# Lee's $O(n^2 \log n)$ Visibility Graph Algorithm Implementation and Analysis

Dave Coleman
*Department of Computer Science**
(Dated: May 2, 2012)

## I. ABSTRACT

Visibility graphs have many applications, including finding the shortest path, robotic motion planning and the art-gallery problem. The first non-trivial algorithm developed for finding the visibility graph runs in $O(n^2 \log n)$ and is presented in this paper. Its correctness, space, and time usage are mathematically analyzed. Because faster algorithms to solve this problem have been discovered, this paper is somewhat original in its analysis and implementation details of an otherwise forgotten algorithm.

## II. INTRODUCTION

### A. Visibility Graph

Visibility is an important property in computational geometry and is used in many different types of problems, structures and algorithms [8]. One of the most basic structures is the *visibility graph*, where an input graph $G$ describes a set of geometric primitives in an $d$-dimensional space, and the output visibility graph $G_v$ describes the visibility between every vertex and every other vertex in $G$. Here, we define *visibility* as the ability to run a straight line between two vertices without crossing any other edge in the input graph $G$. In this way, two visible vertexes are said to be *unobstructed* by any obstacle, and a line is drawn between them in the output $G_v$. An example visibility graph in $d = 2$ dimensions Euclidean space is shown in Figure 1.

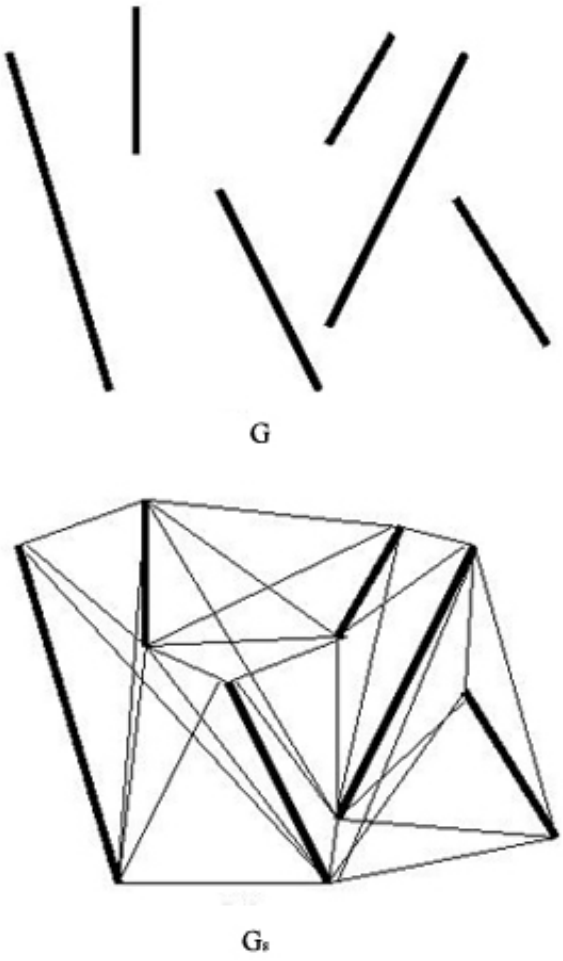


FIG. 1: Top: An input graph $G$ consisting of a set of simple line segments. Bottom: a generated visibility graph $G_v$ describing all possible non-obstructed connections between each vertex.

The set of geometric primitives in$G$ can consist of a variety of different types of shapes: rectilinear, circular, line segments, convex polygons or, most generally, simple polygons. Many different algorithms have been developed based on the assumption of which types of geometric primitives are allowable. In this paper we will focus on simple non-intersecting line-segments, so as to simplify our proofs and analysis. Very little modification would be required to expand the problem space to general polygons.

The layout of the geometric primitives is another variation between computational geometry papers. In some, visibility within a simple polygon is the only problem space, but more often there exists obstacles within the space, also referred to as holes or islands. Another variation of the visibility graph for all points is finding the visibility *tree* for just one point, which is simply a subproblem of the visibility graph for all points.

*david.t.coleman@colorado.edu; www.davetcoleman.com

### B. Applications

Visibility graphs are most often thought of for use in Euclidean shortest-path problems, were a start point $s$ and end point $t$ are given and the task is to find the optimal continuous path through the obstacle space without violating physical constraints. This application exploits the fact that the shortest paths are found on the arcs of the visibility graph. Once the visibility graph has been constructed, the shortest path problem can be trivially solved using well known algorithms such as Dijkstra's [6], A* search[5], or Floyd-Warshall [3] algorithms.

Robotic motion planning is a common sub problem of the shortest path problem, as demonstrated in Lozano-Perez and Wesley's 1979 collision-free paths algorithm [11]. One of the most famous examples of visibility graphs used in robotic motion planning is Shakey the Robot [11]. However, the application of visibility graphs realistically limits the workspace to two dimensions and is generally computationally intractable for modern real-world robotics problems. Sampling-based approaches are considered the current state of the art and, although unable to determine that no path exists, have a probability of failure that decreases to zero as more computational time is spent [7].

Additional applications of visibility graphs include finding the minimum dominating set to help solve the art gallery problem and in solving pursuer-evader problems [10]. Finally, visibility graphs can be used to optimize radio antenna placement, urban planning and architectural design [1].

### C. History

The naive approach to computing the visibility of a graph runs in $O(n^3)$ times. The first non-trivial solution to this problem was developed by D.T. Lee in his 1978 Ph.D. dissertation that ran in $O(n^2 \log n)$ time [9]. The solution is included at the end of his thesis as somewhat of a side thought and it has since then received very little attention in the computational geometry field. Only available upon email request, the typed report includes hand-written edits and drawings. This is the algorithm that will be analyzed in this paper.

In the 1980's a large number of $O(n^2)$ visibility graph papers were published, most of which entailed a topological sort of the vertex pairs. E. Welzl in particular described this technique using an arrangement of the dual of the vertices that required $O(n^2)$ working space [13]. The working storage of the topological sweep was later improved to $O(n)$ by Edelsbrunner and Guibas [2]. Further improvements included handling dynamic updates of the workspace, using less running time on average, or handle sparse graphs more efficiently. One of the last papers published on visibility graphs during this time period achieved $O(|e| + n \log n)$ time bounds, which are output-sensitive algorithms optimal for graphs of a certain minimum density threshold [4].

## III. METHODS

### A. Description of the Algorithm

Lee's $O(n^2 \log n)$ algorithm computes the visibility graph $G_v$ from $G(V, E)$ by computing the visibility graph of a single vertex $n$ times. For each vertex $v_i \in V$, the visibility of all other vertices is calculated by 1) sorting all surrounding vertices in angular order from some starting scan line, 2) using a rotating plane sweep technique to visit each vertex in angular order and 3) keeping track of the distance of each surrounding line segment on the scan line in a sorted data structure. The following details these 3 procedures:

For each $v_i \in V$ a visibility *tree* is generated describing the visibility of all other points $v_j \in V - v_i$ with respect to $v_i$. Each visibility tree is created by setting $v_i$ to be the center vertex $c$. A starting *scan line* vector $\vec{s}$ is initialized for each $c$, with the origin of the vector at $c$. Its direction is irrelevant for the algorithm but in this paper and implementation $\vec{s}$ will be assumed to be the horizontal unit vector $\hat{i}$ pointing straight and to the right from $c$, i.e. $\vec{s} = \hat{i} = [1, 0]$.

From scan line $\vec{s} = \hat{i}$ we calculate the counter clockwise angle $\theta_i = angle(\vec{s}_{c \to v_i}, \hat{i})$ for every vertex $v_i \in V - c$. The angles are inserted to an optimally sorted data structure $A$ from smallest to largest.

