The Visibility Graph Among Polygonal Obstacles: a Comparison of Algorithms

by John Kitzinger
bejmk@yahoo.com
B. S., Computer Engineering,
University of New Mexico,1993

M. S., Computer Science,
University of New Mexico, 2003

ABSTRACT

This paper examines differences of four approaches in finding the visibility graph of a polygonal region with obstacles defined by simple polygons. Each has been implemented and tuned. Experimental comparisons via time measurements have been carried out against a variety of testcases ranging in graph density from maximal, $O(n^2)$, to minimal, $\Theta(n)$. In this manner, expected asymptotic time bounds have been verified with crossover points between the algorithms identified.

Table of Contents

List of Figures..... vi
List of Tables..... vii
Introduction..... 1
Input/Output Assumptions..... 4
Naive Algorithm..... 6
    Overview..... 6
    Implementation Details..... 6
Lee's Algorithm..... 7
    Overview..... 7
    Implementation Details..... 8
Overmars and Welzl's Algorithm..... 10
    Overview..... 10
    Implementation Details..... 12
Ghosh and Mount's Algorithm..... 13
    Overview..... 13
    Implementation Details..... 19
Performance Comparisons..... 23
    Timing Methodology..... 23
    Timing Results..... 23
Other Aspects and Comparisons..... 38
Conclusion..... 41
Appendices..... 42
    Appendix A - Tuning the Naive Method..... 44
    Appendix B - Tuning Lee's Method..... 46
    Appendix C - Tuning Overmars and Welzl's Method..... 48
    Appendix D - Tuning Ghosh and Mount's Method..... 49
    Appendix E - Tables of Measurements Used in the Plots..... 51
    Appendix F - Rotation Tree Pseudocode..... 56
References..... 59

List of Figures

Figure 1 - Examples of Collinear Points....4
Figure 2 - Example of Lee Scan with Edge-List.... 7
Figure 3 - Basic Cases in the Lee Scan....7
Figure 4 - Looping the Rotation Tree (One Iteration Re-Attaching to Grandparent)....11
Figure 5 - Looping the Rotation Tree (One Iteration Re-Attaching to Chain Above).... 11
Figure 6 - Chains in the Mehlhorn Triangulation.... 13
Figure 7 - Various Cases of the Mehlhorn Triangulation.... 14
Figure 8 - A Vertex As Apex of More than one Funnel....15
Figure 9 - An Example of the Lower and Upper Tree for an Edge (x,y) and the Induced Funnel Sequence (Lower Parent in Parens)....15
Figure 10 - The Relevant Vertices of a Split when Adding a New Vertex.... 16
Figure 11 - Pseudocode of the Split Procedure.... 17-18
Figure 12 - A Split Where *r* Is A Special Case....21
Figure 13 - Skipping Vertices in the *s* to *r* Walk.... 22
Figure 14 - Polygonal Region with $O(n^2)$ Visibility Edges, e.g. n=25.... 24
Figure 15 - The Entire Visibility Graph of the Example in Figure 14.... 24
Figure 16 - Plot of Execution Times for the Circle of Obstacles Set.... 25
Figure 17 - n-gon with $O(n^2)$ Visibility Edges, e.g. n=10.... 26
Figure 18 - Plot of Execution Times for the n-gon Set.... 27
Figure 19 - Polygonal Region with $O(n^{3/2})$ Visibility Edges, e.g. n=68....29
Figure 20 - Plot of Execution Times for Square Grid of Obstacles Set.... 30
Figure 21 - Polygonal Region with $O(n)$ Visibility Edges, e.g. n=22.... 31
Figure 22 - Plot of Execution Times for the Line of Triangles Set.... 32
Figure 23 - Spiral with $O(n)$ Visibility Edges, e.g. n=86.... 33
Figure 24 - Plot of Execution Times for the Spirals Set.... 34
Figure 25 - Random Region, e.g. n=154.... 35
Figure 26 - Plot of Execution Times for the Random Set.... 36

List of Tables

Table 1 - Measurements of the *n*-gon Testcase with *n*=5000..... 28
Table 2 - Measurements of the Square Grid of Obstacles Testcase with *n*=7748..... 29
Table 3 - Measurements of the Line of Triangles Testcase with *n*=10,000..... 33
Table 4 - Measurements of the Random Testcase with *n*=5000..... 37
Table A-1 - "linear154" Measurements for Naive Versions (seconds)..... 44
Table A-2 - "box148" Measurements for Naive Versions (seconds)..... 45
Table A-3 - "quad58" Measurements for Naive Versions (seconds)..... 45
Table B-1 - "linear154" Measurements for Lee Versions (seconds)..... 46
Table B-2 - "box148" Measurements for Lee Versions (seconds)..... 47
Table B-3 - "quad58" Measurements for Lee Versions (seconds)..... 47
Table B-4 - Lee Measured Improvements from Version to Version..... 47
Table C-1 - "linear154" Measurements for Overmars/Welzl Versions (seconds)..... 48
Table C-2 - "box148" Measurements for Overmars/Welzl Versions (seconds)..... 48
Table C-3 - "quad58" Measurements for Overmars/Welzl Versions (seconds)..... 48
Table D-1 - "linear154" Measurements for Ghosh/Mount Versions (seconds)..... 49
Table D-2 - "box148" Measurements for Ghosh/Mount Versions (seconds)..... 50
Table D-3 - "quad58" Measurements for Ghosh/Mount Versions (seconds)..... 50
Table E-1 - Average Measures for the Circle of Obstacles Testcases (milliseconds)..... 51
Table E-2 - Average Measures for the *n*-gon Testcases (milliseconds)..... 52
Table E-3 - Average Measures for the Square Grid of Obstacles Testcases (milliseconds)..... 52
Table E-4 - Average Measures for the Line of Triangles Testcases (milliseconds)..... 53
Table E-5 - Average Measures for the Spiral Testcases (milliseconds)..... 54
Table E-6 - Average Measures for the Random Testcases (milliseconds)..... 55

Introduction

In computational geometry problems, visibility has been an important property to find. Of course, computational visibility problems vary in form. Among the two dimensional variety, sometimes the problem is restricted to visibility in a simple polygon (no obstacles) [Her87] [GHLST87] [KM00]. More generally, there can be obstacles, sometimes called holes or islands. The obstacles can be restricted to special shapes, such as rectilinear, circular, line segments, or convex polygons; or they can be more general, such as simple polygons[1]. The problem may be in finding the visibility of just one vertex or finding the visibility of all vertices. A visibility graph, G_s=(V,E), may be the structure that contains the visibility information. The edges of the visibility graph are represented between any two vertices if there are no edges obstructing the visibility between them. Other structures include the vertex-edge visibility graph [GM91] [OS97a] [OS97b], the visibility polygon (or view from a point) [EGA81] [EOW83] [AGHI86] [Poc90] [Veg90] [HM91] [PV93] [OS97a] [Riv97], the visibility diagram [Veg90] [Veg91], the visibility complex [PV93] [PV95] [Riv95] [Riv97], the parallel view [EOW83] [AGHI86] [GM91], and neon visibility [Poc90] [Veg91] [PV93]. The vertex-edge visibility graph is touched upon in the "Other Aspects and Comparisons" section, but none of these other ideas are explored in this paper.

This paper focuses on finding the entire visibility graph among polygonal obstacles. The obstacles are only restricted to being simple, i.e. no edge can intersect any other edge. The visibility graph problem itself has long been studied and has been applied to a variety of areas. A common use for it has been for finding the shortest path[2]. Exploiting the fact that the shortest path consists of arcs of the visibility graph, one can find the shortest path by running Dijkstra's algorithm [Dij59] on it. The shortest path has been used in robot motion planning. This was identified in 1979 in Lozano-Perez and Wesley's work [LPW79]. The visibility graph can also be used to solve the art gallery problem by finding the minimum dominating set of the visibility graph (NP-hard). More recently, visibility has been used in pursuer-evader problems, e.g. in [LSC99]. Finally, the visibility complex, which contains more information than the visibility graph, has been used in illumination problems [ODRP96].

D.T. Lee in his 1978 Ph.D. dissertation [Lee78] wrote about the first nontrivial solution to the visibility problem running in $O(n^2 \log n)$ time. In the mid-to-late1980's a series of

[1] The term "simple" is a misnomer because it actually permits concavity to any degree - the only restriction is that the edges form a closed chain with no intersections.

[2] Some work has been done for finding the partial visibility graph where only the tangents around obstacles are included since the shortest path would not need other visibility edges to the obstacle [KM88] [PV95].

$O(n^2)$ papers appeared. In 1985, E. Welzl described a technique [Wel85] based on an arrangement of the dual of the vertices [CGL83] [EOS83] followed by a topological sort to order the vertex pairs in $O(n^2)$ time. This technique is used in other computational geometry problems as well. Welzl's technique requires $O(n^2)$ working space. It works for a set of line segments and can be adapted for sets of polygons. Edelsbrunner and Guibas [EG86] later improved the working storage of the topological sweep to $O(n)$. About the same time, Asano, et. al. offered two other versions with arrangements also requiring $O(n^2)$ space: the first via triangulation and the second via scanlines and segment splitting. These techniques construct the polar order one vertex at a time as opposed to Welzl's technique that produces a good permutation (not strictly sorted, but good enough) among all vertex pairs at once. Asano's technique can also handle dynamic updates in $O(n)$ time. In 1988, a paper by Overmars and Welzl describes yet another $O(n^2)$ technique that does not need to calculate the dual arrangement and uses only $O(n)$ working space. The paper also describes a second algorithm running in $O(|e| \log n)$ time and $O(n)$ space, where $|e|$ is the number of edges in the visibility graph. It is efficient for sparse graphs, i.e. when $|e| = O(n)$. Of course, all of the $O(n^2)$ algorithms are optimal when the graph is dense, in other words when $|e| = O(n^2)$.

Towards the end of the appearance of the $O(n^2)$ papers, two output-sensitive approaches became known [GM87] [GM91] [KM88] [KM00]. Ghosh and Mount show a planar-scan technique using triangulation and funnel splits to achieve $O(|e| + n \log n)$ time bounds. The other work by Kapoor and Maheshwari essentially achieve the same time bounds using corridors which are based on a triangulation dual. With both of these, the time is basically bounded by the number of edges in the visibility graph, but it can be dominated by the time it takes to triangulate (which optimally is $O(n \log n)$ for a polygon with simple polygonal holes). Thus, both of these approaches are optimal for graphs with density as low as $|e| = O(n \log n)$ but no lower.

What is described here is the performance achievable with several of these algorithms. For comparison, a naive (i.e. trivial) approach is included which runs in $O(n^3)$ time. After this, Lee's algorithm is described which runs in $O(n^2 \log n)$ time, followed by Overmars and Welzl's method (being one of the more elegant $O(n^2)$ algorithms). Finally, the Ghosh and Mount approach represents the last algorithm described which has a theoretical running time of $O(|e| + n \log n)$. Certain parts of the implementations required some adaptation. First, in the Overmars and Welzl method, the algorithm had to be adapted from strictly non-intersecting line segments to polygons. Second, in the Ghosh and Mount paper, problems were identified and corrected.

Input/Output Assumptions

Input

Across all implementations, the same input file format is used. To specify a polygonal region, the user specifies the outer polygon plus *m* polygonal obstacles. The polygons can be simple where the vertices appear consecutively, counter-clockwise for the outer polygon and clockwise for the obstacles. The general rule is that the inside of the polygon (free space) lies to the left of an edge, where direction of the edge is defined going from the lower-numbered vertex to the higher-numbered vertex. Of course the segments of a polygon may not intersect itself, as per the definition of *simple*, and the usual assumption that the obstacles are disjoint from each other is made.

Two other important input assumptions often found with many computational geometry problems are *general position* and *non-collinearity*. These assumptions might be made in order to make a proof easy to follow. However, the implementations described in this paper were coded for practical use, so alleviating these assumptions allows more realistic input.[3]

General position *General position* means the restriction of making all vertices have unique x-coordinates, i.e. not allowing the vertices to lie along the same vertical line. This assumption is *not* made in any of the implementations.

Non-collinearity By definition, non-collinearity disallows any three points from lying on the same line. This restriction has been partially alleviated (see Figure 1). As long as the points are not adjacent along the polygonal boundary, then collinear points are allowed. Only points along adjacent edges may not be collinear. However, even in this case, the restriction is not too severe because the input could be preconditioned in $O(n)$ time to remove instances of this kind of collinearity.

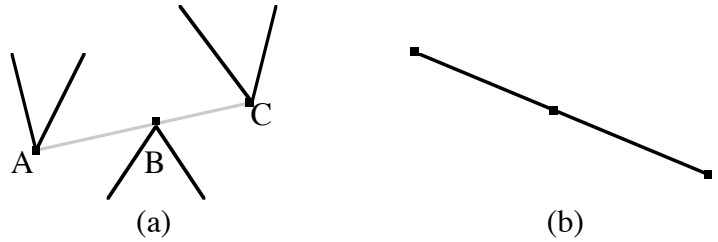


Figure 1 - Examples of Collinear Points
(a) allowed (b) disallowed

[3] Of course, when this kind of assumption is relaxed, it should have no impact on overall time or space bounds. The tradeoff to this is that it may increase code complexity and constant factors of time.

In all the implementations, collinearity does not imply visibility, i.e. collinear points are not transitive with respect to visibility. For example, in Figure 1(a), vertices A, B, and C are collinear to each other, with A being visible to B, and B being visible to C, but A is not considered visible to C. This interpretation actually will produce less edges in the visibility graph as compared to the interpretation of transitivity (making A visible to C). On the downside, this affects shortest path calculations by adding hops; for example, the shortest path from A to C is {A,B,C}, not {A,C}. However, the overall distance will turn out to be the same as compared to visibility among collinear points having a transitive meaning.

Another issue about collinearity is that sometimes it is difficult to exactly specify the coordinates of points that are supposed to be collinear to each other. To help with this, all of the implementations described in this paper allow the user to specify a tolerance (given by some small epsilon) in which the angles or slopes may deviate to still be considered collinear. This also allows slope and angle calculations within the implementations to lose some precision (which is inevitable with IEEE floating point representation).

Output

All of the implementations calculate the *inner* visibility graph. The inner visibility graph only includes those visibility segments that are inside the polygon (and outside the obstacles). The segments that would occur on the outside (and on the inside of obstacles) are not included. In the section called "Other Aspects and Comparisons" section, there is more discussion of the outer visibility graph.

The expected output of the visibility graph can be in standard output text. For example,

```
visible segment from 0 to 1
visible segment from 1 to 0
```

shows that visibility exists between vertex #0 and #1.

Naive Algorithm

Overview

A simple solution to the problem would be to just look at every edge to see if it blocks/interferes with a given pair of vertices. If none interfere, then the two vertices are visible to each other (otherwise not). Of course, to produce the entire visibility graph, the procedure loops through every pair of vertices. The time analysis is simple also: there are (n choose 2) pairs of vertices which is $O(n^2)$ and there are $O(n)$ edges (one for every vertex) so this means the total time is $O(n^3)$. As for storage, the algorithm requires $O(n)$ working space (at least to store the input), and if the visibility graph is stored - not just reported - then it requires $O(|e|)$ memory.

Implementation Details

The implementation is straight forward. Some tricks are obvious and are detailed in Appendix A along with what timing difference they make. Two methods were identified for distance calculation: law-of-sine method and intersection method. It was found that the intersection method is slightly faster. Also, this method can be modified (as was done for the timed version) to calculate squared distances which saves on a square root operation. However, the intersection method deals with slopes, so verticals have to be handled with care since slopes can be `+Inf` and `-Inf`. Collinear points (three or more points on a line) gave some difficulty also.

As with all the algorithms, during development an openGL version was used for visualization and debugging. Of course, the graphics part was removed to make a text-only version for performance measurements (timings).

Lee's Algorithm

Overview

The algorithm attributed to D. T. Lee [Lee78] represents the first nontrivial solution running in $O(n^2 \log n)$ time. The basic idea is simple: for each vertex, sort the other points in angular order around it, then visit each one keeping track of the order of intersected edges made by the scan-line. If the visited point is associated with the first edge in this ordered list, then it can be reported. Otherwise, it must be obscured by some other edge appearing before it (with respect to the center) and so would not be reported. Of course, the edge list must handle inserts and deletes in $O(\log n)$ time which means using optimal sorting (of which many are available).

