An $E \log E$ Line Crossing Algorithm for Levelled Graphs

Vance Waddle and Ashok Malhotra

IBM Thomas J. Watson Research Center,
P.O.Box 704
Yorktown Heights, NY 10598
{waddle,petsa}@us.ibm.com

Abstract. Counting the number of crossings between straightline segments is an important problem in several areas of Computer Science. It is also a performance bottleneck for Sugiyama-style layout algorithms. This paper describes an algorithm for leveled graphs, based on the classification of edges that is $O(e \text{ log } e)$ where e is the number of edges. This improves on the best algorithm in the literature which is $O(e^{1.695}$ log $e)$. The improved crossing algorithm enabled an implementation of a Sugiyama-style algorithm to lay out graphs of tens of thousands of nodes in a few seconds on current hardware.

1 Introduction

The design of algorithms to compute the intersections of straight line segments, and counting their intersections, is a well known problem in computational geometry. A classic use for these algorithms in computer graphics is to determine a line's intersection with a viewing area's "clipping" bounds. There can be e^2 intersections on e line segments, so an algorithm that examines intersections is $O(e^2)$.

A more recent area concerned with line intersections is graph layout for user interfaces [4, 6, 12, 14]. These algorithms lay out a directed graph according to various visual aesthetics to improve the graph's readability as part of a user interface displaying, for example, a flow-chart, a class hierarchy, or a database schema. Two of the most common aesthetics are 1) *leveled graph* - display the nodes in the graph in levels, where all the nodes on a level have the same vertical coordinate and 2) reduce the number of edge intersections to make it easier to visually follow the edges between nodes. The second aesthetic implies that counting the number of line intersections is a fundamental operation in these layout algorithms, since they must compare the number of edge intersections in alternative layouts to determine the better layout. Worse, since Sugiyama-style crossing reduction algorithms [14] are based on sorting, these algorithms could be $O(n \text{ log } n)$ in the number of nodes, except for the intersection count. We have discovered by experience that counting line intersections is a performance bottleneck for the entire algorithm for large graphs.

Sugiyama-style crossing reduction algorithms perform well for "small" graphs of up to a few hundred nodes, even with a simple $O(e^2)$ algorithm for counting the edge crossings. However, recent work [8, 10] has begun to consider the problem of handling graphs of tens of thousands, or even hundreds of thousands of nodes. Munzer [10] argues that more general, non-tree layout algorithms are too slow to process this class of graphs.

In the remainder of the paper, we first discuss previous algorithms, and their relationship to our work. We then describe the layout problems which motivate our algorithm, and show how to compute the number of intersections in a leveled graph by using the graph's geometry. Next, we describe our line intersection algorithm, and prove bounds on its worse case performance. We describe its performance when implemented and run on a sequence of test cases and, finally, give our conclusions.

2 Previous Work

The simplest algorithm for determining the number of intersections of a collection of line segments is to test each pair of line segments for intersection. Where e is the number of line segments, this algorithm is $O(e^2)$. While there can be e^2 intersections in the worst case, there are usually far fewer intersections than this, leaving this algorithm a poor solution in practice. When implemented on a leveled graph, only line segments between adjacent levels are tested for intersection. This algorithm works better than might be expected for small graphs, both because the intersection test is cheap (comparing the positions of the end nodes), and because the nodes tend to be spread among the levels, which divides the problem into smaller pieces.

The main cost of the simple crossing algorithm lies in repeatedly testing segments for intersections as they occur in different segment pairs. Sweep algorithms [1, 3] avoid re-examining edges by performing a single sweep across the plane. Segments are tested for intersection only when they overlap the sweep line. These algorithms list the intersections of straight line segments, and are $O(e \log e + k)$, where e is the number of line segments and k the number of intersections. They are $O(e^2)$ in the number of edges, since there can be that many intersections in the worst case. The best reported algorithm for just counting intersections [2] is order $O(e^{1.695} \log e)$ and is based on an algorithm for partitioning the plane into regions by [5].

Graph layout algorithms only need a count of the number of intersections, not a list of the intersecting line segments. Thus, they do not require examining individual pairs of line segments to determine intersections. In leveled graphs, the levels decompose the graph into disjoint regions (although not the regions required by [2]), so the crossing algorithm does not require the overhead of decomposing the graph into regions if it utilizes the level structure properly. By using the geometry of a leveled graph to classify edges based on their relationship to the current sweep position, we create a simple algorithm to compute the number of line intersections that is $O(e \log e)$, where e is the number of edges.