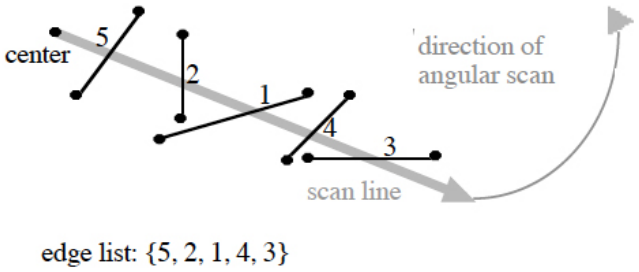


FIG. 2: Example of an initialized edge list with all edges that intersect scan line $s$. Image courtesy of [8]

A second optimally sorted data structure $E_s$ is initialized containing all the line segments $l_i \in E$ that intersect scan line $\vec{s}$ at the start of the algorithm. This operation requires $|E|$ checks, calculating if $\vec{s}$ and line segment $l_i$ intersect. The line segments that are found

to intersect are inserted into $E_s$ with their keys being the distance from $c$ to the intersection point $v_{[i,i-1]}$ such that the root of the sorted data structure always contains the edge closest to $c$. We can intuitively see that this root edge is the only edge visible from $c$ at this scan line instance. Figure 2 depicts an initialized scan list, though in this example vector $\vec{s}$ is not horizontal.

After the initialization phase the algorithm visits every vertex $v_i$ in order of $\theta_i$ in $A$. The scan line does not have to actually visit every angle in the circle, but only those $\theta_i$ where a vertex $v_i$ intersects $\vec{s}$. For each $v_i$ scanned, the algorithm decides if its corresponding line segment $l_{v_i,v_i+1}$ is the first or last vertex seen of its corresponding $l_i$. If it is the first vertex seen, then $l_{v_i,v_i+1}$ is added to $E_s$ and is considered *open*. If it is the second vertex seen, or if it was initialized as open on $\vec{s}_0$, then it is removed from $E_s$ and is considered *closed*.

For each visited $v_i$, a check is made to see if $v_i$ is the root of $E_s$, signifying that $v_i$ is the closest vertex to $c$ with respect to $\vec{s}$. If $v_i$ has this property, then $v_i$ is considered visible and is added to the visibility graph $G_v$; otherwise it is obscured by some other edge appearing before it, with respect to $c$, and is ignored.

In this way every visible vertex with respect to $c$ is found and a visibility *tree* is generated for some $v_i$. This process is repeated $n$ times to build a complete visibility *graph*.

### B. Runtime

The asymptotic runtime of Lee's algorithm is analyzed in the following. The algorithm has four for-loops as well as the operations of the optimal sorting data structure. An outer for-loop iterates once through $n = |V| = 2|E|$ points, finding the visibility tree for every point.

Within the outer for loop, each end point pair for every $|E|$ line segments are inserted into the optimal sorting data structure $A$. Insert, delete, and find all take $O(\log n)$ time using a probabilistic structure such as a skip list, or balance binary search tree, such as an AVL tree. Thus, the insertion time for $A$ takes $2|E| \log n = O(n \log n)$.

Next, the sweep line edge list $E_s$ is initialized by checking all $|E|$ edges in $G$ for intersection with $\vec{s} = \hat{i}$. The edge list $E_s$ uses the same data structure as $A$, and thus all insertions take $O(\log n)$ time. In the worse case, all $|E|$ edges intersect $\vec{s}$ at some $\theta_i$, so the total runtime for this step is $O(|E| \log |E|)$. There are twice as many vertices as edges, and so because $|E| < |V|$, this runtime is asymptotically overshadowed by the previous step and can be ignored.

Finally, the sweeping for-loop begins its check of every $v_i \in V - c$ points. At each vertex a line is either inserted into or removed from $E_s$ once, requiring again $O(\log n)$ time for each operation. Thus the total running time for this step is also $(n-1) \log n = O(n \log n)$.

With these three steps and the outer for-loop combined, our summed running time is $O(n \cdot (.5n + n + n - 1) \cdot \log n)$, which asymptotically reduces to simply $O(n^2 \log n)$.

### C. Space Requirements

The space requirements of Lee's algorithm is analyzed in the following. The input graph $G$ requires $O(V + E)$ space, but we will assume that the input graph is not included in our space requirements.

Two optimum sorting data structures are needed in the algorithm - $A$ and $E_s$. D.T. Lee's original paper suggested an AVL tree be used; in our implementation we have used a skip list. Regardless, both use $O(n)$ space, totaling $2O(n)$. Because each $\theta_i$ is inserted into $A$ once, $n = |V|$ for datastructure $A$. In the worse case all edges are intersected by $\vec{s}$ at the same time, making $E_s$ have size $n = |E| = .5|V|$. No other memory is used in the algorithm, so the total overall space requirements, not including the input graph, is $O(1.5|V|)$, which is equivalent to simply $O(n)$.

### D. Analysis of Correctness

We begin our proof of correctness of Lee's $O(n^2 \log n)$ algorithm by defining the components of the algorithm.

**Definition 1.** A visibility graph $G_v = (V, E_v)$ is the set of all vertices $V$ in input graph $G$, and the set of edges $E_v$ which connects two vertices $v_i, v_j \in V$ without intersecting any obstacles, for all $v_i, v_j \in V$. We assume that two endpoints $i$ and $j$ of the same line are also considered visible. We restrict our obstacle set to the $|E|$ disjoint line segments, in any direction.

**Definition 2.** The line sweep vector $\vec{s}$ is a vector with its origin at some point $c \in V$ that rotates starting from direction $\hat{i}$ a full $2\pi$ radians.

**Definition 3.** A line segment $l_i$ is an obstacle in the 2 dimensional problem space defined between vertices $v_i$ and $v_{i-1}$.

**Definition 4.** The set $E_s$ contains all $l_i$ that intersect with $\vec{s}$ originating at point $c$, ordered in decreasing Euclidean distance from $c$ to $l_i$.

The above definitions we will now begin to prove the algorithms' correctness by observing the existence of optimal substructure of the visibility graph:

**Lemma 1.** *The visibility tree containing set of all edges $E_i$ connecting a single point $v_i$ to all other visible points $v_j \in V - v_i$, with respect to the single point $v_i$, is a sub-solution to finding the visibility graph of all points $v_j \in V$ in $G$.*

*Proof.* Assume the visibility tree $E_i$ for some $v_i$ is generated correctly every time. For $N = 1$ points, by the just stated assumption no other endpoint $v_j$ of a line segment is visible from that $N = 1$ point that is not already in the set $E_{i=1}$. For $N = 2$ points, following the same assumption, no point will be visible to those $N = 2$ points that is not already in the combined visibility tree set $E_{i=[1,2]\cdot}$ For $N = |V|$ points, it follows that no point in $V$ will be visible from any other point in $V$ that is not already in the set $E_{i=1\to N}$. In this case our above definition of a visibility graph is satisfied and our $N$ sub-solutions have resulted in correctly finding the visibility graph $G_v$ for all points $v_j \in V$. □

Lemma 1 assumed that the set of edges $E_i$ defining a visibility tree, for some vertex $i$, was generated correctly every time. We now prove our algorithm for this subproblem. We begin by defining the assumptions of our scan line method:

**Lemma 2.** *No more than one obstacle is visible at any time from a center point c with respect to the direction of a scan line vector $\vec{s}$ at any angle in Euclidean space.*

*Proof.* A Euclidean vector is defined as a geometric object that has a direction and length (or magnitude), but it does not itself have a width, or at least the width could be considered infinitely narrow. An infinitely narrow segment of a directional vector could not be obstructed by more than $n = 1$ geometric element at a time because otherwise the combined width of $n > 1$ geometric elements would have to be infinitely small. The combined width of two objects would be greater than infinitely small. Therefore, because obstacles in our problem space are assumed to be line segments, our lemma stands. □

**Corollary 1.** *The intersection point of $\vec{s}$ and $l_i \in E_s$ with the minimum Euclidean distance to c is the only line segment visible.*

*Proof.* Although $\vec{s}$ may cross several $l_i \in E_s$, by Lemma 2 we know only one point can have the visible property for a given $\vec{s}$, and by the definition of visibility we know it must be the first line segment it reaches. The first line segment a vector crosses from some point $c$ is the segment closest in Euclidean distance. □

With Lemma 2 and Corollary 1 we have proved the correctness of the results of scan line $\vec{s}$ at one $\theta_i$. We will now expand our proof to all $\theta \in 2\pi$ and our discretization method.

