the segments that form it were investigated. The total number of segments that enter all the intersections is $O(N^4)$ and therefore the total complexity is $O(N^4)$. See also [13]. □

Lemma 5. The total cost of updating HFU (HFL) through all steps is $O(k) = O(N^4)$.

Proof. After processing the intersection point of segments $s_i, \ldots s_{i+k}$, for each segment s of the intersection apart from the first one, the HFU is updated by traversing the *bay* formed by the segments above s, starting from the segment immediately above s until we encounter the segment that intersects s (see Section 3.3). Consider the following charging scheme (see also [8]): for each segment traversed charge a unit cost to the intersection x corresponding to the elementary step; if somewhere later an elementary step at intersection z makes the segment that charges x invisible from x, transfer the unit charge to intersection z. At the end of the algorithm each intersection is charged at most once. □

Lemma 6. A topological sweep of a complete graph on N vertices and $k = O(N^4)$ intersections can be carried out in $O(k) = O(N^4)$ time and $O(N^2)$ space.

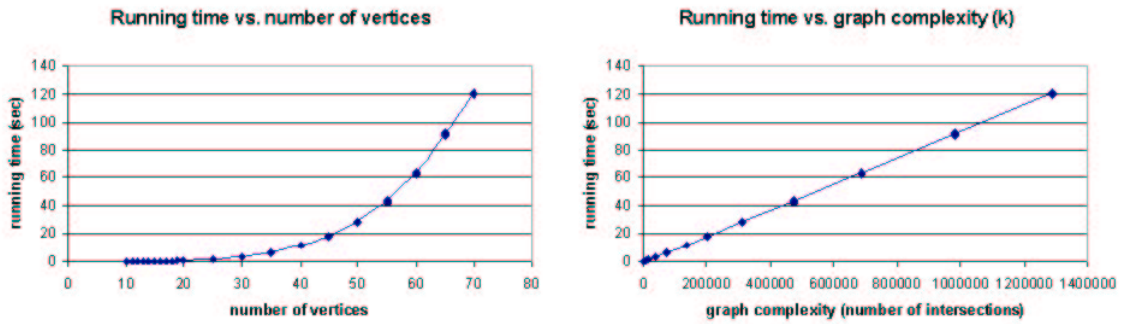


Figure 3: Experimental Results

Proof. The initialization step costs $O(N \log N)$: Sort all vertices and compute the moving wall for each vertex. The cost of each update step for an intersection point is an amortized constant, and there are at most $O(N^4)$ such intersection points. The cost of processing each vertex is as follows: When a vertex v is inserted, the segments terminating (resp. beginning) in v are deleted (resp. inserted) to the set of active segments in $O(N)$ time. Next, the horizon forests are updated and the cut is modified in constant time per affected segment. Lastly, the moving wall is updated. A scan is done through all active segments to check whether ready intersections exist inside the old moving wall but outside the new moving wall. In total, the cost of each update step for a vertex is linear in the size of the cut at the time the vertex is inserted and is therefore $O(N^2)$. There are N vertices so the complexity for vertex events is $O(N^3)$ and in total $O(N^4)$. □

5 Experimental Results

Our experiments checked the behavior of the code on graphs of sizes 10 to 70 vertices (hundreds to a over a million edges and intersections), created by generating N random points. When a vertex is swept all the active segments that terminate in it should be ready. In addition, at the end of a correct sweep the set of active segments must be empty and all the vertices must be swept. If a mistake occurs during the sweeping process one of the conditions above will not hold under the new topology. These conditions are checked to verify that the sweep advances correctly and that the algorithm works well.

Our code is written in C++, does not use any geometric libraries for computations, but uses GEOMVIEW for visualization of the output. If display is not needed the computation can be streamlined further. The code is built modularly and can be easily modified. It is located at: `http://www.cs.tufts.edu/research/geometry/graphsweep`. It was tested on a Sun Enterprise 250 processor, 400 MHz, and was compiled using the GNU C++ compiler. The results are presented in Fig. 3. It can be clearly seen that the running time is linear in the complexity of the graph k.