3 Leveled Graphs and Layout

In this section we briefly describe leveled graphs and the relationship between leveled graphs and the layout algorithms that create them that motivate our intersection algorithm.

A *leveled graph* is a directed graph in which each node in the graph has been assigned to a single *level*, an ordered set of nodes, from an ordered set of levels. When the level is displayed horizontally, all the nodes in the level have the same vertical coordinate. Further, the end nodes of each edge are placed on adjacent levels in the graph, so that each edge spans a single level. If, after nodes are assigned to levels, there are "long edges" for which the source and target end nodes are not on adjacent levels, these edges are "shortened" by adding "virtual" intermediate nodes so that each edge in the resulting graph spans a single pair of adjacent levels. For a complete description of leveled graphs and their layout algorithms, see [4].

Figure 1 shows an example leveled graph consisting of nodes **A** through **H**. Node **A** is on the first (top) level, **B** and **C** on the second level, etc. (The graph is displayed as it might be after the layout process is completed, and coordinates have been assigned to all the nodes in the graph.) Virtual nodes have been added between nodes **A** and **D**, and nodes **C** and **F**. Notice the paths for nodes **C** and **H**, and **D** and **G** allow some freedom in their assignment to a level. We emphasized this in the figure by placing **C** and **H** on the highest possible level, and **D** and **G** on the lowest.

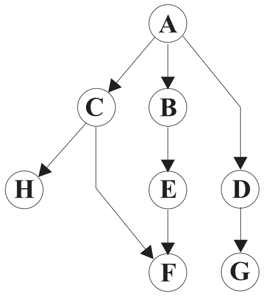

Figure 1. A leveled graph

Sugiyama-style layout algorithms [14] separate the layout problem into three phases. The first phase produces a leveled graph in which each node is assigned to a level, with child nodes appearing on a level beneath their parent. A typical scheme to assign nodes to levels is to use a variation of a topological sort. Virtual nodes are added to shorten long edges during this phase. This procedure only partially constrains the assignment of nodes to levels. Some algorithms add additional processing to improve the distribution of nodes to levels. For example, the DAG system [6] uses integer programming to reduce total edge lengths in the graph.

The second phase re-arranges nodes within their assigned levels to reduce the number of crossings. The algorithms [4] use heuristics, typically based on sorting, to reduce the number of crossings, since it is known that minimizing intersections is an NP complete problem [7]. It is during this phase that counting the number of intersections becomes a critical problem. After crossings have been reduced by the second phase, the nodes have a relative position within the level, but no coordinates. The third phase is a positioning phase that assigns coordinates to levels, and spaces the nodes within their levels.

4 Leveled Graphs and Intersections

In this section we describe how to efficiently compute the number of edge intersections in a leveled graph. We show how to classify edges into types based on the current sweep position and use the classification to compute the number of edge intersections. We then prove that all intersections are captured by our procedure.

Our algorithm makes a single sweep across a level, and uses the geometry of a leveled graph to remove the test for intersection by classifying the edges into types based on the position of their end-nodes with respect to the current sweep position. The number of intersections is then computed using the geometry of the leveled graph, and the position of the edge's end nodes. It follows from the geometry of the graph and the definition of the edge types that the edges of certain types must intersect. We then prove that all intersections are counted at some stage of the sweep via this classification.

Our discussion focuses on a pair of adjacent levels, since all intersections in a leveled graph appear between adjacent levels. (Note: unlike the classical segment crossing problem in computational geometry, edges with a common end node do not count as an intersection). We assume that nodes are numbered on each level, from left to right, starting with 1. We denote an edge e between a node a in the top level and b in the bottom level as the tuple $<a,b>$. In the following, when we compare nodes with arithmetic equalities and inequalities we are comparing the nodes' ordinal positions within the level. Also, note that since the algorithm computes intersections by making a sweep (from left to right) of the nodes in the pair of levels being examined, the sweep position is always a pair of nodes $<a,b>$ with equal positions, i.e., $a = b$.

Definition of edge types: An edge between nodes $<P_i, P_j>$ is classified as being a *top level* edge if $P_i < P_j$, a *bottom level* edge if $P_i > P_j$, and *pair edge* if $P_i = P_j$. If the current sweep position is S, each top-level and bottom-level edge is further classified as

1) *Right* edge if $P_i = S$, and $P_i < P_j$, or if $P_j = S$, and $P_j < P_i$,

2) *Trailing* edge if $P_i < S$ and $S < P_j$, or if $P_j < S$ and $S < P_i$.