**Lemma 3.** *No change is made in the visibility of any line segment with respect to $\vec{s}$ except when $\vec{s}$ intersects an end point of some line segments.*

*Proof.* By contradiction. Assume the set $E_s$ correctly contains all line segments that intersect some vector $\vec{s}$ and assume $\vec{s}$ is at some $\theta_a$ that does not intersect any end points $\forall v_i \in V$. The only way to change the visibility of a line segment at $\vec{s}$ would be to remove the first line segment $E_s$ because this is the line segment closest to $c$. Suppose we removed this line segment, despite having no $v_i$ in intersection with $\vec{s}$. Then there exists a $l_i$ that intersects $\vec{s}$ and $E_s$ violates definition 4 defining what $E_s$ must contain, and by contradiction this lemma is proved. □

**Corollary 2.** *In the non-discrete angular space $\theta$ between 0 and $2\pi$, our scan line need only check $|V| - 1$ discrete steps where $\theta_i = angle(\vec{s}_{c\to v_i}, \hat{i})$,.*

*Proof.* Following from Lemma 3, no changes in visibility occur with respect to the rotation of $\vec{s}$ around $c$ except when $\vec{s}$ intersects an end point $v_i$, and there are only $|V| - 1$ endpoints around $c$ so it follows that only $|V| - 1$ angles of $\theta_i$ need to be checked. □

The utility of a scan line is now sufficiently proven by Lemma 3 and Corollary 2. The mechanism for tracking the removal and insertion of lines into $E_s$ is now proved:

**Lemma 4.** *A line segment $l_i$ with an end point $v_i$ in intersection with $\vec{s}$ must be added to the set $E_s$ if the opposite end point $v_{i-1}$ of $l_i$ has not previously been visited (the line was "closed"). Otherwise, if it has been previously visited, $l_i$ must be removed from the set $E_s$ (the line was "open").*

*Proof.* Following the stated assumption that $\vec{s}$ rotates in a counter clockwise direction, and recalling that at initialization all $l_i$ in intersection with $\vec{s}$ are added to $E_s$ and marked as open, it can be observed that Lemma 4 is required to maintain Definition 4, that $E_s$ must contain all line segments that intersect $\vec{s}$. □

Using Lemmas 1 to 4 and Corollaries 1 and 2 the following theorem is supported:

**Theorem 1.** *Given a set of n disjoint line segments in the Euclidean plane, the visibility graph can be constructed correctly in $O(n^2 \log n)$ time using the rotational sweep method in Lee's algorithm.*

## IV. RESULTS

### A. Implementation

The $O(n^2 \log n)$ algorithm was implemented in C++ and visualized/animated using the open source, cross-platform CImg graphics library. With the graphics library we were able to visually verify geometric results such as shown in Figure 3. The full source code is appended at the end of this paper and is available as an open source project online at `https://github.com/davetcoleman/visibility_graph`

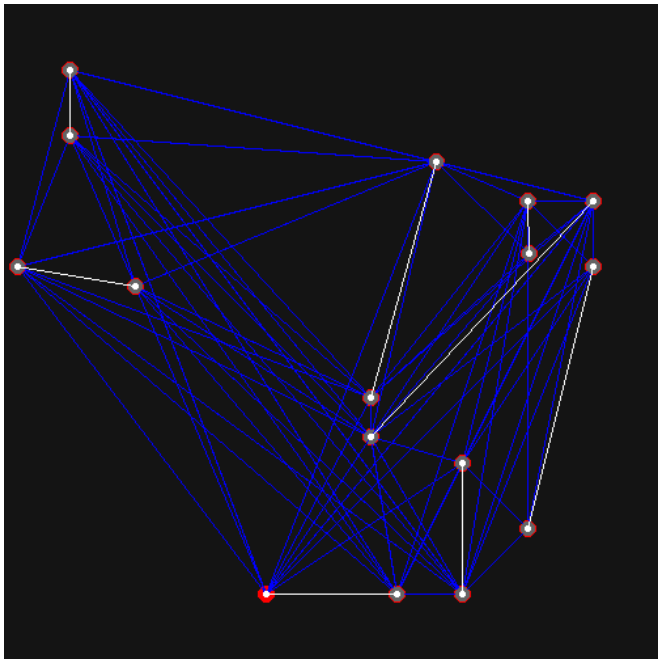

FIG. 3: Test input graph of 8 line segments (white) and the generated visibility graph of $E_v$ lines (blue)

### B. Skip Lists

D.T. Lee's original paper suggested an AVL tree data tree structure be used, but his paper was published before the invention of skip lists in 1989 by W. Pugh [12]. In this implementation we chose to use a skip list due to its average case performance and advantage in concurrent access and modifications. However, a unique property of our algorithm required special modification to the skip list such that the *key values of data already in the skip list are variable*. That is to say, the value of each element in the skip list changes as the scan line rotates around some center $c$.

The need for this property is motivated in Figure 4. In this example line $l_1$ is the first line segment that scan line $\vec{s}$ would visit and so it would be inserted into the skip list $E_s$ with the distance $d_1$ from its first endpoint to $c$. Next, the scan line would visit the first endpoint of $l_2$ and it would add $l_2$ to $E_s$ with distance $d_2$. Thus, $E_s$ would have as its first ordered line segment $l_1$, and for its seconds $l_2$. But by definition 4, $E_s$ should have as its first line segment the segment closest to $c$, and at scan line $\vec{s}'$ the closest intersecting line segment is now actually $l_2$. As it is now clear, in our current example the ordering of $E_s$ would be incorrect at location $\vec{s}'$ unless there was some way to update the value of $l_1$ to reflect its distance from $c$ *with respect to* $\theta_i$ *of* $\vec{s}'$.

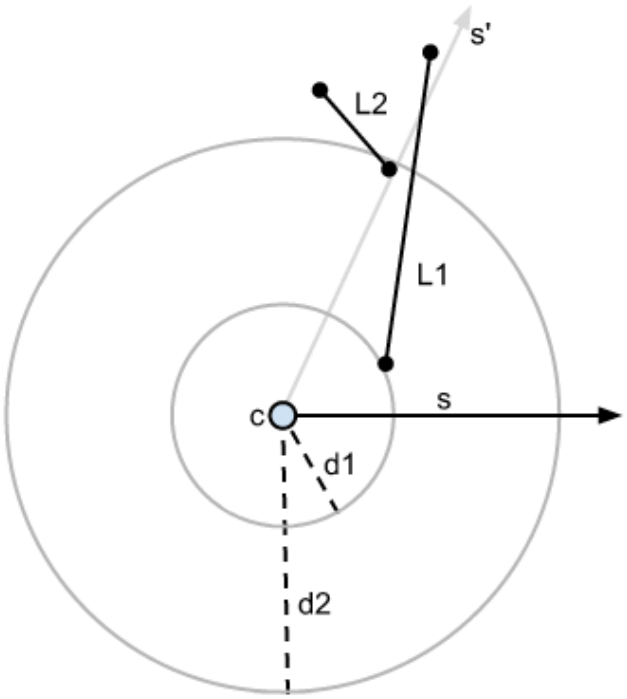


FIG. 4: Line $l_1$ originally was the closest point to $c$, but at scan line $\vec{s}'$ line $l_2$ covers $l_1$. This demonstrates the need for elements variable values in the skip list.

This might seem like an impossible property of a skip list, but in fact there is an additional property that states that the *ordering* of the items in the skip list are guaranteed not to change, just the values. In other words, although $l_1$ at angle $\theta_i$ has an intersection with $\vec{s}'$ that is a greater distance than that of $l_2$ from $c$, the ordering of $l_1$ in $E_s$ with respect to all other open line segments in $E_s$ would remain unchanged due to the assumption that no line segments can intersect in our problem space.

Therefore, in implementing the actual visibility graph, each line segment was represented as an object that could quickly re-calculate its intersection with $\vec{s}$ and then distance to $c$. This was accomplished by caching the slope $m$ and y-intercept $b$ at the initialization of the

*line* object, as well as caching the resultant distance $d$ from $c$ for every $\theta_i$ such that $d$ is only calculated once for every $theta_i$.

### C. Precision Errors

Another issue with our implementation was rounding errors that occurred when calculating the angles between two close points. This was especially problematic as we increased the number of line segments added to our finitely-sized graphics window. Sometimes two unique points would be added to the angle list $A$ with the same angle because of rounding errors, and the result was that some points were mistakenly added as visible.