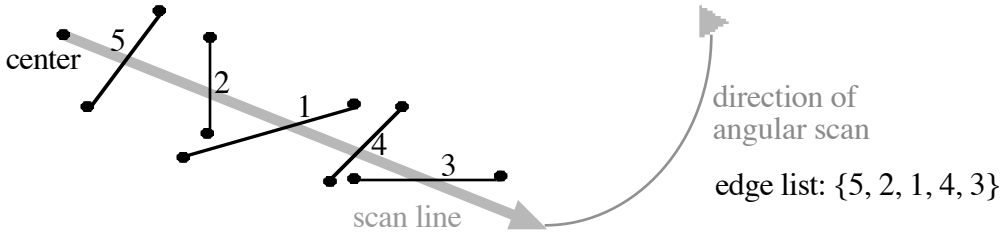


Figure 2 - Example of Lee Scan with Edge-List

Figure 2 shows the intuitive idea. The edge-list here would be {5, 2, 1, 4, 3} - the order of intersecting edges from the center along the scan-line. Of course, in reality, the scanline only stops at vertices (not in the middle of edges).

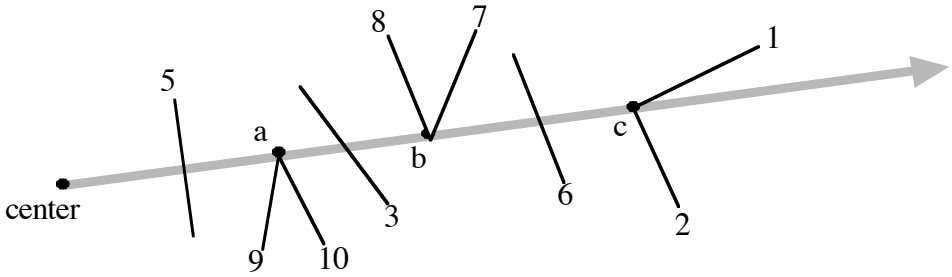


Figure 3 - Basic Cases in the Lee Scan

What happens at each vertex visit depends on the polygonal edges associated with that vertex. There may be two inserts, two deletes, or an insert and a delete. Figure 3 shows these situations with vertices marked *a*, *b*, and *c* with edges marked 1-10. Collinear points are handled as follows: if several points lie along the same scan-line, the order is

determined by the distance from the center. In Figure 3, vertex *a* would be visited, followed by vertex *b*, followed by vertex *c*. Before *a* is visited, the edge-list would be {5, 9, 10, 3, 6, 2}. When *a* is handled, both its edges are deleted, so the edge-list afterwards would be {5, 3, 6, 2}. When *b* is handled, both its edges are inserted, so the edge-list becomes {5, 3, 8, 7, 6, 2}. When *c* is handled, one edge is deleted and the other is inserted. The edge-list at the end would be {5, 3, 8, 7, 6, 1}.

Time analysis: there are (n-1) vertices to be visited for each of n^2 centers. At each of the (n-1), it takes $O(\log n)$ for the search/insert/delete, thus making the time for one scan (n-1)*$O(\log n)$ = $O(n \log n)$. The time for all n scans would then be $n*O(n \log n)$ = $O(n^2 \log n)$.

Space analysis: in the worst case, there may be $O(n)$ edges in the edge-list at any one given time, but no more. The angularly sorted list also requires $O(n)$ storage during one scan, but can be freed after the particular center has completed. Of course, in order to store the visibility graph, it takes $O(|e|)$ space.

Implementation Details

This implementation uses an AVL tree for both the angular-sorted list and the edge-list. Many optimizations can be found which are detailed in Appendix B with the corresponding timings of the performance increases observed. One trick was identified for vertices which require one edge insertion and one edge deletion. Instead of doing these two operations, a single replace can be used keeping the order of the edge-list intact. Also, the scan itself works when going only halfway around since visibility between a pair is mutual. Lastly, distance calculations can be optimized as in the Naive method.

One hard part of implementing the Lee method is initialization of the edge-list. If the scan starts at -//2, all n^2 edges of the input have to be checked for inclusion into it. When one or both of the endpoints are collinear to -//2, it may or may not appear in the initial edge list depending on the direction of the edge. Some other the difficulties arose with collinear points. For instance, collinear edges never appear in the edge-list (at any angle) and this kind of edge's furthest endpoint (and maybe its closest endpoint also) does not get reported as visible (because of the chosen semantics of collinear points).

Another difficulty was handling two adjacent edges that both have to be inserted at the same

time. The basic distance calculation would return values that are *almost* equal (but not quite equal due to precision). In this case, the angle made by the edge to the scan line must be used to determine the order of the two edges with respect to each other.

Overmars and Welzl's Algorithm

Overview

Welzl originally published a paper [Wel85] describing a technique based on a topological sort of the dual arrangement of segments in a plane. Because it effectively sorts all (n choose 2) pairs of vertices, it runs in $O(n^2)$ time (as opposed to Lee's $O(n^2 \log n)$ algorithm which scans one vertex at a time with a sort of all other points at each step). The space required is $O(n^2)$. This was later improved by Edelsbrunner and Guibas [EG86] to O(n) space. Asano, et. al. [AGHI86] has one version of this based on triangulation and set-union and another based on scanlines and splitting. The Overmars and Welzl paper [OW88] represents a practical version without using dualization. Instead, it is based on the concept of *rotation trees*.

The idea is simple: for each vertex, a scanline is kept which runs from -∞ to +∞ hopping from vertex to vertex in its path. During the main loop, it appears that all of the scanlines are proceeding simultaneously. In fact, there are exact rules about determining the next vertex to process, and some vertices may finish their scan before others.

To understand the rules about finding the next vertex, the *rotation tree* must be understood. A rotation tree is a rooted planar tree where each vertex is a node and points to its parent. There are two special nodes: `+Inf` and `-Inf`, where `-Inf` is infinitely below and just to the right of all regular points, and `+Inf` is infinitely above and just to the right of all regular points. Initially, all vertices point to `-Inf` as their parent and `-Inf` points to `+Inf`. Also stored is the rightmost child (if a node is a parent), and its right and left siblings (if they exist). The ordering of children is by slope: the one with the smallest slope is the leftmost.

The loop that examines all pairs simply takes the rightmost leftmost leaf as the next segment to process and then reattaches it to the tree (while maintaining the property of being a rotation tree). It can reattach to the left of its parent or to the tangent of the chain above it. Figure 4 and 5 show examples of the next segment processed and where it reattaches (the thick line). When a vertex attaches to `+Inf`, it is finished. The loop continues when all points have attached to `+Inf`.

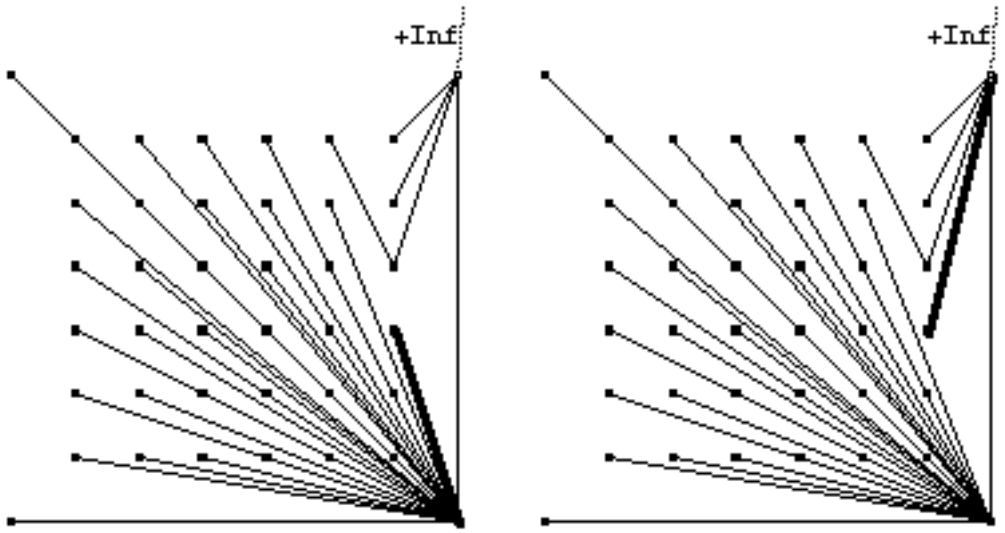


Figure 4 - Looping the Rotation Tree (One Iteration Re-Attaching to Grandparent)

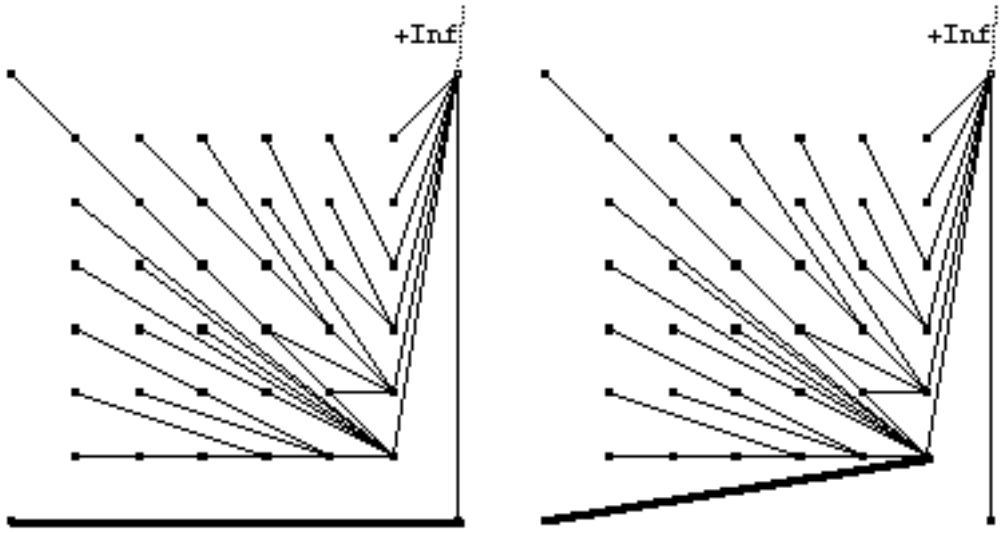


Figure 5 - Looping the Rotation Tree (One Iteration Re-Attaching to Chain Above)

The details of the rotation tree can be found in Appendix F. It has been used for other problems as well, but first appeared for the purposes of determining the visibility graph. Some extra information and processing of course is needed for this. For each vertex, the nearest visible segment is kept. If a point is associated with the nearest visible segment, then visibility between the two points is known. When the point is not associated with this nearest segment, there is no visibility. The nearest visible segment changes only when the scan passes it. In this case, the new nearest visible segment becomes that of the endpoint of the segment just passed. In this way, no edge-list must be maintained as in Lee's algorithm.

Implementation Details

Implementation was straightforward with the pseudocode telling almost every detail. First, all the regular vertices point to `-Inf` in the right order, i.e. the leftmost child is the vertex with the greatest x-coordinate. This requires a reverse-x-sort and was implemented with an AVL tree.

Another implementation decision concerns keeping track of the leftmost leaves. Actually, the paper suggests a stack and it was implemented as an array. So, all leftmost leaves are kept in this stack where the rightmost is kept at the top. This eliminates the need for a priority queue and thus keeps the running time at $O(n^2)$.

A significant part of implementation lies in the fact that the algorithm was designed for visibility among nonintersecting line segments, not polygons. So some adaptation was required. With polygons, it gets a little more complicated because there are two edges associated with each vertex. For instance, the nearest visibility edge can switch at a point to the adjacent edge (instead of inheriting the point's nearest visible edge).

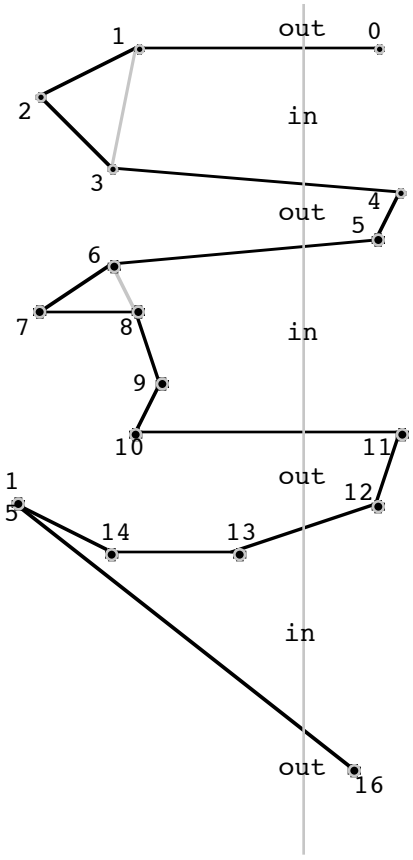
Finally, handling collinear vertices proved to be somewhat troublesome. Here, because of a user-specified epsilon which defines the angular range of collinear vertices, the scanline for a particular vertex may reach any of the vertices among a set of collinears in any order. The solution is to look ahead and delay processing until the scanline has finally gone beyond the set of collinears, keeping track of the nearest one. Only the nearest one has a chance of being visible and only it can be used for the inherited nearest visible edge.

Ghosh and Mount's Algorithm

Overview

The approach detailed by Ghosh and Mount [GM87] [GM91] is basically a planar scan left to right proceeding by a variant of the Mehlhorn triangulation [Meh84c]. It runs in $O(|e| + n\log n)$ time. The $n\log n$ factor represents the time for the triangulation (both at the start for sorting the vertices according the their x-coordinate and a constant number of AVL insert/delete/finds at each point in the scan). The $|e|$ factor is the size of the visibility graph.

The convex chains of triangulation edges form a scan boundary by which new vertices being incorporated into the partial visibility graph can attach - see Figures 6 and 7. This then alters the shape of the scan boundary. The algorithm also maintains a funnel data structure for each such edge and another data structure to hold the visible segments (as the algorithm proceeds in the scan) for each vertex.




```
chain: [ edge(0,1), 1, 3, edge(3,4) ]
rightmost = 1
```

```
chain: [ edge(5,6), 6, 8, 9, 10,
         edge(10,11) ]
rightmost = 9
```

```
chain: [ edge(12,13), 13, 14, 15,
         edge(15,16) ]
rightmost = 13
```

Figure 6 - Chains in the Mehlhorn Triangulation

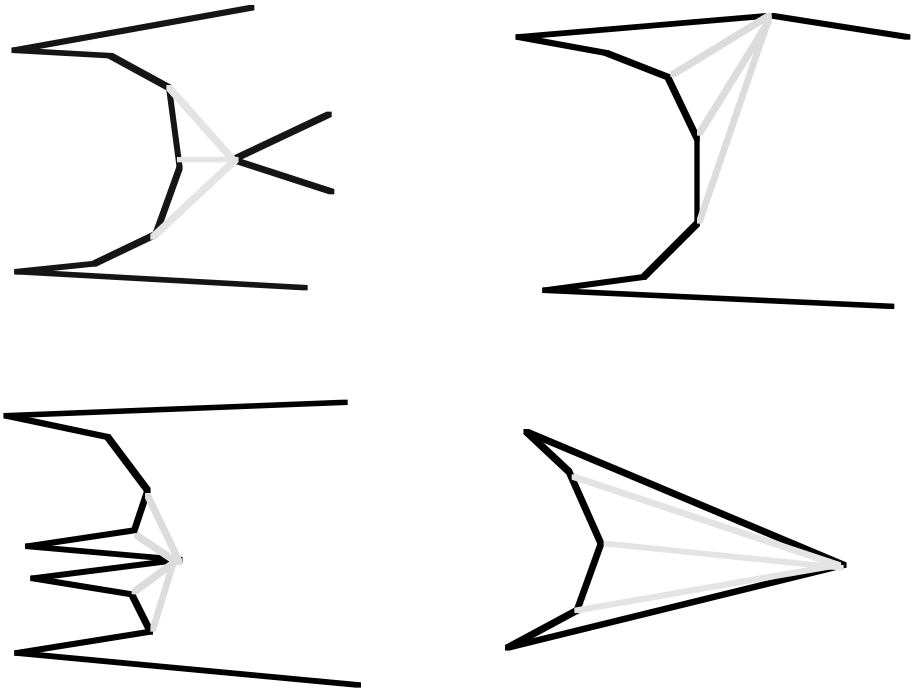


Figure 7 - Various Cases of the Mehlhorn Triangulation

Some description of the funnel data structure follows here. For each triangulation edge on the scan boundary, there are vertices to the left which can see it. Each visible vertex is an apex of something called a funnel (also used in other computational geometry problems), where a funnel is defined uniquely by its apex and its lower parent. An apex has two parents, lower and upper, where a parent is simply the first vertex in the convex chain toward one of the vertices of the triangulation edge. One property of funnels is that they are empty. Another property of funnels is that the apex sees the edge in the range of the intersection of the tangents between the apex and its parents. A vertex visible to the edge can be part of more than one funnel since there can be more than one path to the edge vertices. See Figures 8 and 9.