Note that an edge's type is a function of the current sweep position, and changes along with the sweep position.

We illustrate the edge types through the leveled graph shown in Figure 2. Nodes in the top level are nodes a through e, and the bottom level nodes are u through z. Suppose the sweep is at the pair of nodes $<c,y>$. Top level edges are: $<c,y>$ is the graph's single pair edge, $<c,v>$ and $<c,u>$ are right edges, and $<a,v>$, $<a,u>$, $<b,v>$, and $<b,u>$ are trailing edges. Bottom level edges $<y,d>$ and $<y,e>$ are right edges, and $<z,d>$, $<z,e>$, $<x,d>$, and $<x,e>$ are trailing edges.

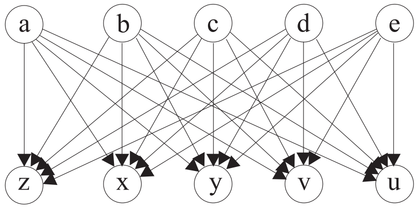

Figure 2. Adjacent levels in graph.

Given the geometry, it is clear that top level trailing edges must cross with bottom right edges, that top and bottom level right edges must cross, etc. The number of these crossings can be calculated simply from knowing the numbers of each type of edge. Let R_T be the number of top level right edges, R_B be the number of bottom level right edges, P the number of pair edges, Tr_T the number of top trailing edges, etc., the number of crossings due to these sources is

$$Tr_T * R_B + Tr_B * (R_T + P) + R_T * R_B$$

We call these crossings *alternate level crossings* because they are produced by edges from the top level intersecting with edges from the bottom level. Unfortunately, there is an additional source of crossings shown in our example graph. Right edges and trailing edges from the same level can intersect, as shown by the intersection of $<c,v>$ with $<a,u>$ and $<b,u>$. We call these *same level crossings.* These intersections are more difficult to determine than our edge types, since not all right and trailing edges from the same level cross, e.g., $<c,u>$ and $<a,u>$ do not cross.

By examining the geometry, we note that a same-level crossing occurs between a right top level edge, $<t_1,b_1>$, and a top level trailing edge, $<t_2,b_2>$, iff $b_1 < b_2$. So the number of crossings due to a right edge $<t_1,b_1>$ is the number of trailing edges to the nodes at positions $b_1 + 1$ through b_{LAST}, where b_{LAST} is the position of the last node on the bottom level. Similar considerations hold for same-level crossings from the bottom level. Note that we count crossings between pair edges and top trailing edges as same-level crossings.

If we let $Same_T$ and $Same_B$ be the number of top and bottom same level crossings at a sweep position, then the number of crossings at a given sweep position is

$$Tr_T * R_B + Tr_B * (R_T + P) + R_T * R_B + Same_T + Same_B$$

This leaves us with two problems: 1) proving that the above expression covers all the intersections, and 2) finding an efficient scheme for answering the range query required to determine the number of same level intersections due to a given right edge.

We prove that all intersections are captured by starting with an intersection between two edges, and working backward to show at which stage of the sweep it was considered, and which type of intersection under which it was classified. The proof follows by simply elaborating the cases under the trichotomy law.

Proof: All intersections are captured either as an intersection between different edge types, or as a same-level intersection between a level's right and trailing edges. Consider an intersection between distinct edges *e1* to nodes <*a,b*> and *e2* to nodes <*c,d*>.

- Cases for $a < b$: If $a < b$, then $e1$ is a top edge.
  - $c > d$: In this case, $e2$ is a bottom edge. If $c \leq a$, then $e1$ and $e2$ do not intersect. If $c > a$ and there is an intersection, then it must be that $d < b$. If $a = d$ then $e1$ and $e2$ intersect as right edges (covered); if $a < d$ then $e1$ is a top trailing edge and the intersection occurs when $e2$ is a bottom right edge (covered); if $a > d$ then e1 is right edge and the intersection occurs when $e2$ is a trailing edge (covered).
  - $c = d$: Edge $e2$ is a pair edge. If $a < c$ and $c < b$, then $e1$ is a trailing edge, and intersects $e2$ when the sweep position is c (covered). If $a \leq c$ or $c \geq b$, there is no intersection.
  - $c < d$: Edge $e2$ is a top edge, and any intersection occurs as a same level intersection between $e1$ and $e2$. For an intersection to occur, either $a < c$ and $b > d$ or $a > c$ and $b < d$. If $a < c$ and $b > d$, then $e1$ is a trailing edge, and intersects with $e2$ when c becomes the sweep position, with $e2$ a right edge (covered). Similarly, if $a > c$ and $b < d$, then $e1$ is a right edge when a becomes the sweep position, and $e2$ is a trailing edge (covered).
- Cases for $a = b$: If $a = b$, then $e1$ is a pair edge.
  - $c > d$: $e2$ is a bottom edge. If $c \leq a$, or $b \leq d$, there is no intersection. If $c > a$ and $d < b$, then $e1$ intersects $e2$ when the sweep position is a, and e2 is a trailing edge (covered).
  - $c = d$: Edge $e1 = e2$, and since they were assumed to be distinct edges, this does not occur.
  - $c < d$: If $c \geq a$ or $d \geq b$, there is no intersection. If $c < a$, then $e2$ intersects $e1$ as a trailing edge (covered).
- Cases for $a > b$. Here, $e1$ is a bottom edge.
  - $c > d$: $e2$ is also a bottom edge. The cases here are the same as when $a < b$ and $c < d$, and both were top edges.