Particularly problematic were perfectly horizontal and vertical lines. With vertical lines the slope $m$ would tend to infinity, but in this implementation it was faked with some very large number. In the same way, a horizontal line has a slope with an infinitely small $m$, and this again suffered from the limitations of our computer hardware.

### D. Numerical Time Usage

To calculate the numerical time usage of this algorithm, the source code was modified to automatically generate a set of $n$ line segments. To test the runtime with exponentially increasing problem space it was instrumented to generate approximately $n = 10^x$ line segments. However, to ensure a useful test set was generated without intersection, each line segment was constrained to a grid area. Within each line segment's grid, padding was added to allow more visibility between grids. Additionally, 4 shapes were used inside the grids: a horizontal, vertical, diagonal increasing and diagonal decreasing line segment. Which of the 4 was chosen was decided at random, such that every test was run on a problem set with a high probability of being unique. Because of the gridded nature of the problem space, in reality only $n_x = (\lfloor (10^n)^{1/2} \rfloor)^2$ line segments were added. An example of an automatically generated problem space is shown in Figure 5.

With this setup, the numerical time usage was measured by counting the number of atomic operations within both the algorithm and the skip lists. The algorithm was tested for $n = 10^1 \rightarrow 10^{3.5}$. At problem size $n = 10^4$ the algorithm crashed on both our laptop and on a node on the Janus super computer. This, however, is mostly due to some memory leaks that were problematic to patch.

The results of the atomic operations measurements are shown in Figure 6. Our data showed performance that was very tightly bound to a run time of $O(n^2 \log n)$. This

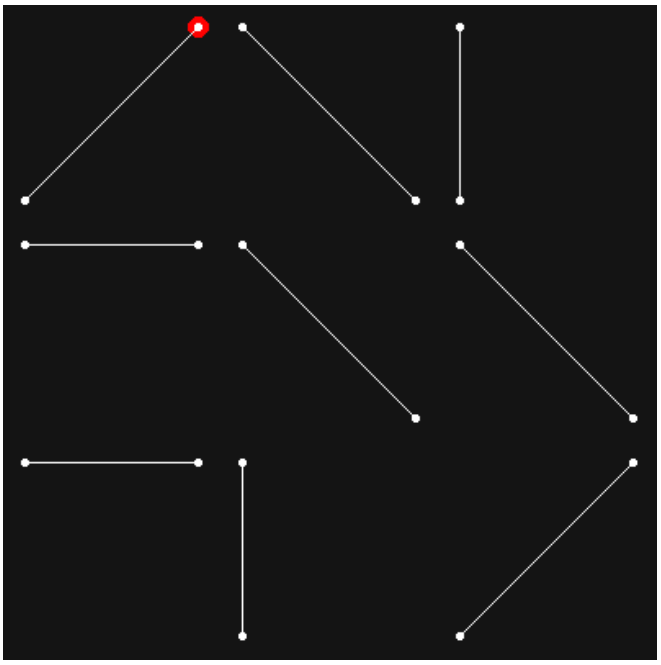

FIG. 5: An example generated problem space for $n_x = (\lfloor (10^n)^{1/2} \rfloor)^2$ line segments, with randomly chosen shapes.

run time is both the worst- and average-case for this algorithm because all points are always added to $A$ and $E$ and all points are always visited to generate their individual visibility tree.

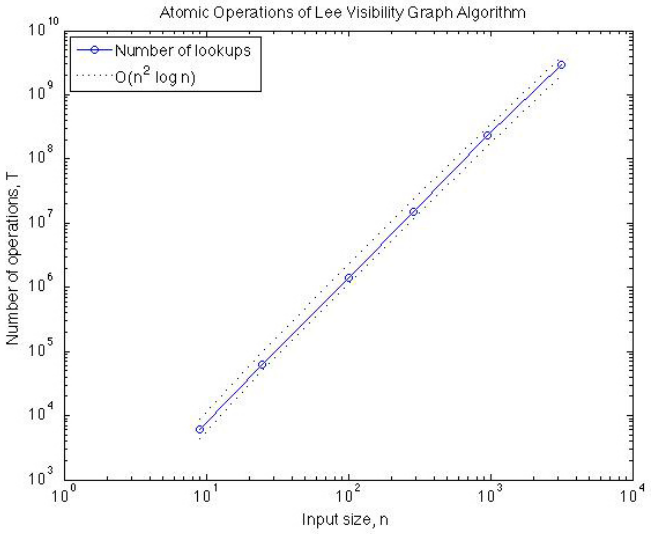


FIG. 6: Atomic operations of Lee's visibility graph algorithm for increasing number of line segments $n$.

Further visual results of the algorithm running for $n = 100$ line segments is shown in Figure 7 and for $n = 1000$ line segments in Figure 8.

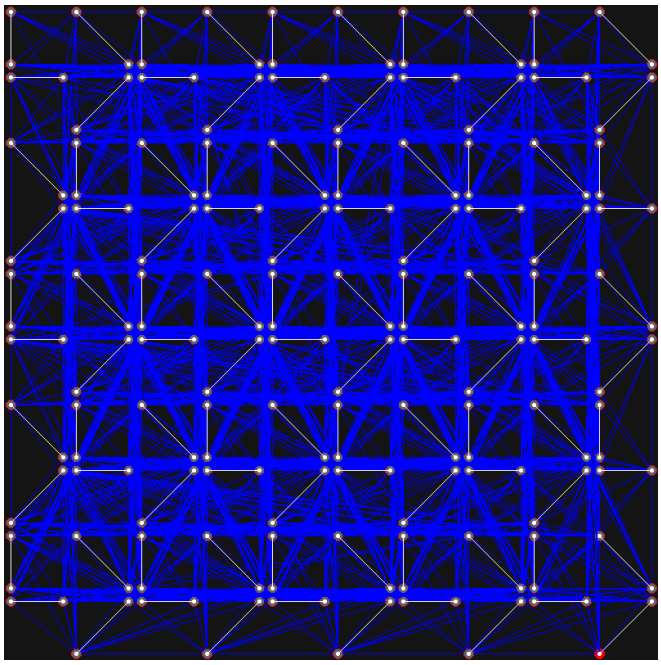

FIG. 7: Generated visibility graph for $n = 100$ line segments.

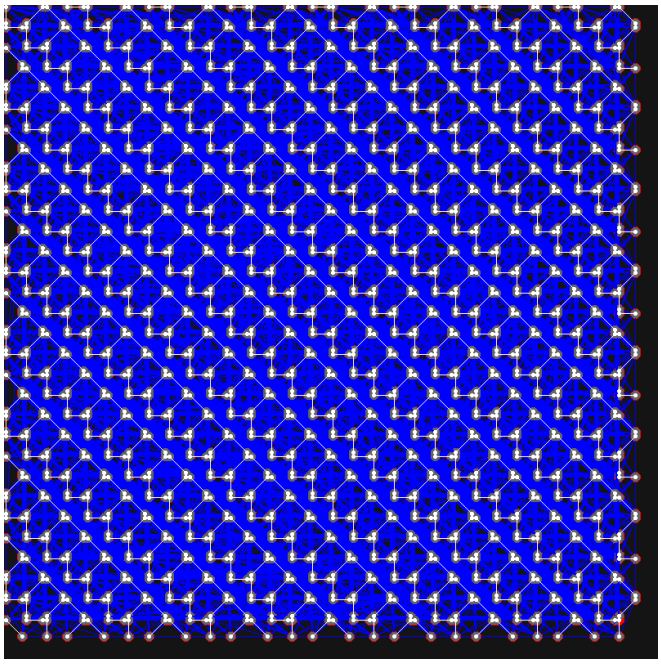

FIG. 8: Generated visibility graph for $n = 1000$ line segments.

### E. Numerical Space Usage

The numerical space usage of this algorithm was measured by tracking the maximum number of nodes from both skip lists combined at any point in the algorithm. Here, we define a node as a level in the skip list structure, such that a root with 3 levels is considered to use 3 atomic memory amounts. A memory counter was incremented for every new node created, and decremented for every node deleted. A secondary counter was used to track the maximum amount of memory used at any point in the algorithm's progress. The results are shown in Figure 9. As expected, the memory usage was on the order of $O(n)$.