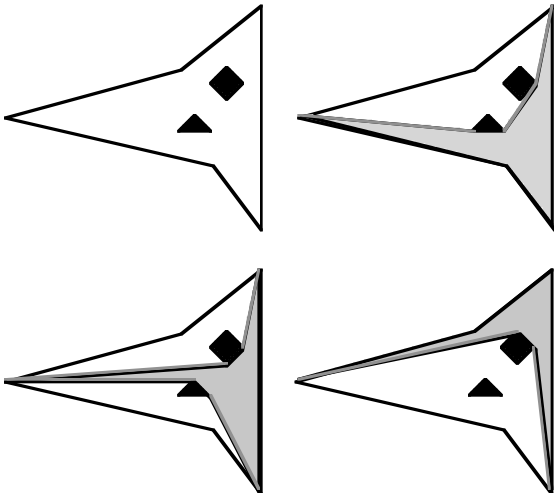


Figure 8 - A Vertex As Apex of More than one Funnel

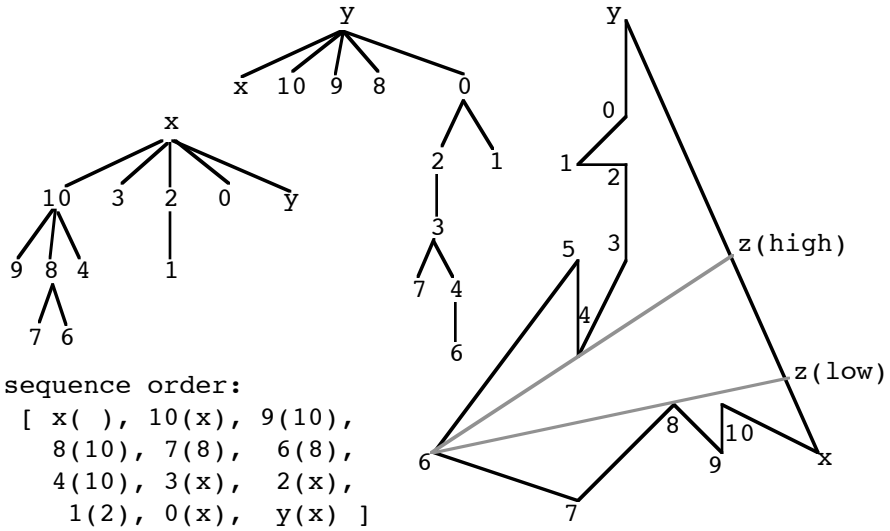


Figure 9 - An Example of the Lower and Upper Tree for an Edge (x,y) and the Induced Funnel Sequence (Lower Parent in Parens)

A hierarchical clockwise ordering of the funnels form a funnel sequence which can be maintained by a doubly-linked list of funnels (id of apex + id of lower parent). The ordering can be found by means of a clockwise preorder traversal of the lower tree or a clockwise postorder traversal of the upper tree (Figure 9). It turns out these are equivalent. However, in implementation, no tree is stored. Instead, a doubly-linked-list maintains this funnel sequence order.

At each new vertex, two new triangulation edges are formed from one already on the

scan boundary. With a procedure called *split*, the funnel sequences of the new edges can be determined from the funnel sequence of the one being split. All of the funnels of the old edge will be visible to one new edge or both (in the case of both it is necessarily visible to the new vertex).

In order to meet the time bounds ($|e|$), the algorithm only visits apexes that are visible (to the new vertex). But it also has to keep track of vertices that are hidden to the new vertex but which can see one of the edges because subsequent splits for vertices after this vertex (to the right) might have visibility. So, from one visible vertex the algorithm must jump to the next one in constant time also appending these hidden pockets. It does this using clever convex chain walks which exploit the emptiness of funnels, e.g. finding the upper and lower parents quickly. For this, the following constant time operations have to be defined: clockwise segment (CW), counter-clockwise segment (CCW), clockwise extension (CX), counter-clockwise extension (CCX), and reverse (REV). CW and CCW are simply the next and previous visible segments. CX(a, b) is the very next vertex visible to b around the corner (turning right) from a to b. CCX(a, b) is the same thing but turning left[4]. The reverse is simply the same visibility segment but going the opposite way, e.g. REV(a, b) = (b, a).

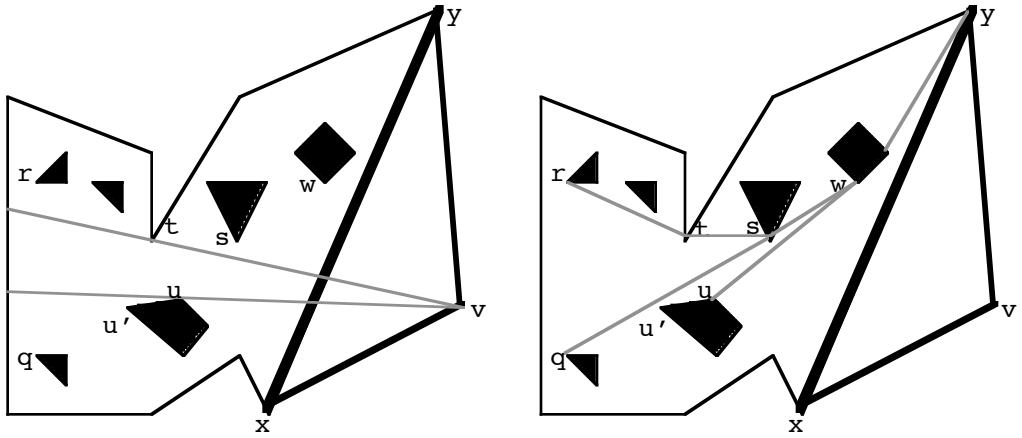


Figure 10 - The Relevant Vertices of a Split when Adding a New Vertex
(a) The Emptiness from a Visible Vertex *u* to the Next One *t*
(b) The Upper Chains

Figure 10 depicts a simple situation during a split. Figure 11 shows pseudocode of the split. The edge split starts with vertices x and y which define the old triangulation edge. Unless it is a trivial edge, there will be points in between x and y

[4] The extensions may not be defined if the polygon edges get in the way.

which are visible. This is where the split procedure becomes recursive. If some visible vertex *u* is found, split will be called between it and its upper parent (parent along the convex chain to *y*). Since this can repeat, the recursion can have arbitrary depth. The work in the split procedure between a visible vertex, *u*, and its upper parent, *w*, has several steps. The object is to find *t*, the next visible vertex in clockwise order. Also, between *u* and *t* in the funnel sequence lie the hidden vertices (there may be none). However, the hidden vertices are visible to either the upper edge or the lower edge. Suppose the funnel sequence (not showing lower parents) has the following form: [..., *u*, ..., *q*, *r*, ..., *t*, ...]. Again, everything between *u* and *t* (non-inclusive) is hidden. The vertices after *u* up to a certain point define the hidden pocket that is only visible to the upper edge. The vertices before *t* up to a certain point define the hidden pocket that is only visible to the lower edge. The algorithm defines vertices *q* and *r* to be the limits of the pockets. Finding *r* actually requires a backward walk from another vertex, called *u'*, which is the CCX of the new vertex and *u*. It is also the child of *u* with respect to the upper edge. In the walk, when the child is no longer the extreme clockwise child, this other clockwise child will be *r*. *q* is just the funnel previous to *r*[5].

After finding *q* and *r*, the algorithm needs *t*. However, in order to do this, it must walk the upper chain from *w* back to *r*[6]. In the walk, there can be many visible vertices, all of which have not been visited. But in order to meet the time bounds, the walks must only occur once (or a constant number of times). This means all the visible vertices found from *w* to *r* must be stored and subsequently split (recursively) because otherwise the walk might occur $O(n)$ times. The newly found visible vertices are stored in a stack/queue. Another point here is that the walk cannot go from *r* to *w* because there could be $O(n)$ hidden points between *r* and *t*.

```
1. find u' - the CCX of (v, u).
2. find r and q by walk from u' along convex chain to x
      u'' = u
      while (u' is extreme CW child of u'' && u'' is not x)
            grandparent = CX of (u', u'')
            u' = u''
            u'' = grandparent
      r = CCW(u'', u')
```

[5] Note that the walk cannot go from *u* to *q* because there may be $O(n)$ vertices in this hidden pocket and this would break the time bounds. Walking from *u* back along the lower convex chain towards *x* only visits visible vertices and is only be carried out a constant number of times, thus preserving the time bounds.

[6] r can be t, t can be s, s can be w, w can be y.

```
      q = r->prev
      (special case if u' does not exist - q=u, r=q->next)
3. find s - the upper parent of q
      s = CCW(q, lower parent(q))
4. find next visible vertices
      k = 0
      for next = walk s to w - found by CCW
            k++
            enqueue(next)
      for next = walk s to r - found by CCX(q,s) then CX
      (can stop when vertex is not visible to newv)
            k++
            push(next)
      let t be the last visible vertex (top of stack/queue)
      note w is the on the bottom of the stack/queue
6. recurse on newly found visible vertices
      for i = 1 to k - the visible vertices from t to w
            split(pop(), pop())
```

Figure 11 - Pseudocode of the Split Procedure

To explain how CW, CCW, CX, CCX, and REV are implemented also explains how the visibility segments are stored. The visibility segments are actually stored in two phases - A and B. After incorporating a new vertex, the only known visibility segments lie to the left - this is known as phase A. These are found in clockwise order and can be stored in an array after the new vertex has finished its splitting(s). Vertices visible on the right-side are in the vertex's phase B. When phase B segments are found, they come in any order, so they have to be stored in a linked list. The two phases relate to each other in the CW, CCW, CX, CCX, and REV operations. For REV, if (*a*, *b*) is in *a*'s phase A, REV(*a*, *b*) will be in *b*'s phase B. As for CW and CCW, they usually stay in the same phase, but CW could go from phase A to phase B or from phase B to phase A, and similarly for CCW.

For the extensions (CX and CCX), the paper requires the use of phase A intervals (or groups) and the split-find data structure[7] [HU73] [GT85] [LaP90]. When no phase B segments have been found, only one phase A interval exists which holds all the phase A segments. When a phase B segment is found, its extension may split up an interval into two. The intervals are kept in a linked list. Each phase B segment points to an associated phase A interval and each interval has an associated phase B segment. Initially, there is an imaginary vertical representing the associated phase B segment for the one and only phase A interval.

[7] An interval split should not be confused with an edge split.

To find the CX of a phase A segment, it is first necessary to locate its interval. Then CX is found by taking the next interval's associated phase B segment. To find the CCX of a phase A segment, it is simply the interval's phase B segment. To find the CX of a phase B segment, one takes the first phase A segment of the associated interval. To find the CCX of a phase B segment, one takes the last phase A segment of the interval previous to the associated interval.[8]

When a phase B segment is added, it may split the next higher phase B segment's phase A interval. The first challenge is finding out where to make the interval split. The algorithm makes use of a dovetailed doubling search[9] which is a binary search preceded by finding the exponential bounds on the right side of the array. This is necessary to meet the time bounds. Specifically, if m_a is the number of phase A segments, the following recurrence holds:

$$T(m_a) = \max_{0 \le k < m_a} \left[T(k) + T(m_a - k) + \min(\log(k), \log(m_a - k)) \right]$$

It solves to $O(m_a)$. This means that the amortized time for interval splits is linear in the number of phase A segments. The second challenge is to make finds run in amortized $O(m_b)$ time, where m_b is the number of phase B segments. Together the split-finds associated with one vertex would run in $O(m_a + m_b)$ time which is O(|*e*|) over all vertices. It is assumed that the linear time union-find algorithm by Tarjan-Gabow [GT85] can be reversed for the rest of the split-find, i.e. regrouping.

Implementation Details

An AVL tree holds the scan order (left to right, followed by bottom to top when x-coordinates are equal), and another AVL tree holds the chains (y-structure) of the triangulation. Doubly-linked lists represent the chains of the scan boundary, the funnel sequences, the phase A intervals, and the phase B segments. The phase A segments for each vertex exist as an array.

It was difficult to generalize the split procedure to all geometric situations. Part of the difficulty of implementing the algorithm is that the loops (or walks) are deceptively simple as described in the paper. This is so because the walk can go from one phase to the other. For instance, in the walk from *u''* to x along this

[8] CX and CCX may wind up in the same phase if there is no previous or next interval or if the phase B segment is the imaginary vertical.

[9] This is equivalent to the finger tree method described in Ghosh and Mount's earlier paper [GM87].

convex chain, the (*u''*, *u'*) segment starts out in phase A of *u''*, but after one iteration, this may turn over to phase B. Also, the backward walk might end at *x* which is similar but requires extra code than if not.

Other difficulties are the special cases. For instance, *u'* may not be valid if not visible to (*v*, *y*). A check must be in place for this. It will be invalid if it is in phase B because it would not be in the funnel sequence if it is to the right of *u*. Of course, handling collinear vertices requires extra coding and checks.

A major difficulty when coding was relating the funnel sequence to the visibility segments. There was more than one occasion when it was necessary to have the position in the funnel sequence, but only the identity of the vertex was known. For this, re-coding added funnel position into the segment structure. Some of these positions then have to be maintained when splitting.

An instance of this is when finding *r* via the clockwise operation (CW). In fact, one particular situation was overlooked in the paper and had to be corrected. If the CW flips over into phase B, using the first phase B segment may not be the right one. This is so because this phase B segment might not have been added to the funnel sequence (even though visible to the (*x*,*y*) edge). The case happens when a vertex is found to the right of the edge, as in vertex 8 for edge (6,7) of Figure 32. When such a vertex is in phase B of a vertex that needs it as *r*, it will not be found in the funnel sequence. One solution is to skip phase B segments until one is found that is in the funnel sequence. However, the requirement of constant time makes finding a better solution necessary. For this, storing the first phase B segment added (not necessarily the one with the highest slope) will yield the right *r* in this situation. It can be shown that it will exist in the funnel sequence, and the vertices that got skipped will not be in the funnel sequence and so will not be visible to the new vertex anyway (at least in this path to the edge being split).

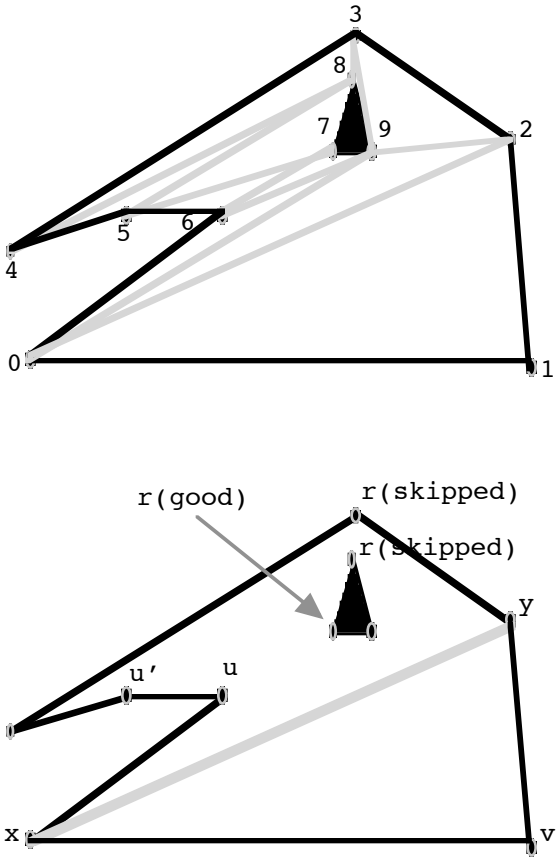


Figure 12 - A Split Where *r* Is A Special Case

Going through the example of this in Figure 12, at vertex 7, the algorithm produces triangulation edges (5,7) and (6,7). At vertex 8, because it lies to the right of the (6,7) edge, it will not be added to the (6,7) funnel sequence (even though it is clearly visible to the edge). Because (0,9) and (0,2) are derivatives of (6,7), vertex 8 will not be in their funnel sequences either. So when vertex 1 is processed (and splits (0,2)), vertex 8 cannot be the *r* when *u* is 6. The same is true for vertex 3. The *r* that works here is vertex 7 - and this is the first phase B segment seen by vertex 6.

A case similar to the preceding arises in the traversal from *s* to *r*. If in phase B, the CCX walk may visit a vertex that is not in the funnel sequence (because it is to the right of *q*). It was solved with an extra check to skip one or more vertices. If the vertex is to the right of *q*, then it must be skipped. But, in order to meet the theoretical time bounds (because this may happen many times) the result of this skipping is cached.