– $c = d$: $e2$ is a pair edge. If $a \geq c$ or $b \geq d$, there is no intersection. If $a < c$, and $b > d$, then the edges intersect with $e1$ a trailing edge (covered).
– $c < d$: $e2$ is now a top edge, and the cases follow similarly to when the roles of $e1$ and $e2$ were reversed (when $a < b$ and $c > d$).

End proof.

We introduce a structure we call an *accumulator tree* to efficiently compute the number of trailing edges to a range of nodes. There are two trees, one for the top level and one for the bottom level. Each leaf in the binary tree is the current count of trailing edges into a node of the given level. Non-leaf nodes contain the sum of the counts in their children. The number of trailing edges to a range of nodes $N_{j,...,}N_{LAST}$ is computed by walking up the tree from N_j, and summing the counts of the left subtrees into the path from the node to tree's root, and subtracting it from the value in the tree's root. In effect, we keep a sum of all the ranges in the tree, and then subtract out the portions in which we are not interested.

The accumulator tree is implemented by mapping it into an array, with the tree's root node at position 1, its left and right children at positions 2 and 3, etc. [9]. Thus, if the sequence of positions from a leaf node to the root is $P_{1,...,}P_k$, then if P_j is odd, we add the value of the entry in the tree for P_{j-1}. Figure 3 illustrates this for a level with 8 nodes, and shows the path through the tree to query the count into nodes at positions 6 through 8. The bracketed number is the position of a node in the array, and the unbracketed number is the position number of the node within the level. The square nodes are the path from node 6 (at entry "[13]") to the root node, and the nodes with a dot (nodes at "[2]" and "[12]") are nodes whose counts are summed to remove from the root node at "[1]".

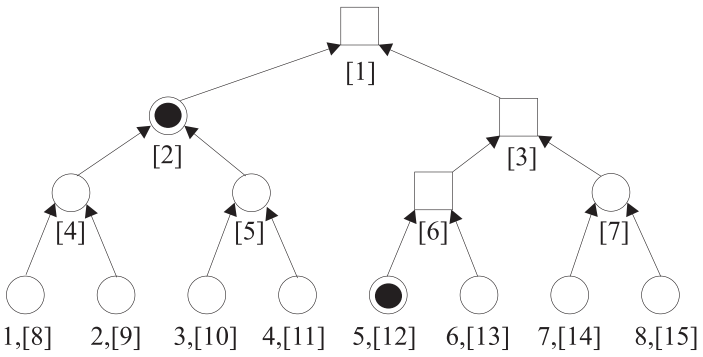

Figure 3. Accumulator tree.

The three operations of interest on the accumulator tree are: 1) CountEdges - compute the number of edges in a range of nodes (our range query), 2) AddEdges - add an edge to the tree, and 3) RemoveEdge - remove an edge from the tree.

We give below the procedure for CountEdges. The argument Tree gives the array implementing the accumulator tree, and n is the end node for the edge being counted. AddEdge adds one to each node on a path from from a leaf node to the root node. RemoveEdge is identical to AddEdge, except it decrements the count by one. "Base" is the amount added to map the node at position 1 into its position in the tree's array.

```
CountEdges(Tree, n, last) {
    if ( n == last )
      return 0;
    Pos = n + Base + 1;
    Sum = 0;
    while( Pos >= 1) {
       if ( Odd(Pos) )
         Sum = Sum + Tree[Pos - 1]
       Pos = Pos / 2;
    }
    return( Tree[1] - Sum);
}
```

Notice that Tree[1] is the sum of all the trailing edges for the current sweep position.