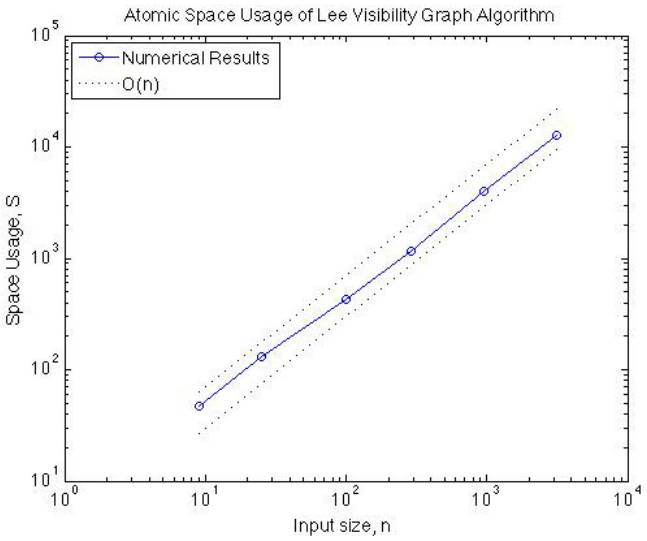


FIG. 9: Measured space usage of Lee's visibility graph algorithm. The upper and lower dotted line bounds are multiplied by constants of 3 and 7, respectively.

## V. CONCLUSIONS

There exists many additional optimization tweaks that could be applied to this algorithm. One such optimization is to limit the scan line rotation to only half the circle, from the observation that visibility between a pair is mutual. Other optimizations could be made in the geometric calculations such as studying the performance advantages between finding the distance using the line-of-sine method versus the intersection method and dealing with slopes of negative and positive infinity. Lastly, the dynamic-valued skip list structure discussed in section IV. B. requires a large number of distance calculations at every rotation of the scan line, and could be reduced by only checking a vertices' change in distance immediately around the skip lists' chosen insertion point for a new line segment. While simple in explanation, it would be complicated in required modification to the skip list algorithm.

In this paper we have explained, analyzed, proved and implemented D.T. Lee's 1978 visibility graph algorithm. As explained in the introduction, faster algorithms have been developed that run on $O(n^2)$ time and other

optimizations have been discovered for special case problems where certain geometric tricks can be utilized. However, basic applications such as the shortest path planning problem with no more than order $O(n^3)$ line segments has been shown in this paper section to be feasible with this algorithm and in our experiments have generated visibility graphs in seconds.

**Acknowledgments**

This paper was written for Aaron Clauset's Graduate Algorithms class at the University of Colorado Boulder.

[1] Burcin Cem Arabacioglu. Using fuzzy inference system for architectural space analysis. *Appl. Soft Comput.*, 10(3):926–937, June 2010.

[2] Herbert Edelsbrunner and Leonidas J. Guibas. Topologically sweeping an arrangement. *Journal of Computer and System Sciences*, 38(1):165 – 194, 1989.

[3] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[4] Subir Kumar Ghosh and David M. Mount. An output sensitive algorithm for computing visibility graphs. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 11–19, Washington, DC, USA, 1987. IEEE Computer Society.

[5] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100 –107, july 1968.

[6] Donald B. Johnson. A note on dijkstra's shortest path algorithm. *J. ACM*, 20(3):385–388, July 1973.

[7] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *Int. J. Rob. Res.*, 30(7):846–894, June 2011.

[8] John Kitzinger and Computer Engineering. The visibility graph among polygonal obstacles: a comparison of algorithms, 2003.

[9] Der-Tsai Lee. *Proximity and reachability in the plane.* PhD thesis, Champaign, IL, USA, 1978. AAI7913526.

[10] Jae-Ha Lee, Sung Yong Shin, and Kyung-Yong Chwa. Visibility-based pursuit-evasion in a polygonal room with a door. In *Proceedings of the fifteenth annual symposium on Computational geometry*, SCG '99, pages 281–290, New York, NY, USA, 1999. ACM.

[11] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570, October 1979.

[12] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[13] Emo Welzl. Constructing the visibility graph for n-line segments in o(n2) time. *Information Processing Letters*, 20(4):167 – 171, 1985.