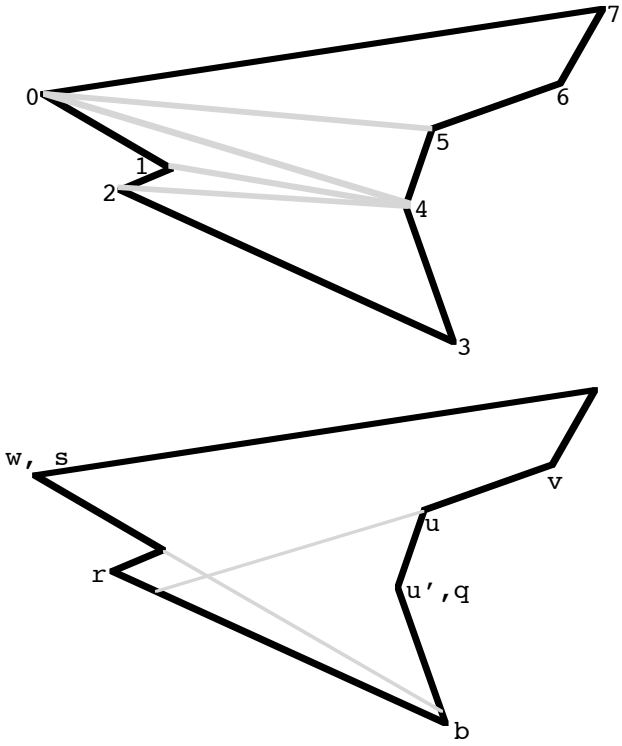


Figure 13 - Skipping Vertices in the *s* to *r* Walk

An example of this is shown in Figure 13. Here, vertex 3 will not be in the funnel sequence of edge (0,5) even though it is visible to it. When vertex 6 splits (0,5), it will have *u*=5 first. In this case, *u'*=4 and since vertex 5 has no other children, *r*=2 and *q*=4 since it is the predecessor of vertex 5 in the linked list. Then, *q*'s upper parent (*s*) is vertex 0. In the walk from *s* to *r*, it should visit vertices 1 then 2. However, after vertex 1, the CX(0,1) is found to be vertex 3(*b*) instead of vertex 2(*r*). Because vertex 3 is to the right of *q*, it should be skipped. The skipping goes to vertex 2 - the correct one. After vertex 6 is complete, vertex 7 is processed. A same situation will occur, but because the result of the skip should have been cached (when handling vertex 6), the walk can go directly to the right vertex (2 in this case).

Finally, the split-find data structure was not fully implemented. The split part was implemented (dovetailed doubling search), but the interval finds were coded as a linear search starting at the list head. In the regular regression testcases, this search iterated no more than four times (effectively a constant). Only in the random testcases did this count go up (to 16). However, even with these testcases, the average number of iterations stays less than 4.0. Given the theoretical nature of the full split-find, one would might even see a slow down if fully implemented.

Performance Comparisons

Timing Methodology

At a high level, some rules about the implementation were enforced to get more meaningful comparisons. For example, all implementations must handle collinearities and must use the same non-visibility interpretation (see Input/Output Assumptions). Also, all codes must actually store the visibility graph - not just report the edges. Finally, if the code uses a balanced binary search tree, it must use the same basic AVL implementation (only key comparisons can be altered). As mentioned above, every algorithm should be a reasonable implementation, with performance enhancements added when found. A reasonable attempt was made to find as many improvements as possible (none of which change the asymptotic bounds, only the effective constants). These improvements are detailed in the appendices. Of course, the fastest running version of each algorithm was used for the timings.

Actual timings were carried out by instrumenting the codes with `gettimeofday()` calls. The start time is read *after* the input is read (in order to eliminate I/O variations in the operating system). The end time is read when the algorithm has finished (and just before `exit()`). For correctness, all implementations were first verified against a set of about 200 regression tests. Of course, during timings, all output is turned off (again to eliminate I/O delays in the timings).

The usual steps have to be taken to ensure equal comparisons. All binaries were compiled with the same compiler and with the same optimization flags[10]. All tests were done on the same hardware[11] , same operating system[12] , etc. All timings were taken on a non-networked machine with no other user-level task running (and identical system-level processes).

Finally, when timings were taken, five consecutive runs were carried out on each testcase. The high time and the low time were discarded. The middle three were then averaged.

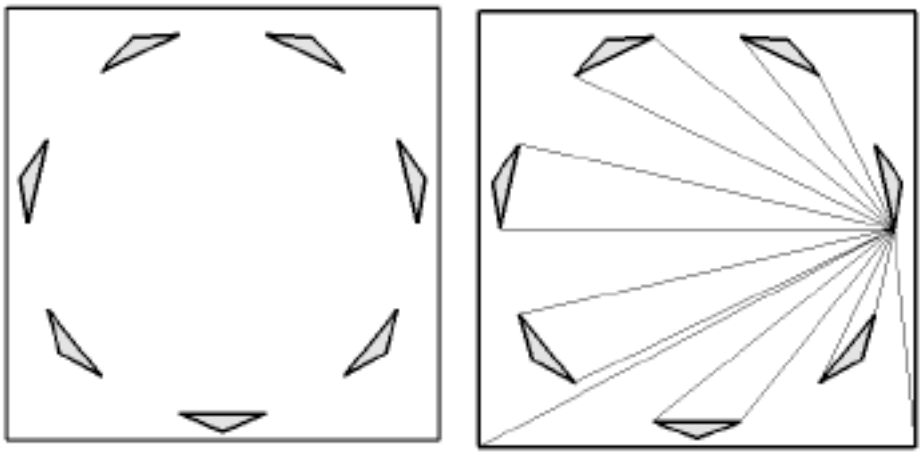
Timing Results

Five basic sets of testcases were run and the results of these were plotted to verify the implementation's asymptotic running time and to show the divergence of the algorithms' running time for larger n. Also, it is instructive to know the crossover points, i.e. when one

[10] Gnu gcc 2.95.2 compiler with -O3 optimization
[11] iMac G3 PowerPC 750 400Mhz CPU, 32K L1 cache, 512K L2 cache, 320MB memory
[12] Mac OS X 10.1.5 (based on FreeBSD)

algorithm becomes better than another due to constant factors. These sets will be subsequently referred to as A) Circle of Obstacles, B) *n*-gon, C) Square Grid of Obstacles, D) Line of Triangles, E) Spirals, and F) Random. Each (except for Random) has a known bound of the number of visibility edges.

Circle of Obstacles - The first set described here has $O(n^2)$ visibility edges. It consists of a number of triangles spread equally apart around a center - see Figure 14. In this manner the outer two vertices can see those of every other obstacle, so the bound is $O(n^2)$. The number of obstacles range from three (13 vertices) to 24 (76 vertices).



(a) (b)

Figure 14 - Polygonal Region with $O(n^2)$ Visibility Edges, e.g. n=25
(a) region with no visibility edges shown (b) one point's visibility

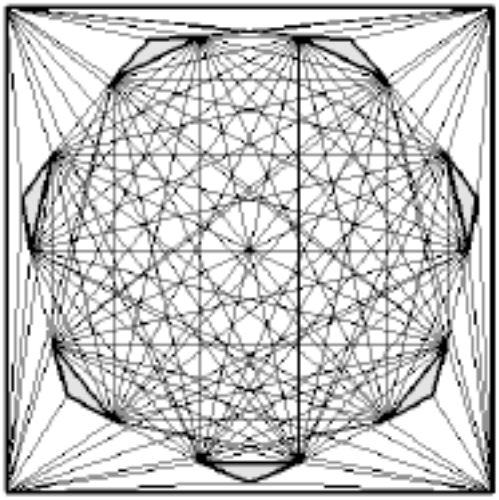


Figure 15 - The Entire Visibility Graph of the Example in Figure 14

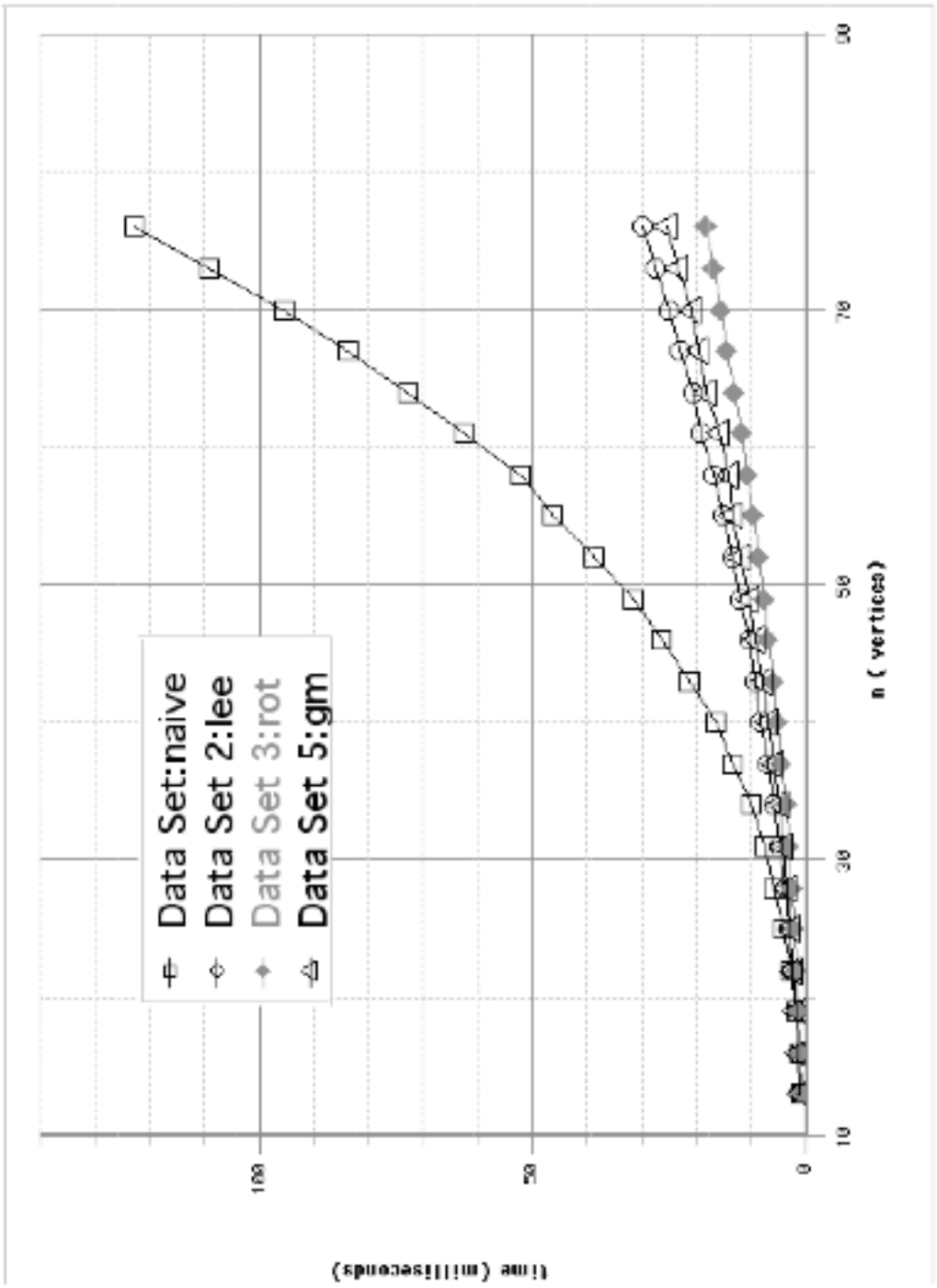
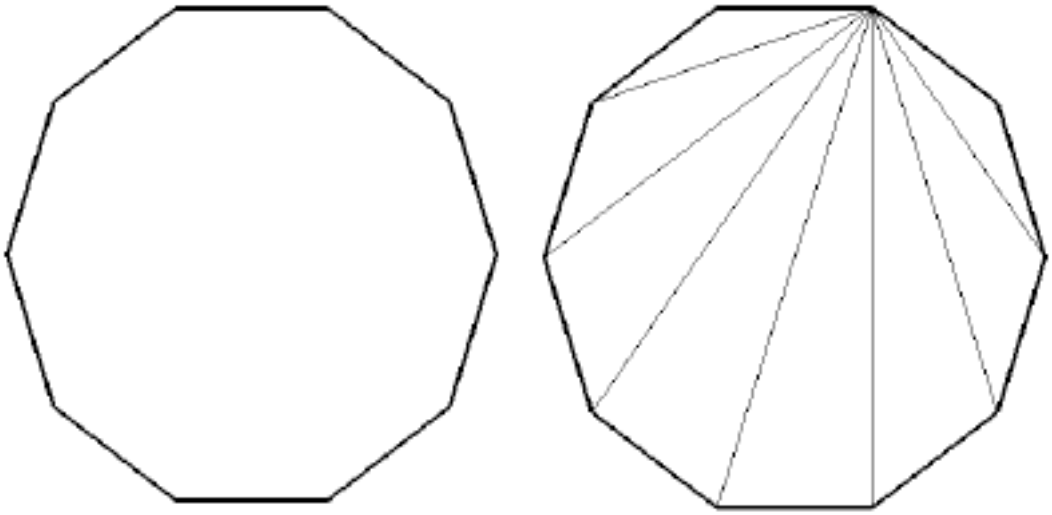


Figure 16 - Plot of Execution Times for the Circle of Obstacles Set

Figure 16 shows the results of the timings on this set of testcases. The actual numbers are tabled in Appendix E. Obviously, the Overmars/Welzl algorithm does best for large n. The reason it runs faster than Ghosh/Mount (even though both have the same complexity with $|e| = O(n^2)$), is that it has a low constant. For example, it has no dynamic allocation (except to store the visibility graph itself). Further analysis shows certain crossover points. The Naive algorithm does better than any other for $n \| 13$, after this Overmars/Welzl method still performs the best. Also, Lee's method does better than Ghosh/Mount for $n \| 19$, but not beyond this.

n-gon - While similar to the Circle of Obstacle set, the *n*-gon set has no obstacles (holes). It also has $O(n^2)$ visibility edges. Figure 17 shows an example. The set has a range of vertices from ten to 90 vertices. (Note: obviously a specialized program could be written for just convex polygons and would run incredibly fast in comparison, but it was thought that it might be interesting to see how these generalized algorithms will perform on a trivial problem.)



(a) (b)

Figure 17 - *n*-gon with $O(n^2)$ Visibility Edges, e.g. n=10
(a) without visibility shown (b) one point's visibility

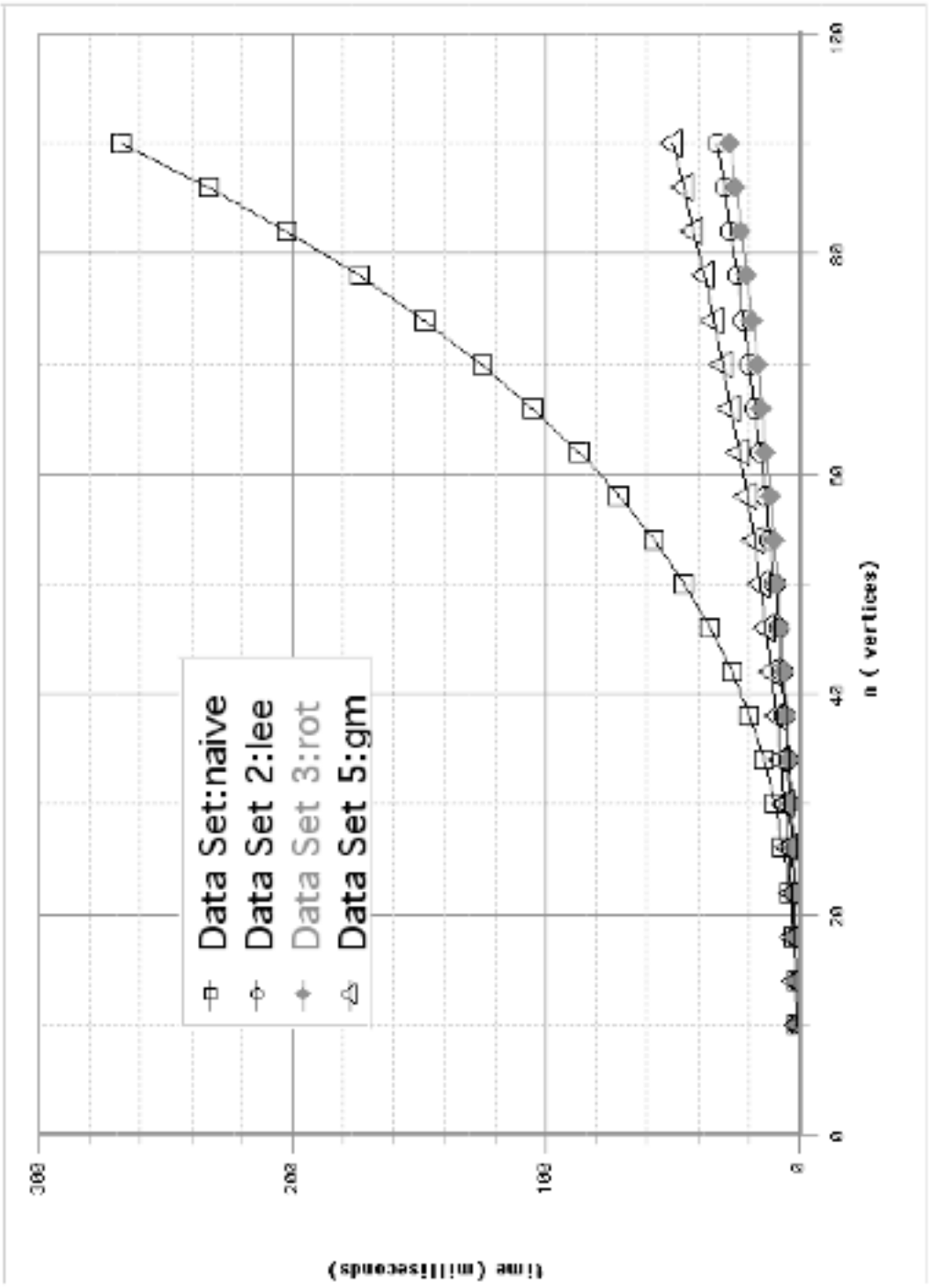


Figure 18 - Plot of Execution Times for the *n*-gon Set

Figure 18 contains the plots of the results (see Appendix E for the exact numbers). The main performance difference between this set and the Circle of Obstacles: Lee outperforms Ghosh/Mount at large n. (But the gap between Lee and Ghosh/Mount becomes smaller for large n. At n=90, Lee runs in about 65% the time of Ghosh/Mount and at n=5000, Lee runs about 76% of Ghosh/Mount.) This somewhat surprising result can be explained by looking closer at what the Ghosh/Mount code is doing. At each new vertex, only one Mehlhorn triangle gets added - thus at the top there is only one split operation. All the other vertices must be reached by recursive calls to the split operation. The Ghosh/Mount algorithm does well when it can eliminate pockets of visibility because these can be concatenated onto the end of the funnel sequence in constant time. But, since there are no hidden pockets in this set of testcases, the constant is high for all pairs of vertices. Table 1 shows the running times for large n (n=5000).