6 Future Research

We are currently working on modifying the topological sweep to work for **any** graph embedding. For dense graphs, with a large number of intersections, the implementation will achieve

the same time complexity as other efficient algorithms, since the number of intersections k will be the dominant factor. Another factor affecting the complexity will be the size of the cut, since the sweep line will be aligned with a moving wall N times, each time requiring additional tests, with a cost that is linear in the number of active segments. For sparse graphs, this factor can lower the attractiveness of the algorithm, since the maximal size of a cut can be large compared to the total number of intersections. However, for some graphs, the proposed algorithm might be an attractive alternative, where the total size of the cuts, along the moving wall, is limited.

Acknowledgment The authors wish to thank Prof. Ileana Streinu, Michael A. Burr and Ryan Coleman.

References

[1] G. Aloupis, S. Langerman, M. Soss, and G. Toussaint. Algorithms for bivariate medians and a fermat-torricelli problem for lines. *Comp. Geom. Theory and Appl.*, 26(1):69–79, 2003.

[2] T. Asano, L.J. Guibas, and T. Tokuyama. Walking on an arrangement topologically. *Internat. J. Comput. Geom. Appl.*, 4:123–151, 1994.

[3] I. J. Balaban. An optimal algorithm for finding segment intersections. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 211–219, 1995.

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

[5] K. Q. Brown. Comments on "Algorithms for reporting and counting geometric intersections". *IEEE Trans. Comput.*, C-30:147–148, 1981.

[6] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.

[7] D.Z. Chen, S. Luan, and J. Xu. Topological peeling and implementation. In *Algorithms and computation*, volume 2223 of *Lecture Notes in Comput. Sci.*, pages 454–466. Springer, Berlin, 2001.

[8] H. Edelsbrunner and Leonidas J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38:165–194, 1989. Corrigendum in 42 (1991), 249–251.

[9] H. Edelsbrunner and D. L. Souvaine. Computing median-of-squares regression lines and guided topological sweep. *J. Amer. Statist. Assoc.*, 85:115–119, 1990.

[10] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series F*, pages 307–325. Springer-Verlag, Berlin, West Germany, 1988.

[11] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM*, 25:739–747, 1982.

[12] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Comm. ACM*, 22(7):402–405, 1979.

[13] E. Rafalin, D. Souvaine, and I. Streinu. Topological sweep in degenerate cases. In *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, volume 2409 of *Lecture Notes in Comput. Sci.*, pages 577–588, Berlin, 2002. Springer-Verlag.

7 Appendix - Detailed algorithm

Initialization:

- Sort the vertices and store them in $V[]$.
- Compute the moving wall for each vertex by using the incremental convex hull algorithm.

Sweep

- While there are more vertices to be swept:
  1. Process the next vertex to be swept v_{next}.
  2. Delete all edges terminating in v_{next} from the list of active edges E and insert all edges starting at v_{next} and compute their horizon forests endpoints.
  3. If needed, update the horizon forest edges for all active segments affected by the newly added lines, by scanning through the list of active segments and comparing the right endpoint of the horizon forest to its intersection point with the newly added segment.
  4. Update the cut, according to the changes in the horizon forests.
  5. Update the moving wall and rescan the list of active segments E to detect if there are any ready intersections after the updates to the horizon forests and the cut or intersections that were inside the old moving wall and are now outside the new moving wall.
  6. While the stack of ready intersections I is not empty:
     - Pop the next ready intersection from the stack I
     - Update the order of the affected edges (that participate in the intersection) in the active list E, their horizon forest edges and their cut edges, as described above.
     - Compute ready intersections, using the cut. If a ready intersection is inside the moving wall, discard it (since it is not ready yet).