## C++ Code For Visibility Graph Implementation

**Index:**



### vgraph.cpp

```
#include "CImg.h"// Include CImg library header.
#include <iostream>
#include "line.h"
#include "point.h"
#include "skiplist.h"
#include <cmath>

using namespace cimg_library;
using namespace std;

const unsigned char WHITE[] = { 255, 255, 255 };
const unsigned char GREY[] = { 100, 100, 100 };
const unsigned char BLACK[] = { 0, 0, 0 };
const unsigned char RED[] = { 255, 0, 0 };
const unsigned char GREEN[] = { 0, 255, 0 };
const unsigned char BLUE[] = { 0, 0, 255};
const int screen_size = 800;

//-----------------------------------------------------------------------------
//  Prototypes
//-----------------------------------------------------------------------------
void vgraph(double order);
double vectorsAngle( int x, int y, int basex, int basey);
double distance( Point * a, Point * b );

//-----------------------------------------------------------------------------
//  Main procedure
//-----------------------------------------------------------------------------
int main()
{
   cout << endl << endl << "Visibility Graph by Dave Coleman ---------------------- " << endl
       << endl;

   for( double order = 2; order < 3; order += 0.5 )
   {
      vgraph(order);
   }

   return EXIT_SUCCESS;
}

void vgraph(double order)
{
   // Variables ---------------------------------------------------------------

   // Atomic operation counter
   atomic = 0;
```

```
47
48  // Graphics:
49  bool visual = true;
50  bool live = true;
51
52  CImg<unsigned char> img(screen_size,screen_size,1,3,20);
53  CImgDisplay disp(img, "Visibility Graph");      // Display the modified image on the
        screen
54
55  // Line segments:
56  int size = pow(10.0, order);
57  int row_col = sqrt(size);
58  int seg = row_col * row_col;
59
60  // Coordinates:
61  double width = screen_size / row_col; // size of each grid box
62  double margin = 0.1 * width;  // padding inside each box
63  double top, bottom, left, right; // coordinates of box with padding
64
65  // Generate space for SEG number of lines
66  Line * segs[seg];
67
68  // Track what index we are on
69  int index = 0;
70
71  // Now generate seg line segments
72  for(int x = 0; x < row_col; ++x)
73  {
74     for(int y = 0; y < row_col; ++y)
75     {
76        top = y*width + margin;
77        bottom = (y+1)*width - margin;
78        left = x*width + margin;
79        right = (x+1)*width - margin;
80
81        // Create line segment in box of size width*width
82        // x1, y1, x2, y2
83        switch( rand() % 4 )
84        {
85        case 0: // verticle line
86           segs[index] = new Line( left, top, left, bottom );
87           break;
88        case 1: // horizontal line
89           segs[index] = new Line( left, top, right, top );
90           break;
91        case 2: // diagonal left to right
92           segs[index] = new Line( left, top, right, bottom );
93           break;
94        case 3:
95           segs[index] = new Line( left, bottom, right, top );
96           break;
97        }
98
99        index++;
100    }
101 }
102
103 //cout << "SEGS " << seg << " INDEX " << index << endl;
104
105 /*
106   Line segs[] =
107   {
108   Line(280,300,330,120), // 0 first
109   Line(450,150,280,330), // 1 second
110   Line(400,150,401,190), // 2 third, later
111   Line(400,400,450,200), // 3 far right
112   Line(350,350,350,450), // 4
113   Line(10,200,100,215),  // 5
```

```
    Line(50,50,50,100),     // 6
    Line(200,450,300,450)   // 7
    };
*/

// Reusable pointer locations
Line * l;
Point * p;

int center_id;
bool isPointA;

// Visit each vertex once and perform the visibility algorithm
    for(int outer = 0; outer < 2*seg; ++outer)
{
    ++atomic;

    // First or second number on each line?
    center_id = outer / 2;

    // Garbage Collect
    if( outer )
    {
        delete center;
        delete center_line;
    }

    //cout << "LINE ID: " << center_id << endl;
    if( ! (outer % 2) ) // is even
    {
        center = new Point( segs[center_id]->a->x, segs[center_id]->a->y );
        isPointA = true;
    }
    else // is even
    {
        center = new Point( segs[center_id]->b->x, segs[center_id]->b->y );
        isPointA = false;
    }

    // Center Line Calc:
    center_line = new Line( center->x, center->y, center->x+1, center->y );

    // Add pointers to all points back to parent line
    center->parentLine = segs[center_id];

    // Draw sweeper:
    //img.draw_line( center->x, center->y, center->x+200, center->y, RED);
    if(visual)
        img.draw_circle( center->x, center->y, 6, RED);

    /*cout << "LINE ID " << center_id << " ";
      if(isPointA)
      cout << "A" << endl;
      else
      cout << "B" << endl;
    */

    // Datastructures:
    skiplist <Point*> angleList;
    skiplist <Line*> edgeList;

    // Algorithm ------------------------------------------------------------

    // Draw segments and insert POINTS into skiplist ordered by ANGLE -------
    for(int i = 0; i < seg; ++i)
    {
        ++atomic;
        l = segs[i];
```

```
182
183      // Add pointers to all points back to parent line
184      l->a->parentLine = l;
185      l->b->parentLine = l;
186
187      // Reset visited flags
188      l->visited = false;
189      l->visitedStartPoint = false;
190
191      if(visual)
192         img.draw_line(l->a->x, l->a->y, l->b->x, l->b->y, WHITE);
193
194      if( !(i == center_id && isPointA) ) // point is not line A
195      {
196         if(visual)
197            img.draw_circle(l->a->x, l->a->y, 2, WHITE);
198
199         // Calculate the angle from center line:
200         l->a->theta = vectorsAngle( l->a->x, l->a->y, center->x, center->y );
201
202         // Sort the verticies:
203         angleList.add( l->a);
204
205            //cout << "Added A for line " << i << " theta " << l->a->theta << endl;
206         //cout << "POINT "; l->a->print(); cout << endl;
207      }
208
209      if( !(i == center_id && isPointA == false) ) // point is not line B
210      {
211         if(visual)
212            img.draw_circle(l->b->x, l->b->y, 2, WHITE);
213
214         // Calculate the angle from center line:
215         l->b->theta = vectorsAngle( l->b->x, l->b->y, center->x, center->y );
216
217         // Sort the verticies:
218         angleList.add( l->b);
219         //cout << "Added B for line " << i << " theta " << l->b->theta << endl;
220
221         //cout << "POINT "; l->b->print(); cout << endl;
222      }
223
224      //cout << endl;
225   }
226
227
228   // Test SkipList
229   //cout << "Angle List - points ordered CC from base line";
230   //angleList.printAll();
231
232
233   // Initialize Edge List Of Lines -----------------------------------------------
234   for(int i = 0; i < seg; ++i)
235   {
236      ++atomic;
237
238      l = segs[i]; // get next line to check
239
240      // check if the current line is connected to the center point
241      if( l->id == ((Line*)center->parentLine)->id )
242      {
243         // one center's line
244         //cout << "ONE CENTER'S LINE!!!" << endl;
245      }
246      else
247      {
248         // Check each line and see if it crosses scan line
249         double xi, yi;
```

```
            l->center_intercept( xi, yi ); // these are reference parameters
            // Now we know that xi,yi is on center line.
            // Next we check if X is between a & b. We know a.x > b.x, thus:
            if( l->a->x >= xi && l->b->x <= xi )
            {
               // check that xi > center->x
               if( xi >= center->x )
               {
                  // It does intersect
                  edgeList.add( l );
                  // Mark as opened, somewhere on line
                  l->visited = true;
                  // Visualize:
                  if(visual)
                     img.draw_line(l->a->x, l->a->y, l->b->x, l->b->y, GREEN);
               }
            }
         }
      }

      if(live)
         disp.display(img);

      //cout << "Edge List:";
      //edgeList.printAll();

         // Sweep ---------------------------------------------------------------

      //sleep(1);
      //usleep(500*1000);
         for(int i = 0; i < 2*seg - 1; ++i)
      {
         ++atomic;

         //cout << "\n\n\n ------------------ STARTING NEW SWEEP ------------------- \n\n\n";

         //cout << "SWEEP VERTEX " << i << endl;
         //if( i > 0 )
         // break;

         // take the first vertex in angular order
         p = angleList.pop();
         //cout << "Sweep at "; p->print();

         // Update the center_line to the sweep location and update m,b
         center_line->b = p;
         center_line->updateCalcs();

         // Update center point to contain theta between baseline and
         // current point, so that our line function can cache
         center->theta = p->theta;

         // decide what to do with it
         l = (Line*)p->parentLine; // cast it
         //cout << "\t"; l->print();

         // check if the current line is connected to the center point
         if( l->id == ((Line*)center->parentLine)->id )
         {
            // one center's line
            // ignore
         }
         else if( l->visited  ) // remove it from edgeList
         {
```

```
            //cout << "remove" << endl;

            if( ! l->visitedStartPoint )
            {
               l->visited = false; // allow this line to be visisted again for its start point
            }

            // check if its first in the edge list. if it is, its VISIBLE
            if( edgeList.isRoot( l->id ) )
            {
               //cout << "Drawing Line" << endl;

               if(visual)
                  img.draw_line( center->x, center->y, p->x, p->y, BLUE );
            }

            // remove
            //cout << "Value: " << l->value() << " " << l->id << endl;

            edgeList.remove( l->value(), l->id );

            if(visual)
               img.draw_line(l->a->x, l->a->y, l->b->x, l->b->y, WHITE);
         }
         else // add it to edge list
         {
            //cout << "add" << endl;
            l->visited = true; // mark it as having been visited somewhere
            l->visitedStartPoint = true; // mark it as having found the first vertex

            // Store distance of line from center
            l->dist = distance( p, center );

            edgeList.add( l );

            // check if its first in the edge list. if it is, its VISIBLE
            if( edgeList.isRoot( l->id ) )
            {
               //cout << "Drawing Line" << endl;
               if(visual)
                  img.draw_line( center->x, center->y, p->x, p->y, BLUE );
            }

            if(visual)
               img.draw_line(l->a->x, l->a->y, l->b->x, l->b->y, GREEN);
         }

         if(visual)
            img.draw_circle(p->x, p->y, 5, GREY);

         //debug
         //cout << "Edge List:";
         //edgeList.printAll();
         //angleList.printAll();
         //cout << endl << endl;

         if(live)
         {
            disp.display(img);
            //usleep(1*1000);
            //sleep(1);
         }
      }
      //cout << "breaking" << endl;
      //break;
      if(live)
      {
```

```
            //usleep(1*1000);
            disp.display(img);
        }
        //break;
        //img.fill(20);
        //cout << outer << endl;
    }

    if(visual)
    {
        // Redraw obstacle lines just for fun:
        for(int i = 0; i < seg; ++i)
        {
            l = segs[i];

            img.draw_line(l->a->x, l->a->y, l->b->x, l->b->y, WHITE);
            img.draw_circle(l->a->x, l->a->y, 2, WHITE);
            img.draw_circle(l->b->x, l->b->y, 2, WHITE);
        }
        disp.display(img);


        img.save("result.png"); // save the image
    }

    cout << seg << "," << atomic << endl;

    if(visual)
    {
        // Show window until user input:
        while (!disp.is_closed()) {
            if (disp.is_keyESC() )
                break;
            disp.wait();
        }
    }

    // Garabage collect
    //delete [] segs;
    //free(segs);
}

//-----------------------------------------------------------------------------
//  Calculate Angle Btw 2 Vectors
//-----------------------------------------------------------------------------
double vectorsAngle( int x, int y, int basex, int basey)
{
    // Convert input point x & y to be vectors relative to base point
    double x2 = double(x - basex);
    double y2 = double(y - basey);

    // Hard code scan line to point right:
    double x1 = sqrt( x2*x2 + y2*y2 ); // make it with ratio?
    double y1 = 0.0;

    //cout << "x1: " << x1 << " - y1: " << y1 << endl;
    //cout << "x2: " << x2 << " - y2: " << y2 << endl;

    double stuff = (  (x1*x2)+(y1*y2)   ) / (  sqrt(x1*x1+y1*y1)  *  sqrt(x2*x2+y2*y2) );
    //cout << "Stuff: " << stuff << endl;

    // Calculate angle:
    double result = acos( stuff );
    //cout << "Result: " << result << endl;

    // Now add PI if below middle line:
    if( y >= basey )
        result = 2*M_PI - result;
```

```
                                                                                453
   //cout << "Result: " << result*180/M_PI << " degrees" << endl;                454
                                                                                455
   return result;                                                               456
}                                                                               457
//-------------------------------------------------------------------------------458
//  Distance Btw 2 Points                                                       459
//-------------------------------------------------------------------------------460
double distance( Point * a, Point * b )                                         461
{                                                                               462
   return sqrt( pow(b->x - a->x, 2.0) + pow(b->y - a->y,2.0) );                 463
}                                                                               464
```