Naive	Lee	Overmars/Welzl	Ghosh/Mount
>4 hours	171 seconds = 2.84 minutes	90.7 seconds = 1.51 minutes	224 seconds = 3.73 minutes

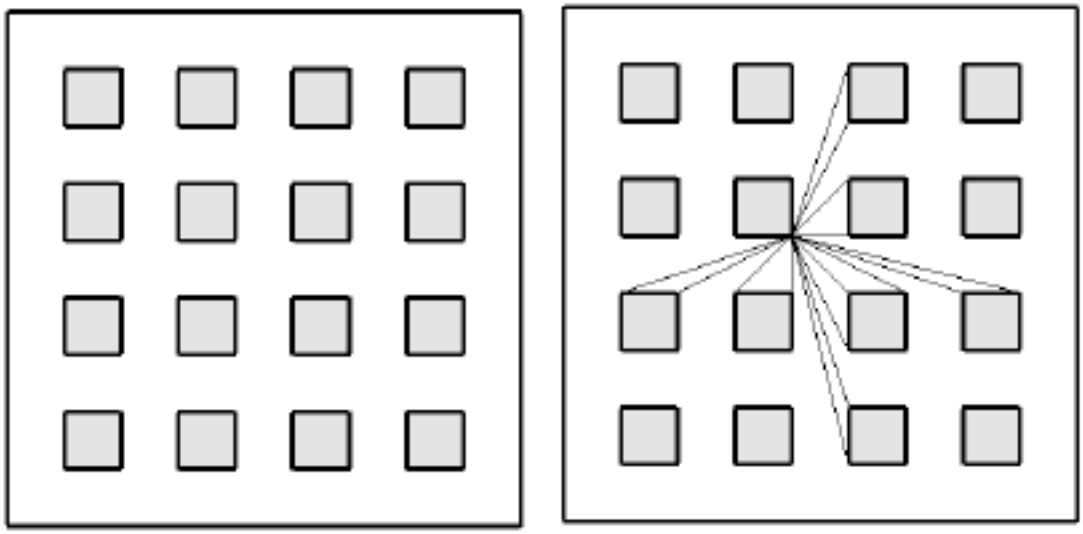
Table 1 - Measurements for the n-gon Testcase with n=5,000[13]

For very large visibility graphs, the Ghosh/Mount algorithm actually runs out of memory. With vertices of an n-gon at n=10,000, it ran out of memory (even with `unlimit` set in the shell). The reasons are various. First, the actual split function as coded has many local variables and because it is recursive, all the frames put together eat up stack space. Also, the t-to-s chain of subsequent splits are placed on a linked list which is dynamically allocated. But more basically is the fact that each visibility pair (both from/to and to/from) requires15 words of storage in the data structure. When there are $O(n^2)$ edges in the graph, this really eats up memory. This is in contrast to the other algorithms that require only two or three words per pair. The reason the Lee algorithm does not hit this limit in the n-gon set is that at any stage in any of the scans, edge tree only has one node in it.

[13] for these big testcases, the Naive method did not finish after 4 hours user time; also, the other measurements here are based on one sample with the BSD `time` utility (user time)

Square Grid of Obstacles - The next set of testcases test the algorithms with a $|e|$ somewhere between quadratic and linear. In fact, these have $O(n^{3/2})$ vertices. This set ranges n^2 from eight to 148, with n=68 shown in Figure 19.

Figure 20 shows the plots of the timings (see Appendix E for the exact values). Here, the timings for large n reflect exactly what one might expect, i.e. Ghosh/Mount does best, followed by Overmars/Welzl, followed by Lee, followed by Naive. The crossover points show that Naive does best with $n||8$, Lee never does better than Overmars/Welzl, and Ghosh/Mount does better than Overmars/Welzl only when n exceeds 68. Also, to give an idea of running times with many vertices, this testcase was run on n=7748 (a grid of 44x44 squares). Table 2 shows the results.



(a) (b)

Figure 19 - Polygonal Region with $O(n^{3/2})$ Visibility Edges, e.g. n=68
(a) region with no visibility edges shown (b) one point's visibility

Naive	Lee	Overmars/Welzl	Ghosh/Mount
>4 hours	540 seconds = 9.00 minutes	128 seconds = 2.13 minutes	9.37 seconds

Table 2 - Measurements of the Square Grid of Obstacles Testcase with n=7748[14]

[14] for these big testcases, the Naive method did not finish after 4 hours user time; also, the other measurements here are based on one sample with the BSD `time` utility (user time)

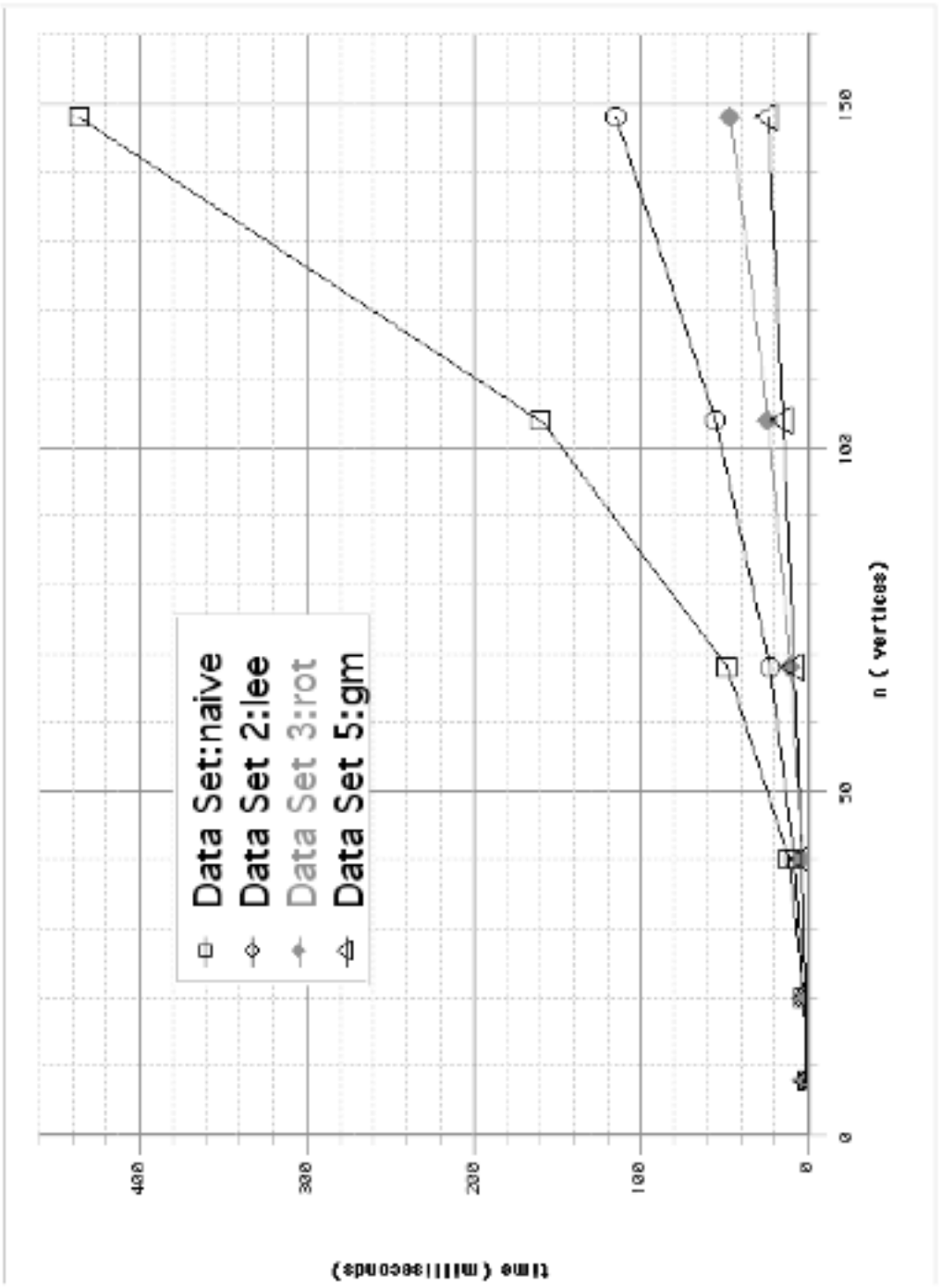
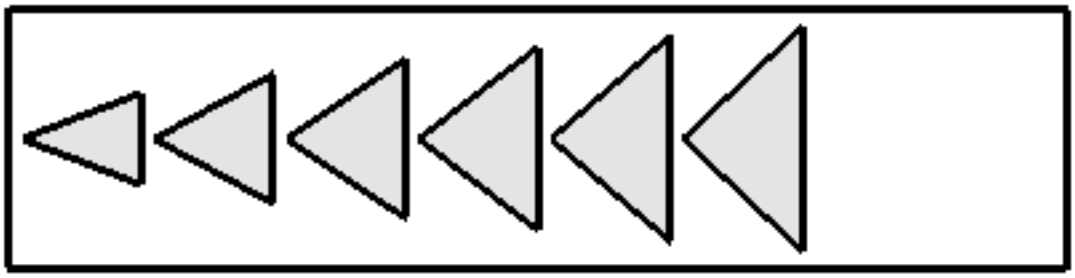


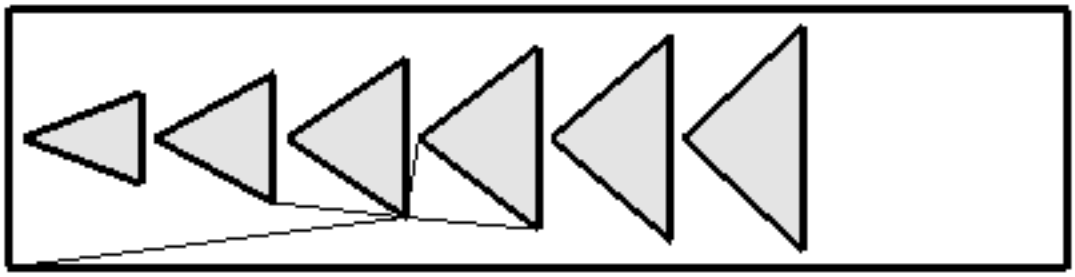
Figure 20 - Plot of Execution Times for Square Grid of Obstacles Set

Line of Triangles - This set has a linear number of visibility edges, i.e. $O(n)$ with most vertices being able to see just a few others (essentially a constant). The timings were carried out in the range $22 \le n \le 154$. Figure 21 shows an example.

The timings across n are plotted in Figure 22 (see Appendix E for the exact values). The results are as expected for large n with Ghosh/Mount being quicker than Overmars/Welzl, followed by Lee, and lastly Naive. The crossover points in this set: Naive faster than Lee until n=46, Lee never beating Overmars/Welzl, and Ghosh/Mount outdoing Overmars/Welzl at $n \ge 28$. To give an idea about how much faster Ghosh/Mount is for large n, some additional measurements were taken for n=10,000. Table 3 shows the results.



(a)



(b)

Figure 21 - Polygonal Region with $O(n)$ Visibility Edges, e.g. n=22
(a) region with no visibility edges shown (b) one point's visibility

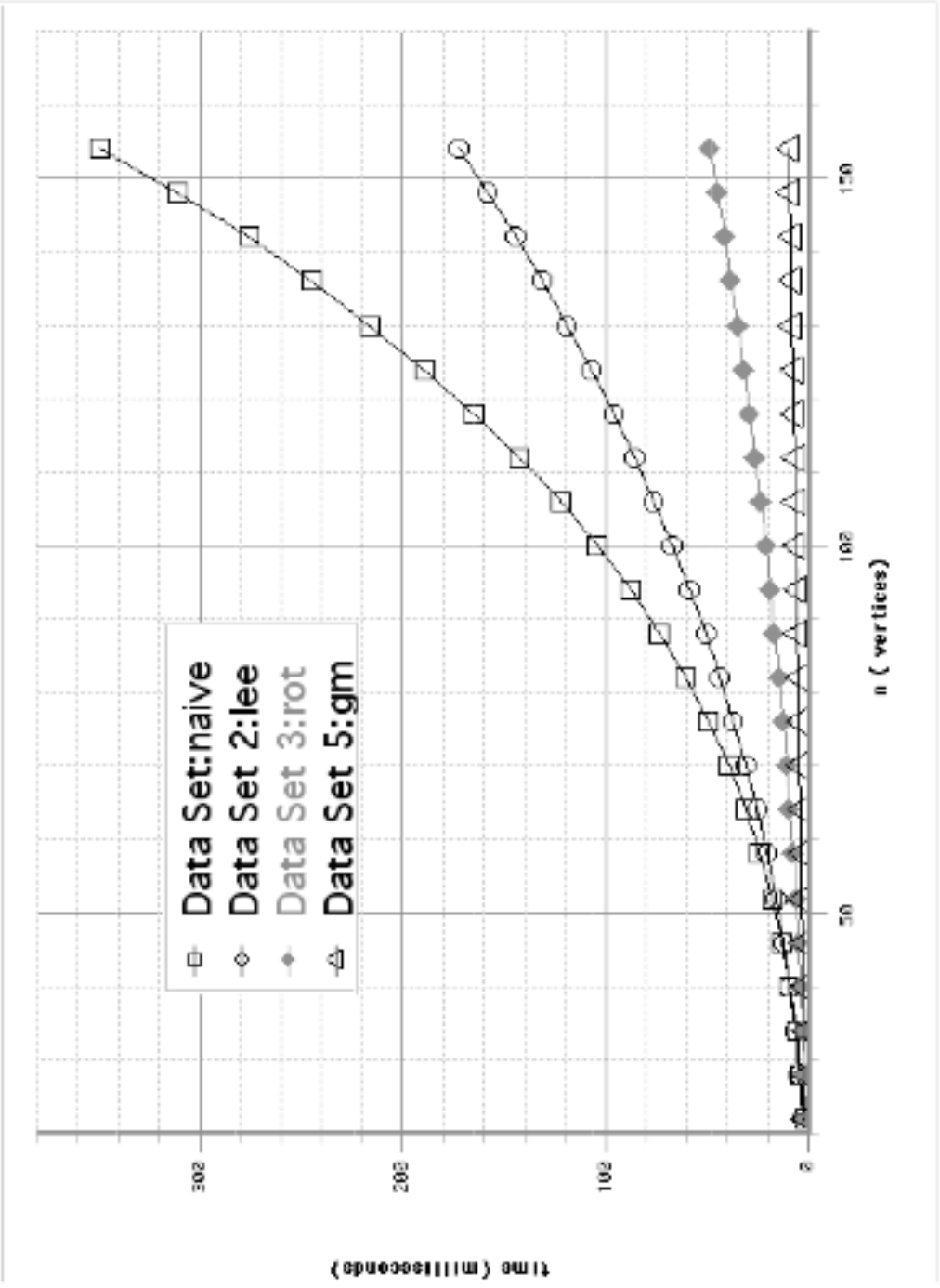


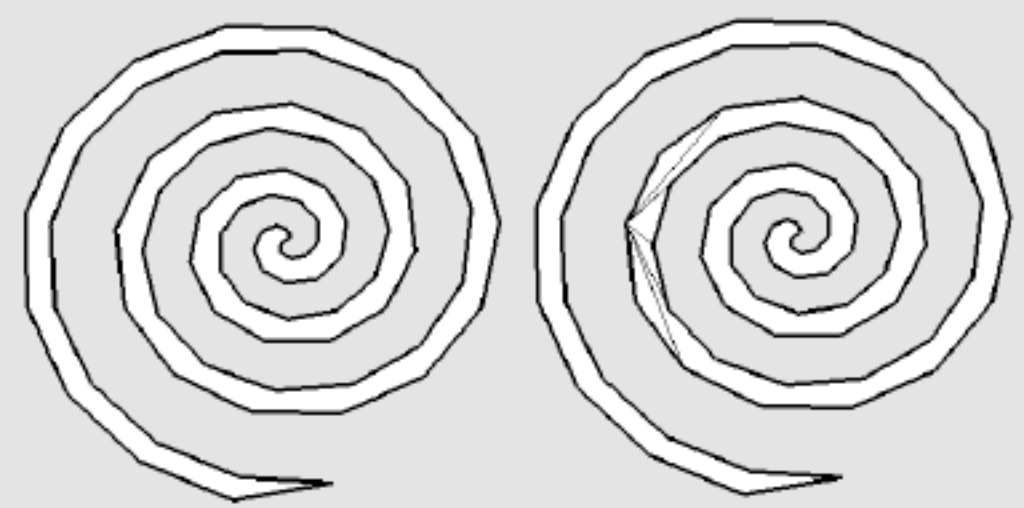
Figure 22 - Plot of Execution Times for the Line of Triangles Set

Naive	Lee	Overmars/Welzl	Ghosh/Mount
>3 hours	1410 seconds = 23.6 minutes	234 seconds = 3.90 minutes	0.730 seconds

Table 3 - Measurements of the Line of Triangles Testcase with n=10,000[15]

Spirals - The spirals represent another linear set of testcases, except this set is a single polygon (no obstacles/holes). $O(n)$ is the upper bound on visible edges since each vertex has a small constant of other vertices which it can see. Here n ranges from six to 196. Figure 23 shows an example with n=86.

Figure 24 shows the timings plot. For large n, the algorithms diverge as the Line of Triangles set. One difference earns mention. Here, Lee does considerably worse than with the Line of Triangles set. One explanation is that with Lee and this sort of polygon, many vertices have deep edge trees as the scan sweeps around them.



(a) (b)

Figure 23 - Spirals with $O(n)$ Visibility Edges, e.g. n=86
(a) without visibility shown (b) one point's visibility

[15] for these big testcases, the Naive method did not finish after 4 hours user time; also, the other measurements here are based on one sample with the BSD `time` utility (user time)

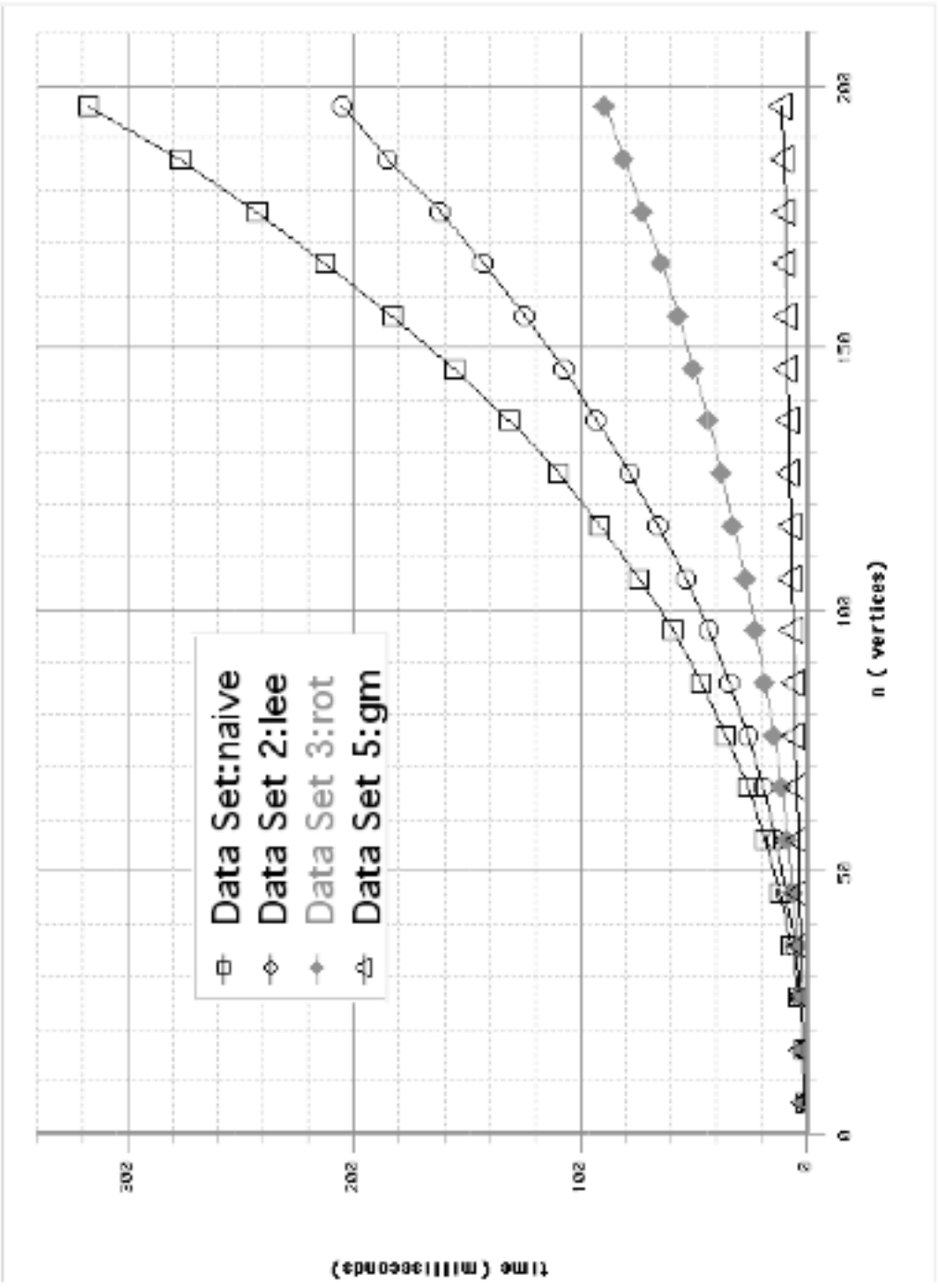
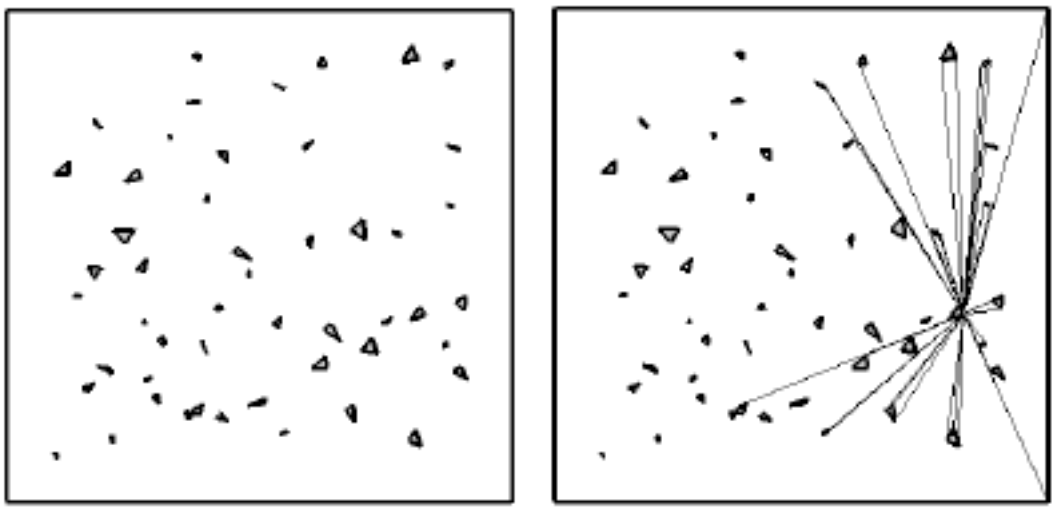


Figure 24 - Plot of Execution Times for the Spirals Set

As for crossovers, these are similar to the other linear set (see Appendix E for exact values). Naive does better than Lee until $n \geq 26$. Lee has better times than Overmars/Welzl at $n=6$ but not after. Finally, Overmars/Welzl is better than Ghosh/Mount but only for $n \| \| 26$.

random - This set was randomly generated where the number of triangles can be specified. Their size, orientation, and placement are random within ranges. Also, no overlaps are allowed. Figure 25 shows an example with 50 triangles (n=154).



(a) (b)

Figure 25 - Random Region, e.g. n=154
(a) without visibility shown (b) with one point's visibility

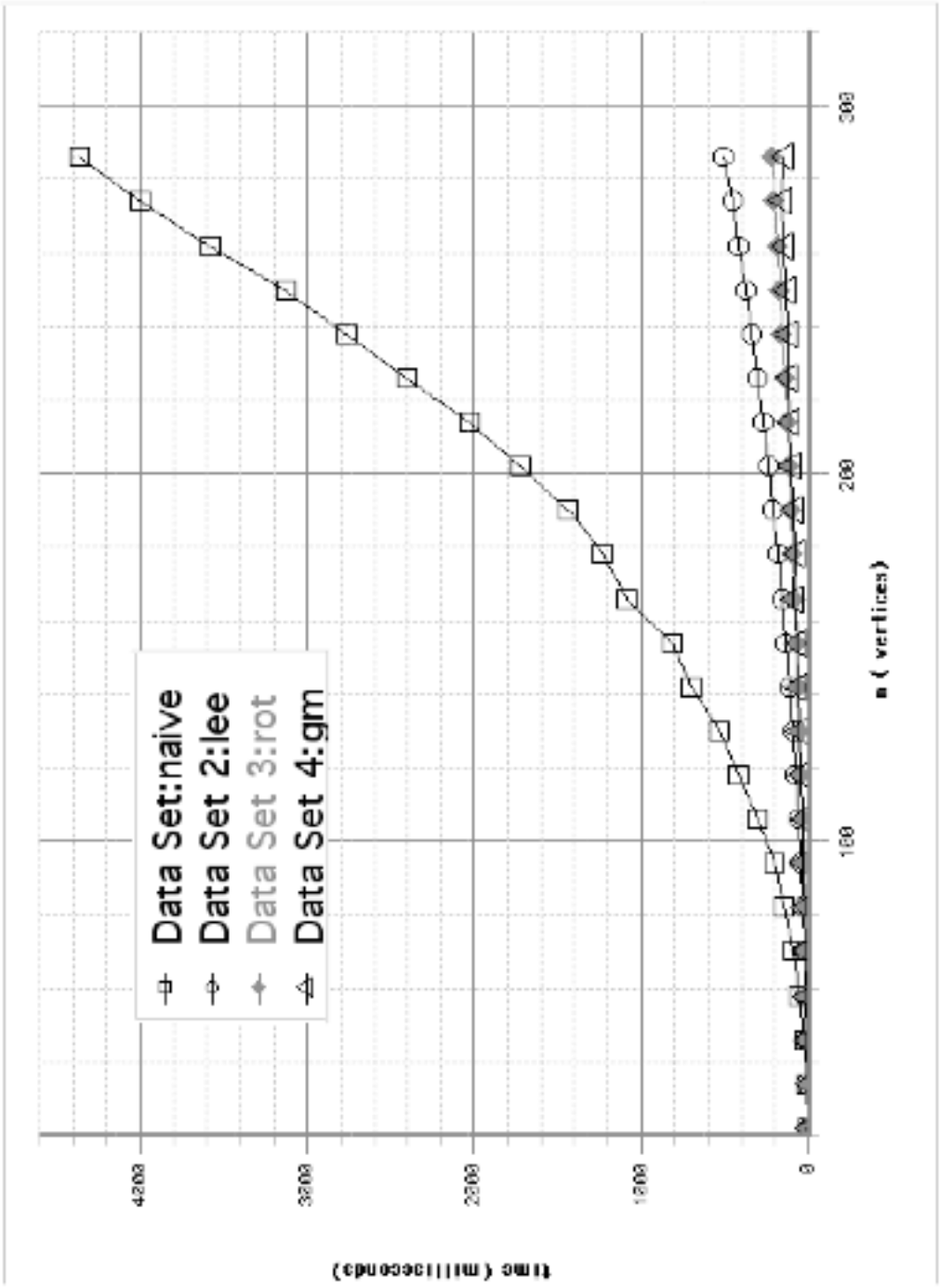


Figure 26 - Plot of Execution Times for the Random Set

Figure 26 shows the timing results (with the exact figures in Appendix E). Towards larger *n* one sees the results resembling some of the other sets, i.e. Lee does worse than Overmars/Welzl and Ghosh/Mount does slightly better than Overmars/Welzl. However, the visibility graph becomes more and more sparse as *n* becomes bigger. So one sees a relatively high crossover point between Overmars/Welzl and Ghosh/Mount. Here, the crossover happens at about *n*=150. Another phenomena seen is that there is a point at which Ghosh/Mount actually performs better at larger *n* than it did for smaller *n* due to the fact that the triangular obstacles become more dense in the same area. This is only seen between the last two data points of the graph, but it has been observed nonetheless.

As for very big *n*, one would expect an even greater separation and this can be shown in Table 4, when *n*=5000.

Naive	Lee	Overmars/Welzl	Ghosh/Mount
>4 hours	253 seconds = 4.22 minutes	63.9 seconds = 1.06 minutes	1.08 seconds

Table 4 - Measurements of the Random Testcase with *n*=5,000[16]

[16] for these big testcases, the Naive method did not finish after 4 hours user time; also, the other measurements here are based on one sample with the BSD `time` utility (user time)

Other Aspects and Comparisons

This section merely discusses some other issues involved with the visibility graph and how they relate to the algorithms implemented here.

Space Complexity

The Naive implementation uses O(n) working storage and stores the graph in $O(|e|)$. The working arrays are allocated at the beginning after n has been read from the input. The Lee implementation has the same space complexity, but requires dynamic allocation/deallocation in the AVL trees as each vertex is processed. The Overmars/Welzl implementation has the same working storage and graph storage complexities and uses dynamic allocation in its only AVL tree, but it only needs to do this once to fill the tree at the beginning. The Ghosh/Mount implementation inherently requires $O(|e|)$ working storage and this can not be avoided since the walking operations (CX, etc.) need what has been discovered previously in order to work correctly. This is in contrast to the other algorithms where it is an option whether or not to actually store the graph (however, in these tests, storing the graph was chosen in order to obtain a fair comparison). One solution that can be applied to the Ghosh/Mount implementation is to free up the heavy representation (15 words) after the algorithm has finished and copy the relevant information into a lighter structure (two or three words) as the others have.

As for running out of memory, only the Ghosh/Mount algorithm ran into this problem (in some very large n-gon testcases as mentioned previously). However, unrestricting memory usage in the shell alleviated the problem in all but the very biggest testcase. In the paper by Agarwal, et. al. [AAAS93], it is shown that the visibility graph can be more compactly stored using clique covers, however, there is not much savings. The paper shows that the space required for a visibility graph with quadratic number of edges has a lower bound of $\Omega(n^2/\log^2 n)$ and an upper bound of $O(n^2/\log n)$, where the sum of the clique sizes is of course n. So, this would be some help for all the implementations, especially the Ghosh and Mount code since it needs so many memory words per visible pair.

Code Complexity

Naturally, the Naive method is the simplest, with Lee and Overmars/Welzl being about equal with medium complexity. The Ghosh/Mount method is clearly the most complex requiring a triangulation, funnel sequences, the complicated split procedure, and structures needed for

the graph traversal operations. Overmars/Welzl state something about Ghosh/Mount in their paper [OW88]. "Although the latest method is optimal in time, the authors themselves state that the method will be very hard to implement and implementations might be slow due to high constants". It goes without saying that code complexity is a factor for development time and maintenance.