../visibility_graph/vgraph.cpp

**skiplist.h**

```
/* Skip List
   CSCI 5454 Algorithms
   Dave Coleman | david.t.coleman@colorado.edu

   2/2/2012

   Implementation of Skip Lists
*/

#include <math.h>
#include <iostream>
#include <cstdlib>
#include "node.h"
//#include "point.h"
using namespace std;

// ------------------------------------------------------------------------------
// Skip List Class
// ------------------------------------------------------------------------------
template <class T>
class skiplist{
      // used for testing
 public:
   int maxLevel;

 private:
   node<T> *root;

   // ------------------------------------------------------------------------------
   // Get Random Level
   // ------------------------------------------------------------------------------
   int getRandLevel()
   {
      int randResult = 1;
      int level = 0;
      while(randResult)
      {

         randResult = rand() % 2;
         if(randResult)
         {
            ++level;
         }

         if(level > maxLevel)
         {
            randResult = 0; // to end the while loop
         }
      }
      return level;

   }
   // ------------------------------------------------------------------------------
   // Create New Node
   // ------------------------------------------------------------------------------
   node<T>* createNode( int level, int height, T data)
   {
      // Check if we are below level 0
      if(level < 0)
      {
         return NULL;
      }
      else // make a new node below
      {
         node<T> *newNode = new node<T>();
         newNode->level = level;
```

```
        newNode->next = NULL;
        newNode->below = createNode( level - 1, height, data);
        newNode->height = height;
        newNode->data = data;
        return newNode;
      }
    }

public:

  // Constructor:
  skiplist()
  {
      root = NULL;
      maxLevel = 0;

      srand ( time(NULL) ); // seed the random generator
  }
  // ------------------------------------------------------------------------
  // ADD
  // ------------------------------------------------------------------------
  void add( T data)
  {
      //cout << "ADD: ";
      //data.print();

      // Special Cases ---------------------------------------------------------

      if(!root) // no root has been established yet
      {
          root = createNode( 0, 0, data);
          return;
      }

      if( root->data->value() > data->value() ) // new value goes before root
      {
         T temp_data = root->data;
         node<T> *n = root;

         for(int l = maxLevel; l >= 0; --l)
         {
            atomic += 1;
            // change the root to the new value
            n->data = data;
            n = n->below;
         }
         data = temp_data;
      }

      // Regular insert after root ---------------------------------------------

      // Determine what level this new node will be at
      int level = getRandLevel();

      // If new node is at whole new level, go ahead and update root node to be higher
      if(level > maxLevel)
      {
          maxLevel ++;
          node<T> *newRoot = new node<T>();
          newRoot->data = root->data;
          newRoot->next = NULL;
          newRoot->below = root;
          newRoot->level = maxLevel;
          root = newRoot;
      }

      // Create the new node
      node<T> *newNode = createNode( level, level, data);
```

```
    // Now add the node to the list
    node<T> *i = root;

    // Loop down through all levels
    for(int l = maxLevel; l >= 0; --l)
    {
        atomic += 1;
        // move forward until we hit a value greater than ours
        while( i->next != NULL )
        {
            atomic += 1;
            if( i->next->data->value() > data->value() ) // insert before i.next
            {
                break;
            }
            i = i->next;
        }

        // Check if we should add a pointer at this level
        if( l <= level )
        {
            newNode->next = i->next;
            i->next = newNode;

            // Now move the new node pointer one level down:
            newNode = newNode->below;
        }

        // Always move the i node pointer one level down:
        i = i->below;
    }

}
// ------------------------------------------------------------------------------------------
// Find
// ------------------------------------------------------------------------------------------
bool find(double x)
{
    node<T> *i = root;

    // Special case: skip list is empty
    if( !root )
    {
        return false;
    }

    // Special case: check root
    if( root->data->value() == x)
    {
        return true;
    }

    for(int l = maxLevel; l >= 0; --l)
    {
        atomic += 1;
        // move forward until we hit a value greater than ours
        while( i->next != NULL )
        {
            atomic += 1;
            if(  i->next ->data->value() > x ) // x is not found on this level
            {
                break;
            }
            else if(  i->next->data->value() == x ) // bingo!
            {
                return true;
            }
```

```
            i = i->next;
        }

        // Always move the i node<T> pointer one level down:
        i = i->below;
    }

    return false;
}
// ------------------------------------------------------------------------
// REMOVE
// the id is to confirm the correct node, just in case x is not unique
// ------------------------------------------------------------------------
bool remove(double x, int id)
{
    node<T> *i = root;

    // Special case: remove root ------------------------------------------
    if( root->data->value() == x && root->data->id == id)
    {
        // Get level 0 of root
        for(int l = root->level; l > 0; --l)
        {
            atomic += 1;
            //cout << "Level " << l << endl;
            i = i->below;
        }

        // Check if there are any more nodes
        if( !i->next ) // the skip list is empty
        {
            root = NULL;
            maxLevel = 0;

            return true;
        }

        // Change value of root to next node
        node<T> *n = root;
        node<T> *nextNode = i->next;

        for(int l = maxLevel; l >= 0; --l)
        {
            atomic += 1;
            // change the root to the new value
            n->data = nextNode->data;

            // update next pointer if the next next exists
            if( n->next )
            {
                n->next = n->next->next;
            }

            // Move down to next level
                n = n->below;
        }

        return true;
    }

    // Normal case: remove after root -------------------------------------
    bool found = false;

    for(int l = maxLevel; l >= 0; --l)
    {
        atomic += 1;
        // move forward until we hit a value greater than ours
```

```
        while( i->next != NULL )
        {
            atomic += 1;
            // remove this one, confirmed by id
             if( i->next->data->value() == x && i->next->data->id == id )
            {
                found = true;

                // pass through the pointer if exists
                if( i->next )
                {
                    i->next = i->next->next;
                }
                else
                {
                    i->next = NULL;
                }
                break;
            }
            else if( i->next->data->value() > x ) // x is not found on this level
            {
                break;
            }

            i = i->next;
        }

        // Always move the i node pointer one level down:
        i = i->below;
    }

        return found;
}
// ------------------------------------------------------------------------------------
// POP FROM FRONT
// ------------------------------------------------------------------------------------
T pop()
{
    node<T> *i = root;

    // Store the first item on the list that we want to later return
    T result = root->data;
    /*
      cout << "POP WITH VALUE: " << root->value << " - ";
      result.print();
      cout << endl;
    */

    // Check if skip list is empty
    if( !root )
    {
        cout << "An error has occured: skip list is empty";
        exit(1);
    }

    // Get level 0 of root
    for(int l = root->level; l > 0; --l)
    {
        atomic += 1;
        i = i->below;
    }

    // Check if there are any more nodes
    if( !i->next ) // the skip list is empty
    {
        root = NULL;
        maxLevel = 0;
```

```
        return result;
    }

    // Change value of root to next node
    node<T> *n = root;
    node<T> *nextNode = i->next;

    for(int l = maxLevel; l >= 0; --l)
    {
        atomic += 1;
        // change the root to the new value
        n->data = nextNode->data;

        // update next pointer if the next next exists
        if( n->next )
        {
            n->next = n->next->next;
        }

        // Move down to next level
        n = n->below;

    }

    return result;
}
// ------------------------------------------------------------------------
// Is Root
// ------------------------------------------------------------------------
bool isRoot(int id)
{

    if( !root ) // there is no root!
    {
        std::cout << "there is no root!" << std::endl;
        return false;
    }
     return (root->data->id == id);
}
// ------------------------------------------------------------------------
// PRINT ALL
// ------------------------------------------------------------------------
void printAll()
{
    std::cout << std::endl << "LIST ------------------------------------" << std::endl;

    // Special case: skiplist is empty
    if( !root )
    {
        std::cout << "------------------------------------" << std::endl;
        return;
    }

    node<T> i = *root;

    // Get level 0 of root
    for(int l = root->level; l > 0; --l)
    {
        //cout << "Level " << l << " - ";
        //i.data.print();
        //cout << endl;
        i = *(i.below);
    }
    //std::cout << "we are on level " << i.level << std::endl;

    // Hack: update root 0 level with maxLevel count, because we don't update this
    // when growing root level size
    i.height = maxLevel;
```

```
                                                                                                                    407
      int counter = 0;                                                                                              408
      bool done = false;                                                                                            409
                                                                                                                    410
      while(!done)                                                                                                  411
      {                                                                                                             412
         std::cout << counter;                                                                                      413
                                                                                                                    414
         for(int l = i.height; l >= 0; --l)                                                                         415
         {                                                                                                          416
            std::cout << " | ";                                                                                     417
         }                                                                                                          418
         std::cout << " " << i.data->value() << " - ";                                                              419
         i.data->print();                                                                                           420
                                                                                                                    421
         counter ++;                                                                                                422
                                                                                                                    423
         if( i.next )                                                                                               424
         {                                                                                                          425
            node<T> *ii = i.next;                                                                                   426
            i = *ii;                                                                                                427
         }                                                                                                          428
         else                                                                                                       429
         {                                                                                                          430
            done = true;                                                                                            431
         }                                                                                                          432
      }                                                                                                             433
                                                                                                                    434
      std::cout << "----------------------------------------" << std::endl << std::endl;                           435
   }                                                                                                                436
};                                                                                                                  437
```