Efficiency of Queries

The shortest path between two vertices (or even all-pairs shortest path) can be calculated using Dijkstra's algorithm as mentioned previously. However, often enough the shortest path between two arbitrary points within the free space of the polygonal region is desired. Without rerunning the program, it would be nice to keep the basic visibility graph stored, then run something quick on just the query points, and finally apply shortest path to the resultant graph. This process could then be repeated as often as desired given that the polygonal region does not change - only the query points.

Perhaps the easiest way to do this in Lee's algorithm is to run one iteration of the algorithm on the start and end points and include those visibility segments in the graph before running Dijkstra's. Here the running time of the queries would be $O(n \log n)$ which beats any algorithm run completely from scratch. However, there does not seem to be a simple way to run just part of Overmars/Welzl or part of Ghosh/Mount. For these algorithms, Lee's algorithm could be imported for the queries.

Dynamic Changes to the Region

None of the algorithms compared in this paper were designed with dynamic changes to the region in mind. As implemented, each would have to be rerun. However, it is interesting to note that the Asano algorithm [AGHI86] can handle dynamic changes (inserts and deletes) each in $O(n)$ time to maintain the visibility graph. Also, one Vegter paper [Veg91] shows some results for dynamic changes via the *visibility diagram*. The time here is $O(\log^2 n + k \log n)$ where k is the number of visibility edges created or destroyed at the change.

Some other papers allow dynamic changes for maintaining the parallel view [EOW83] or the view from a point [EOW83] [Poc90] [Riv97].

The Outer Visibility Graph

As the "Input/Output Assumptions" states, only the *inner* visibility graph is calculated. The *outer* visibility graph is not. Comparing what it would take to allow this, in the Naive, Lee, and Overmars/Welzl methods, it is as easy as eliminating one line of code (one check). However, the Ghosh/Mount implementation would need some work. The part that would require change is the Mehlhorn triangulation. Instead of doing only the inner triangulation, it would have to calculate the convex hull followed by the complete triangulation. As such, it would have to keep track of chains in the out-intervals as well (not just the in-intervals).

The Vertex-Edge Visibility Graph

Each algorithm can also be relatively easily modified to compute the *vertex-edge visibility graph*. The vertex-edge visibility graph is a bipartite graph where the vertices are on one side and the edges are on the other. It is considered a weak visibility graph since a vertex may be able to see only part of an edge. From it, other structures including the visibility polygon, the visibility graph, and the visibility complex can be found [OS97a] (with some restrictions on collinearities).

Conclusion

Four classical approaches to the visibility graph problem with general polygonal holes have been coded. Each was tuned to get a reasonable implementation before overall measurements were taken. The results follow the time complexity of each: $O(n^3)$ for Naive, $O(n^2 \log n)$ for Lee, $O(n^2)$ for Overmars/Welzl, and $O(|e| + n \log n)$ for Ghosh/Mount. Crossover points have been identified so as to give an idea of which algorithms do best at different testcase sizes.

This paper's main result is in the implementation of the Ghosh and Mount algorithm with complications identified and solutions found. As a bit of a surprise, the algorithm does not have outrageous running-time constants and often betters other approaches for even medium-sized testcases.

Future work

A second algorithm for sparse graphs was detailed in Overmars/Welzl's paper [OW88], and this could be compared against the implementations described here, especially against Ghosh/Mount. If this $O(|e| \log n)$ approach is as simple to implement as the $O(n^2)$ approach (first algorithm in the paper), it may end up being fastest on testcases yielding sparse graphs. If it holds true, then for any conceivable testcase, exactly one of the two Overmars/Welzl approaches would run faster than any other approach. However, choosing which one to run might be difficult to decide beforehand. For completeness, Asano's $O(n^2)$ approaches [AGHI86] may do well also; but without an implementation, it is difficult to say.

Other output-sensitive algorithms exist. First, Kapoor and Maheshwari's approach [KM00] via triangulation corridors could be implemented to see how it does against Ghosh/Mount. Also, Riviere's work [Riv97] and Pocchiola and Vegter's work [PV95] (appearing simultaneously) find the visibility graph of convex objects by effectively carrying out a topological sweep of the visibility complex [PV93]. This approach should be convertible to line segments (convex polygons with two vertices) and then adapted to simple polygons in the same way that the Overmars/Welzl implementation was adapted here. Doing this would add another $O(|e| + n \log n)$ algorithm to the offerings.

Appendices

Appendix A - Naive Method
Appendix B - Lee's Method
Appendix C - Overmars and Welzl's Method
Appendix D - Ghosh and Mount's Method
Appendix E - Tables of Measurements Used in the Plots
Appendix F - Rotation Tree Details

Appendices A-D list local optimizations for the methods under study. This basically simply means "tricks" that were found in order to make the implementation more of a "reasonable implementation". These range from minor things like removing checks (after verification, of course) to more significant improvements, such as only scanning half the plane (-|/|/2 to |/|/2) versus (-|/|/2 to 3|/|/2). However, everything reasonable found to speed it up was put in (or taken out as the case may be), so as to make for comparisons that are as unbiased as possible when the algorithms are run against each other. Code profiling[17] served as a good tool in order to see where the most time was being spent. The heavy functions would then yield the most gain if improvements could be found there.

During this stage, the algorithms were run against themselves as the optimizations were put in place one after another. This shows incremental improvement. The same test cases were used as each algorithm was improved in this local manner. These testcases are the following:

> "linear154" - a testcase with a linear number of visible edges. It has 154 vertices.
>
> "box148" - a testcase with a grid of boxes as obstacles. It has $O(n^{3/2})$ number of edges and 148 vertices.
>
> "quad58" - a testcase with $O(n^2)$, or quadratic, number of edges. It has 58 vertices.

These testcases are some of the same as those used in the algorithm-to-algorithm comparisons seen in the main body of this paper. The point was simply to get a rough idea of how the algorithm was running and then to show how well the optimizations were helping. Of course, the tests were run under similar conditions, i.e. the same compiling (level 1 optimization), same operating system under similar loads, same hardware, etc. Times were taken for three consecutive runs and these averaged. (See also "Test Methodology" under Performance Comparisons.)

[17] profiling via the gcc `-pg` compile option and `gprof`

Appendix A - Tuning the Naive Method

The vanilla version of the Naive algorithm has none of the tricks to increase performance. About the only optimization it does do is to break out of the third level loop if a blocking (interfering) edge is found. This is natural since there would be no sense in continuing with the loop if an interfering edge was already known to exist.

The second version has certain basic tricks to increase performance. First, the second-level loop does not have to iterate between 0 and (n-1). It can start at (center+1). Thus, instead of n^2 third- level loop executions, it is (n^2- n)/2 such loop executions. Second, the third-level loop does not have to be run at all if the visibility segment would be outside the polygon (the check for this runs in constant time). Third, visibility segments to adjacent edges can automatically be reported. This case, of which there are $O(n)$ instances, allows the third-level loop to be skipped entirely.

The third version uses squared distances for all the distance calculations. This saves on a square-root calculation for every such distance calculation. There were also some checks removed and other minor cleanup in this final version.

Tables A-1 through A-3 shows the timings.

	version A	version B	version C
run 1	1.089229	0.400753	0.381233
run 2	1.089229	0.398765	0.381034
run 3	1.092338	0.397977	0.381415
average	1.092338	0.399165	0.381227

Table A-1 - “linear154” Measurements for Naive Versions (seconds)

	version A	version B	version C
run 1	1.138958	0.471737	0.470984
run 2	1.131248	0.487799	0.469526
run 3	1.135616	0.485430	0.467120
average	1.135274	0.481655	0.469210

Table A-2 - “box148” Measurements for Naive Versions (seconds)

	version A	version B	version C
run 1	0.159817	0.091614	0.086841
run 2	0.160112	0.091391	0.089992
run 3	0.162831	0.093823	0.089207
average	0.160920	0.092276	0.088680

Table A-3 - “quad58” Measurements for Naive Versions (seconds)

An analysis on the data shows that the optimizations in the second version (version B) have the most effect. Here the three tests show 63.5%, 57.6%, and 42.7% decreases in running time. The third version improvements (from version B) are 4.49%, 2.58%, and 3.90%. The net improvements (A to C) are 65.1%, 58.7%, and 44.9%.

Appendix B - Tuning Lee's Method

The initial version has no special optimizations.

The second version adds an AVL replace optimization. This occurs when the scan line encounters a vertex where an edge has to be removed from the edge tree but at the same time the adjacent edge has to be added. The initial version explicitly does an AVL remove followed by an AVL insert. This version makes use of AVL replace because the structure of an AVL tree will be the same in this situation.

The third version cuts the scan to half the plane. The technique was actually seen in Welzl's paper [Wel85]: the scan only goes from -/ //2 to / //2 instead of -/ //2 to 3/ //2. In other words, points to the left can be ignored since visibility between left points and the center are checked when those left points become center. This works because vertex #1 being visible to vertex #2 implies vertex #2 being visible to vertex #1. The expected improvement would be from n^2 to (n choose 2) which is $n^2 - n$.

The fourth version uses a different distance calculation. The previous versions used a calculation based on law of sines. This version uses an intersection method.

The fifth version further improves the distance calculation by using squared distances, saving a square root calculation each time.

Tables B-1 through B-3 show the timing results.

	version A	version B	version C	version D	version E
run 1	0.466150	0.403958	0.232384	0.235857	0.205477
run 2	0.465280	0.403789	0.231688	0.235235	0.204244
run 3	0.468591	0.400092	0.230854	0.236261	0.207602
average	0.466674	0.402613	0.231642	0.236261	0.205774

Table B-1 - "linear154" Measurements for Lee Versions (seconds)

	version A	version B	version C	version D	version E
run 1	0.306626	0.253694	0.161173	0.163999	0.149053
run 2	0.305961	0.252035	0.157366	0.160045	0.146518
run 3	0.306121	0.252590	0.156148	0.160793	0.146863
average	0.306236	0.252773	0.158229	0.161612	0.147478

Table B-2 - "box148" Measurements for Lee Versions (seconds)

	version A	version B	version C	version D	version E
run 1	0.068181	0.064328	0.018498	0.018899	0.017319
run 2	0.068015	0.062913	0.018289	0.019601	0.018023
run 3	0.067858	0.063352	0.018289	0.018905	0.017181
average	0.068018	0.063531	0.018359	0.019135	0.017508

Table B-3 - "quad58" Measurements for Lee Versions (seconds)

The improvements are summarized in Table B-4. As one can see, version D actually makes the performance decrease slightly. However, with squared distances in version E (not possible in versions A, B, or C since they use the law-of-sine method for distances), the gain in going to intersection method for distances is realized in version E.

	A to B	B to C	C to D	D to E	C to E	net (A to E)
linear154	13.7%	42.5%	-1.99%	11.2%	11.2%	55.9%
box148	17.3%	37.4%	-2.14%	8.75%	6.79%	51.8%
quad58	6.6%	71.1%	-4.23%	8.50%	4.64%	74.3%

Table B-4 - Lee Measured Improvements from Version to Version

Appendix C - Tuning Overmars and Welzl's Method

The primary version of this method has most optimizations already in place. A priori, the paper [OW88] specifies a fast loop and gives detailed pseudocode, so it is easy to code well the first time. Also, the half-plane optimization for the scan, i.e. scanning in the range (-/ //2 to / //2) versus (-/ //2 to 3/ //2) is inherent. Next, initially choosing an array (of size n) to define the stack will automatically beat a slower dynamic memory implementation.

The secondary version basically just has slightly tighter integration and adds some "inline" keywords to key functions.

Tables C-1 through C-3 list the measurements taken for the three testcases.

	version A	version B
run 1	0.089400	0.079857
run 2	0.087995	0.079690
run 3	0.087848	0.082210
average	0.082027	0.080586

Table C-1 - "linear154" Measurements for Overmars/Welzl Versions (seconds)

	version A	version B
run 1	0.079429	0.078487
run 2	0.076375	0.081570
run 3	0.077023	0.076842
average	0.077781	0.076733

Table C-2 - "box148" Measurements for Overmars/Welzl Versions (seconds)

	version A	version B
run 1	0.011185	0.010864
run 2	0.011243	0.010867
run 3	0.011216	0.010854
average	0.011215	0.010862

Table C-3 - "quad58" Measurements for Overmars/Welzl Versions (seconds)

Some improvement can be seen here: 1.76%, 1.35%, and 3.15% for the three testcases run.

Appendix D - Tuning Ghosh and Mount's Method

The first version represents a good basic working version of the algorithm. Some attempts at the start were made to make it efficient. Some of these are listed here:

> Inlining all the graph traversal primitives, such as CW, CCW, CCX, and CX instead of making them separate routines as the paper suggests. This allows some invocations to be more optimal. For example, sometimes CCX (counter-clockwise extension) does not need a check to see it exists before retrieving it.
>
> Pre-calculating slopes between adjacent vertices since they are referenced so often.
>
> Saving slope calculations between nonadjacent visible vertices.

The second version makes one optimization: it replaces dynamically allocated phase-A segments in favor of an array. This is possible because although during integration of a new vertex there is no way to know a-priori how many segments in phase-A there might be (making a linked-list the obvious choice), an array can be used because the count will never exceed *n*.

The third version removes over 20 checks, inlines some functions, and does some other minor cleanup.

	version A	version B	version C
run 1	0.009963	0.009430	0.009218
run 2	0.009927	0.009318	0.009276
run 3	0.009941	0.009290	0.009240
average	0.009944	0.009346	0.009245

Table D-1 - "linear154" Measurements for Ghosh/Mount Versions (seconds)

	version A	version B	version C
run 1	0.026982	0.024531	0.024255
run 2	0.026874	0.024217	0.024438
run 3	0.026652	0.024338	0.024218
average	0.026836	0.024362	0.024304

Table D-2 - “box148” Measurements for Ghosh/Mount Versions (seconds)

	version A	version B	version C
run 1	0.017291	0.015657	0.015962
run 2	0.017674	0.015633	0.015882
run 3	0.017602	0.015854	0.016026
average	0.017522	0.015715	0.015957

Table D-3 - “quad58” Measurements for Ghosh/Mount Versions (seconds)

The improvement from version A to B shows better times with 6.01%, 9.22%, 10.3% decreases in execution time. The improvements between B and C show 1.08%, 0.238%, and -1.54% decreases.

Appendix E - Tables of Measurements Used in the Plots

n	Naive	Lee	OverMars/Welzl	Ghosh/Mount
13	0.627333	0.876	0.655333	1.08633
16	1.10867	1.27633	0.922	1.413
19	1.80967	1.77733	1.201	1.84667
22	2.70533	2.381	1.55767	2.232
25	3.94933	3.02867	1.97167	2.739
28	5.66433	3.763	2.49933	3.54
31	7.62767	4.785	3.07633	4.26067
34	10.0883	5.699	3.598	4.89433
37	13.252	6.80833	4.33033	5.928
40	16.5177	8.15167	5.01	6.73767
43	21.3013	9.16267	5.923	7.94167
46	26.5263	10.461	6.73467	9.35733
49	31.6517	11.9803	7.55033	10.2873
52	38.8597	13.5293	8.69633	11.9067
55	46.207	15.181	9.72867	13.3927
58	52.241	16.9097	10.5493	14.0037
61	62.3763	19.1317	11.7917	15.9427
64	72.5783	20.7137	13.01	18.1357
67	83.6147	22.9063	14.3497	19.5777
70	95.256	25.013	15.5633	21.0793
73	109.251	27.3473	16.9723	23.1813
76	122.891	29.6913	18.2403	25.3453

Table E-1 - Average Measures for the Circle of Obstacles Testcases (milliseconds)

n	Naive	Lee	Overmars/Welzl	Ghosh/Mount
10	0.386	0.451	0.489	0.888
14	0.976	0.811333	0.775667	1.50833
18	2.05467	1.30733	1.21967	2.23967
22	3.738	1.90933	1.76667	3.20667
26	6.27067	2.61367	2.35667	4.38
30	9.64367	3.494	3.15367	5.65767
34	14.143	4.55833	3.99167	7.232
38	19.7243	5.64533	4.98967	8.96967
42	26.6477	6.963	6.098	10.9047
46	35.2433	8.331	7.261	13.0893
50	45.4027	9.86333	8.543	15.5073
54	57.0717	11.532	9.907	17.834
58	71.0367	13.3893	11.396	20.755
62	86.759	15.265	13.231	23.7163
66	104.987	17.3873	14.753	26.9067
70	125.075	19.52	16.5143	30.1803
74	147.968	22.0687	18.4097	33.7983
78	173.65	24.3713	20.4317	37.5737
82	201.851	27.101	22.7453	41.652
86	233.055	29.7587	24.928	45.7953
90	267.318	32.6047	27.421	50.1567

Table E-2 - Average Measures for the *n*-gon Testcases (milliseconds)

n	Naive	Lee	Overmars/Welzl	Ghosh/Mount
8	0.217	0.362667	0.344667	0.647333
20	1.714	1.79867	1.12633	1.62767
40	11.0333	7.41567	3.76	3.80667
68	48.2607	22.3253	10.3	7.90133
104	159.865	55.0297	23.225	14.427
148	436.143	115.255	46.1103	24.0993

Table E-3 - Average Measures for the Square Grid of Obstacles Testcases (milliseconds)

n	Naive	Lee	Overmars/Welzl	Ghosh/Mount
22	1.98033	2.53667	1.29133	1.53433
28	3.67667	4.19267	1.94833	1.832
34	5.91833	6.32233	2.77533	2.18367
40	9.05167	9.11333	3.71933	2.52
46	12.971	12.3993	4.816	2.92167
52	17.8503	16.1127	6.036	3.261
58	23.8447	20.3453	7.37067	3.59167
64	31.156	25.233	8.90767	3.98
70	39.362	30.7983	10.619	4.27833
76	49.1217	37.0413	12.3717	4.71833
82	60.677	43.7177	14.2687	5.06867
88	73.2313	50.739	16.44	5.429
94	87.9147	58.6857	18.5923	5.66067
100	103.873	66.8503	20.8123	6.06467
106	122.203	76.0343	23.3143	6.369
112	142.427	85.741	25.9703	6.90167
118	164.607	96.1663	28.8047	7.173
124	189.036	107.041	31.682	7.609
130	215.851	119.017	34.8673	7.96667
136	244.684	131.251	37.8487	8.39033
142	275.976	144.505	41.1747	8.74333
148	310.975	158.051	44.9147	9.02367
154	348.924	172.73	48.3293	9.48333

Table E-4 - Average Measures for the Line of Triangles Testcases (milliseconds)

n	Naive	Lee	Overmars/Welzl	Ghosh/Mount
6	0.134667	0.243333	0.285333	0.51933
16	1.11533	1.17733	0.936333	1.29767
26	3.47867	2.71467	2.00033	2.03833
36	7.055	5.235	3.614	2.77133
46	12.0387	9.09733	5.65233	3.284
56	18.1737	13.3447	8.081	3.676
66	25.774	19.349	10.97	4.177
76	35.6173	25.8687	14.2067	4.69367
86	46.3217	34.3657	17.9927	5.37067
96	59.2153	43.2343	22.1043	5.894
106	74.2973	53.2263	26.7817	6.45633
116	91.11	65.6593	32.076	6.87267
126	109.587	77.739	37.6443	7.14867
136	131.42	93.226	43.5777	7.545
146	155.649	107.57	49.878	8.02067
156	182.56	124.506	56.762	8.55333
166	212.922	143.226	64.0077	8.953
176	242.648	162.067	72.139	9.44967
186	276.804	185.11	80.9153	9.958
196	317.619	205.62	89.2953	10.411

Table E-5 - Average Measures for the Spiral Testcases (milliseconds)

n	Naive	lee	rot	gm
22	3.52033	2.45867	1.77333	3.10633
34	11.614	5.93733	3.846	5.38967
46	28.2883	10.9027	6.89967	10.072
58	54.2437	17.5553	10.6533	14.3207
70	88.9927	26.293	15.0187	19.3233
82	148.126	36.048	20.591	26.2823
94	206.711	47.8077	26.315	30.3683
106	302.939	61.814	33.2183	38.8783
118	417.984	77.1103	41.2883	48.563
130	530.488	94.045	49.6703	56.9717
142	706.105	113.798	59.0237	65.5533
154	812.984	134.946	67.03	65.4707
166	1088.76	156.479	79.8877	86.0473
178	1237.04	183.463	88.7133	84.136
190	1441.2	209.734	101.147	95.6897
202	1717.15	240.279	112.437	103.5
214	2026.58	269.549	125.927	109.288
226	2403.76	304.545	140.215	121.109
238	2762.12	340.734	154.054	129.29
250	3123.42	375.169	170.146	141.015
262	3585.12	418.165	185.35	144.365
274	4004.03	455.583	200.732	153.243
286	4360.9	510.073	215.486	142.9

Table E-6 - Average Measures for the Random Testcases (milliseconds)

Appendix F - Rotation Tree Details

Listed here are properties, lemmas, theorems, and pseudocode of the rotation tree [OW88].

Property 1

1. p_{+INF} is the root of the tree

2. The incoming edges of a node are ordered by slope where the edge with the smallest slope is the leftmost

3. For each edge $\overline{pq}$, $-\frac{\pi}{2} < \text{slope}(\overline{pq}) < \frac{\pi}{2}$.

4. If $\overline{pq}$ and $\overline{qr}$ are edges then slope($\overline{pq}$) $<$ slope($\overline{qr}$)

Property 2 (order)

Let *p*, *q*, and *r* be nodes in the tree where $\overline{pq}$ is an edge.

If slope($\overline{pq}$) < slope($\overline{pr}$) $< \frac{\pi}{2}$, then *r* precedes *p* in the preorder of the tree, or, in other words, either *r* lies on the path from *p* to the root or it is in a left subtree of this path. If $-\frac{\pi}{2} <$ slope($\overline{pr}$) < slope($\overline{pq}$) then *r* succeeds *p* in the preorder of the tree.

Property 3 (cone)

Let $\overline{pq}$ and $\overline{rs}$ be edges in the tree where slope($\overline{pq}$) $<$ slope($\overline{pr}$) < slope($\overline{rs}$). Then no point in V lies in the cone which is the intersection of the open halfplane to the right of $\overline{rs}$ and the closed halfplane to the left of $\overline{pr}$.

Lemma 1

Let G be a rotation tree on V and let $\overline{pq}$ be the rightmost leftmost leaf-edge in G. If $\overline{pq}$ is replaced by $\overline{pz}$ where z is the next point (after q) in order around p then the resulting graph is also a rotation tree.

Lemma 2

Let G be a rotation tree on V and let $\overline{pq}$ be the leftmost edge of q (q != p_{+INF}). Let r be the father of q and let z' be the left brother of q, if it exists. Then the next point z around p (after q) is the tangent (from the right) from p to the chain ending in $\overline{z'r}$, or, if z' does not exist, $z=r$.

Theorem 1

Given a set of n points in the plane, in time O(n^2) and storage O(n) one can find for each point p the other points sorted by angle around p.

Theorem 2

Given a set S of n non-intersecting line segments in the plane, the visibility graph G_S can be constructed in time O(n^2) and storage O(n).

The Rotation Tree Data Structure needs four (4) pointers:

(1) pointer to parent
(2) pointer to left sibling
(3) pointer to right sibling
(4) pointer to righmost child

Pseudocode for needed functions:

```
b[] Sort(a[]) - sorts vertices a[] by decreasing x,
                returns as set b[]
void AddRightmost(a,b) - adds a as rightmost son to b
```

```
void Handle(a,b) - process between a and b
void Remove(a) - detaches a from the tree
void AddLeftOf(a,b) - adds a as left brother of b
bool LeftTurn(a,b,c) - returns true if c lies left of ab,
                       else false
+ basic stack operations
```

Pseudocode of main routine:

```
p = Sort(v);
AddRightmost( p_-INF , p_+INF );
for(i = 0 to n-1) AddRightmost( p_i , p_-INF );
InitStack; Push( p_0 );
while !EmptyStack
  p = pop();
  p_r = RightBrother(p);
  q = Father(p);
  if (q != p_-INF ) Handle(p,q);
  z = LeftBrother(q);
  Remove(p);
  if ( z == null || !LeftTurn(p,z,Father(z)) ) AddLeftOf(p,q);
  else
    while (RightmostSon(z) != null &&
           LeftTurn(p,RightmostSon(z),z) )
      z = RightmostSon(z);
    AddRightmost(p,z);
    if (z == top()) z = pop();
  if (LeftBrother(p) == null && Father(p) != p_-INF ) push(p);
  if ( p_r != null) push( p_r );
```

References

[AVL62] Adelson-Velskii, G. M. and E. M. Landis, *An algorithm for the organization of information*, Soviet Mathematics Doklady, 3 (1962), 1259-1263.

[AAAS93] Agarwal, P. K., N. Alon, B. Aronov and S. Suri, *Can visibility graphs be represented compactly?*, in Proceedings of the 9th Annual ACM Symposium on Computation Geometry, San Diego, CA (1993), pp. 338-347.

[Ang00] Angel, E., *Interactive Computer Graphics a Top-Down Approach with OpenGL, 2nd ed.*, Addison Wesley Longman, Inc., Reading, MA (2000).

[Ang02] Angel, E., *OpenGL: A Primer*, Addison-Wesley Longman, Inc., Boston, MA (2002).

[AGHI86] Asano, T., T. Asano, L. Guibas, J. Hershberger, H. Imai, *Visibility of disjoint polygons*, Algorithmica, 1 (1986), pp. 49-63.

[CGL83] Chazelle, B., L. J. Guibas and D. T. Lee, *The power of geometric duality*, in Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science, Tucson, AZ (1983), pp. 217-225.

[BKOS00] de Berg, M., M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry Algorithms and Applications, 2nd ed.*, Springer-Verlag, Berlin (2000).

[Dij59] Dijkstra, E. W., *A note on two problems in connexion with graphs*, Numerische Mathematik, 1 (1959), pp. 269-271.

[EG86] Edelsbrunner, H. and L. Guibas, *Topologically sweeping in an arrangement*, in Proceedings of the 18th Annual Symposium on Theory of Computing (1986), pp. 389-403.

[EOS83] Edelsbrunner, H., J. O'Rourke and R. Seidel, *Constructing arrangements of lines and hyperplanes with applications*, in Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science, Tucson, AZ (1983), pp. 83-91.

[EOW83] Edelsbrunner, H., M. H. Overmars and D. Wood, *Graphics in flatland: a case study*, Advanced in Computing Research, 1 (1983), JAI Press, Inc., pp. 35-59.

[EGA81] El-Gindy, H. and D. Avis, *A linear algorithm for computing the visibility polygon from a point,* Journal of Algorithms, Vol. 2 (1981), pp. 186-197.

[GT85] Gabow, H. N. and R. E. Tarjan, *A linear-time algorithm for a special case of disjoint set union*, Journal of Computer and System Sciences, 30 (1985), pp. 209-221.

[GM87] Ghosh, S. K. and D. M. Mount, *An output sensitive algorithm for computing visibility graphs*, in Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA (1987), pp. 11-19.

[GM91] Ghosh, S. K. and D. M. Mount, *An output-sensitive algorithm for computing visibility graphs*, SIAM Journal on Computing, Vol. 20, No. 5 (1991), pp. 888-910.

[Gho88] Ghosh, S. K., *On recognizing and characterizing visibility graphs of simple polygons*, in Proceedings of the 1st Scandinavian Workshop on Algorithm Theory, Springer-Verlag Lecture Notes in Computer Science, vol. 318, (1988), pp. 96-104.

[GHLST87] Guibas, L., J. Herschberger, D. Leven, M. Sharir, and R. E. Tarjan, *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, 2 (1987), pp. 209-233.

[HM91] Hefferman, P. J. and J. S. B. Mitchell, *An optimal algorithm for computing visibility in the plane*, in Proceedings of the 2nd Workshop WADS'91, Ottawa, pp. 437-448.

[Her87] Herschberger, J., *Finding the visibility graph of a simple polygon in time proportional to its size*, in Proceedings of the 3rd Annual Symposium on Computational Geometry, Ontario, (1987), pp. 11-20.

[HU73] Hopcroft, J. E. and J. D. Ullman, *Set merging algorithms*, SIAM Journal on Computing, 2 (1973), pp. 294-303.

[KM88] Kapoor, S. and S. N. Maheshwari, *Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles*, in Proceedings of the 4th Annual ACM Symposium on Computational Geometry, Urbana, IL, (1988), pp. 172-182.

[KM00] Kapoor, S. and S. N. Maheshwari, *Efficiently constructing the visibility graph of a simple polygon with obstacles*, SIAM Journal on Computing, Vol. 30, No. 3 (2000), pp. 847-871.

[LaP90] LaPoutre, J.A., *Lower bounds for the union-find and the split-find problem on pointer machines*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (1990), 34-44.

[Lee78] Lee, D. T., *Proximity and reachability in the plane*, Ph. D. thesis and Tech. Report ACT-12, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL (1978).

[LSC99] Lee, J., S. Y. Shin and K. Chwa, *Visibility-based pursuit-evasion in a polygonal room with a door*, in Proceedings of the 15th Annual ACM Symposium on Computational Geometry, Miami Beach, FL, (1999), pp. 119-128.

[LPW79] Lozano-Perez, T. and M. A. Wesley, *An algorithm for planning collision-free paths among polyhedral obstacles*, Communications of the ACM, 22 (1979), pp. 560-570.

[Meh84a] Mehlhorn, K., Data Structures and Algorithms, *Volume 1: Sorting and Searching*, Springer-Verlag, Berlin (1984).

[Meh84b] Mehlhorn, K., Data Structures and Algorithms, *Volume 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlin (1984).

[Meh84c] Mehlhorn, K., Data Structures and Algorithms, *Volume 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin (1984).

[MS91] Moret, B. M. E. and H. D. Shapiro, *Volume I Design and Efficiency*, The Benjamin-Cummings Publishing Company, Inc., Redwood City, CA, (1991).

[OS97a] O'Rourke, J. and I. Streinu, *The vertex-edge visibility graph of a polygon*, Smith Technical Report 047, (1996).

[OS97b] O'Rourke, J. and I. Streinu, *Vertex-edge pseudo-visibility graphs: characterization and recognition*, in Proceedings of the 13th Annual ACM Symposium on Computational Geometry, Nice, France, (1997), pp. 119-128.

[ODRP96] Orti, R., F. Durand, S. Riviere and C. Peuch, *Using the visibility complex for radiosity computation*, in 1st ACM Workshop on Applied Computational Geometry, WAGC'96, Philadelphia, PA, Springer-Verlag Lecture Notes in Computer Science, vol. 1148, (1996), pp. 177-190.

[OW87] Overmars, M. H., and E. Welzl, *Construction of sparse visibility graphs*, Technical Report RUU-CS-87-9, Department of Computer Science, University of Utrecht, (1987).

[OW88] Overmars, M. H., and E. Welzl, *New methods for constructing visibility graphs*, in Proceedings of the 4th Annual ACM Symposium on Computational Geometry, Urbana, IL (1988), pp. 164-171.

[Poc90] Pocchiola, M., *Graphics in flatland revisited*, in Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Springer-Verlag Lecture Notes in Computer Science, vol. 447, (1990), pp. 85-96.

[PV93] Pocchiola, M. and G. Vegter, *The visibility complex*, in Proceedings of the 9th Annual ACM Symposium on Computational Geometry, San Diego, CA (1993), pp. 328-337.

[PV95] Pocchiola, M. and G. Vegter, *Computing the visibility graph via pseudo-triangulations*, in Proceedings of the 11th Annual ACM Symposium on Computation Geometry, Vancouver, B.C., (1995), pp. 248-257.

[Riv95] Riviere, S., *Topologically sweeping the visibility complex of polygonal scenes*, in Proceedings of the 11th Annual ACM Symposium on Computational Geometry, Vancouver, B.C., (1995), pp. 436-437.

[Riv97] Riviere, S., *Dynamic visibility in polygonal scenes with the visibility complex*, in Proceedings of the 13th Annual ACM Symposium on Computational Geometry, Nice, France, (1997), pp. 421-423.

[TV84] Tarjan, R. E. and J. van Leeuwen, *Worst-case analysis of set union algorithms*, Journal of the ACM, Vol. 31, No. 2 (1984), 245-281.

[Veg90] Vegter, G., *The visibility diagram: a data structure for visibility problems and motion planning*, in Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Springer-Verlag Lecture Notes in Computer Science, vol. 447, (1990), pp. 97-110.

[Veg91] Vegter, G., *Dynamically maintaining the visibility graph*, in Proceedings of the 2nd Workshop WADS'91, Ottawa, pp. 425-436.

[Wei95] Weiss, M. A., *Data Structures and Algorithm Analysis, 2nd ed.*, The Benjamin-Cummings Publishing Company, Inc., Redwood City, CA, (1995).

[Wel85] Welzl, E., *Constructing the visibility graph for n-line segments in O(n^2) time*, in Information Processing Letters, 20 (1985), pp. 167-171.