../visibility_graph/skiplist.h

**node.h**

```
#include "line.h"

//
// Node Class
//

template <class T>
class node{
 public:
   node *below; // node below in tower
   node *next; // next node in skip list
   int level; // level of this current node
   int height; // full number of levels in tower
    T data;
};
```

../visibility_graph/node.h

**line.h**

```
#ifndef LINE_H_INCLUDED
#define LINE_H_INCLUDED

#include <iostream>
#include "point.h"
#include "geometry.h"
#include <cmath>

class Line: public Geometry
{
 public:
   Point * a;
    Point * b;
   bool visitedStartPoint; // has the base/sweep line crossed at least one of
                  // the verticies?
   bool visited;  // has the sweep line been on the line (as in, maybe it was init on it)

   int id;
   double dist; // distance from center
   double theta_cache; // used for deciding if the dist cache needs to be refreshed
   double m; // slope of line
   double y_intercept; // y-intercept of line

   Line();
   Line(int _x1, int _y1, int _x2, int _y2);
   ~Line();
   virtual void print();
    virtual double value();

   void updateCalcs();
   void distance();
   void center_intercept(double &xi, double &yi);
};


// This global needs to be visible to classes:
extern Point * center;
extern Line * center_line;
extern double atomic;

#endif
```

../visibility_graph/line.h

**line.cpp**

```
#include "line.h"

Point * center;
Line * center_line;
double atomic;

using namespace std;

Line::Line()
{
   cout << "You are calling the function wrong";
   exit(0);
}
Line::Line(int x1, int y1, int x2, int y2)
{
   // Order a and b such that a.x > b.x
   if( x1 > x2 )
   {
      a = new Point(x1, y1);
      b = new Point(x2, y2);
   }
```

```
    else
    {
        b = new Point(x1, y1);
        a = new Point(x2, y2);
    }

    // Change ID
    static int id_counter = 0;
    id = id_counter++;

    // Keep track of its visited history
    visited = false;
    visitedStartPoint = false;

    // cout << "LINE" << endl;
    updateCalcs();

    // cout << "LINE" << endl;

    // Used for checking if we need to refresh our distance amount
    theta_cache = 3*M_PI; // some angle bigger than 2PI, aka INF
    //distance();

    // cout << "END LINE \n" << endl;
}
Line::~Line()
{
    //delete a;
    //delete b;
}
void Line::print()
{
    cout << "Line: x1: " << a->x << " y1: " << a->y << " x2: " << b->x
         << " y2: " << b->y << "\t ID: " << id << endl;
}
double Line::value()
{
    // calculate distance from midpoint at a given theta,
    // with resepct to the baseline

    if( theta_cache != center->theta ) // check if our cached version is still fresh enough
    {
        //cout << "Recalculaing distance for line " << id << endl;
        distance();
    }

    return dist;
}
void Line::updateCalcs()
{
    // Find Slope and y-intercept of this line for future distance calculations
    double denom = (b->x - a->x);
    if( denom == 0 )
    {
        //cout << "This program does not support perfectly verticle lines." << endl;
        //exit(0);

        // Perturb:
        //    b->x = b->x + 1;
        denom = 0.000000001; //(b->x - a->x);
    }
    m = (b->y - a->y)/denom;

    // cout << m << " M " << endl;

    y_intercept = a->y - m*a->x;
    // cout << y_intercept << " m " << endl;
}
```

```
void Line::distance()
{
   // First find the intesection of this line and the sweep line:
   double xi;
   double yi;
   center_intercept( xi, yi );

   //cout << "The intercept is x: " << xi << " y: " << yi << endl;
   //cout << "M: " << m << " b: " << y_intercept << endl;

   // Now find the distance between these two lines:
   dist = sqrt( pow(center->x - xi, 2.0) + pow(center->y - yi,2.0) );

   //cout << "Distance: " << dist << endl << endl;
   theta_cache = center->theta;
}

void Line::center_intercept(double &xi, double &yi)
{
   xi = double( y_intercept - center_line->y_intercept ) / double( center_line->m - m );
   yi = m*xi + y_intercept;
}
```

../visibility_graph/line.cpp

**point.h**

```
#ifndef POINT_H_INCLUDED
#define POINT_H_INCLUDED

#include <iostream>
#include "geometry.h"

class Point: public Geometry
{
 public:
   int x;
   int y;
   void* parentLine;
   int id; // for removing, comparing, etc
   double theta; // anglular amount from base line

   Point();
   Point(int _x1, int _y1);

   virtual void print();
   virtual double value();
};

#endif
```

../visibility_graph/point.h

**point.cpp**

```
#include "point.h"


Point::Point()
{
   static int id_counter = 0;
   id = id_counter++;
}
Point::Point(int _x1, int _y1)
{
   x = _x1;
   y = _y1;
   Point();
}
void Point::print()
{
   std::cout << "Point x: " << x << " y: " << y << " \t ID: " << id << std::endl;
}
double Point::value()
{
   // this is the angular distance from the base line
   // for point.cpp, we just cache the initial calculation
   return theta;
}
```

../visibility_graph/point.cpp

**geometry.h**

```
#ifndef GEOMETRY_H_INCLUDED
#define GEOMETRY_H_INCLUDED

class Geometry
{
 public:
   // int id; // for removing, comparing, etc

   virtual void print() = 0;
   virtual double value() = 0;

};

#endif
```

../visibility_graph/geometry.h

**Matlab plot.m Used For Generating Plots**

```
clear
clc

data = csvread('data.cvs')

n = data(:,1)
logger1 = 50.*(n.^2) .* log ( n );
logger2 = 25.*(n.^2) .* log ( n );

loglog( data(:,1),data(:,2),'bo-', ...
    data(:,1),logger1,'k:',data(:,1),logger2,'k:')


set(gca,'FontSize',14)

legend('Number of lookups','O(n^2 log n)','Location','NorthWest')
xlabel('Input size, n')
ylabel('Number of operations, T')
title('Atomic Operations of Lee Visibility Graph Algorithm');
```

../visibility_graph/plot.m

**Runtime Data Restuls**

```
9,6026
25,62585
100,1.42313e+06
289,1.47751e+07
961,2.34027e+08
3136,2.914e+09
```

../visibility_graph/data.cvs