5 The LevelCross Algorithm

We can now give our algorithm to count crossings in a leveled graph, which we call the LevelCross algorithm. In the following, "AccL" and "AccM" are the arrays for the accumulator trees for levels L and M. "Count" is the number of crossings, "Pair" the number of pair edges, and "Same" the number of same level edges. Rt and Rb are the number of right-top and right-bottom edges, respectively, and Trt and Trb the number of trailing top and bottom edges. The predicate "IsPair(e)" is true iff e is a pair edge, and the predicate "Right(e)" is true iff e is a right edge.

For each level in the graph, the algorithm makes a sweep over the level, counting the edges of each type, based on the current sweep position. After the number of edge intersections is computed for the sweep position, the accumulator trees and counts are updated to reflect the change to the next sweep position.

LevelCross:
```
Count = 0;
for each level L of nodes $\{L_1, ..., L_K\}$
x       with a successor level M of nodes $\{M_1, ..., M_J\}$
    initialize trees AccL and AccM, for levels L and M, respectively, to 0
    for I = 1 to min$(K, J)$  // sweep position is $< L_I, M_I >$
        Pair = Rt = Rb = Same = 0;
        for each edge e = $< L_I, x >$ from $L_I$  // Count top edges
```

```
            if ( Right(e)) {
                Rt = Rt + 1
            } else if ( IsPair(e)) {
                Pair = Pair + 1
            }
            Same = Same + CountEdges( AccM, x, $J$)
    }
    Trt = AccL[1]
    Trb = AccM[1]
    for each edge e = $< x, M_I >$ from $M_I$  // Count bottom edges
        if ( Right(e))
            Rb = Rb + 1
            Same = Same + CountEdges( AccL, x, $K$)
        }
    }
    Count = Count + Trt*Rb + Trb*(Rt + Pair) + Rt*Rb + Same
    Move Sweep Position:

        1) Remove trailing edges to $L_I$ and $M_I$ from their respective trees.
        2) Right edges from $L_I$ become trailing edges in bottom tree
            (and are added to the tree), and similarly for right
            edges from $M_I$ in top tree.
    }
}
```

The intersection counting algorithm runs in O(e log e) time:

Proof:
The number of nodes on a level ranges between 1 and e, so each pass through a level's accumulator tree requires log e time, so within a loop over edges it requires *O(e log e)*. Based on simply the number of loops over edges, the algorithm might appear to run in time $6e \log e$. However, it really requires only $3e \log e$ since each edge is traversed once, as either a top or bottom edge. Further, the "sweep" loop over the nodes in a level does not contribute another multiple of "e" to the running time. This is because the nodes "partition" the edges in the sense that an edge is only examined via its end nodes, so edges are not repeatedly examined for each node. In effect, the nodes are used as a mechanism to iterate over the edges.

Thus, the algorithm's running time is *O(e log e)*.
End Proof.

6 Implementation Results

This section describes the performance measured when the LevelCross algorithm was implemented as part of the layout code in the NARC (Nodes and ARC) graph toolkit, which is an internal IBM tool that provides a high-level service to display, layout, and edit graphs as part of a user interface. LevelCross was implemented to replace the previous intersection counting algorithm in NARC's Sugiyama-style layout algorithm. It replaced the naive algorithm of simply sweeping over the nodes in the level and counting the crossings. The measurements were done on a 200 MHz Pentium Pro running OS/2 Warp (4.0). Notice that this is a much harsher test than just the performance of the intersection algorithm. The intersection algorithm must both be faster than the simpler algorithm, *and* counting intersections must be a bottleneck to the entire layout process. Otherwise, improvements due to the intersection algorithm will be hidden by the time required to perform the rest of the layout algorithm.

We devised an algorithm for generating random leveled graphs to produce the graphs used as input data. This allowed us to create graphs with controlled sizes and properties on demand. Briefly, the input to the algorithm consists of a set of input "tiles" (subgraphs), a desired graph size (number of nodes), the desired aspect ratio (number of levels divided by number of nodes per level), and the specification of "generators" to randomly select properties such as in-degree and the length of edges used to connect nodes. The nodes in each tile have a pre-computed level structure, and the tile has a set of its nodes on its top level designated as input nodes, and a set of nodes on its bottom level as output nodes. Each row of tiles is filled by creating instances of randomly selected tiles until the row had the desired width. The input nodes in each tile are connected to a randomly selected <tile, output node> pair from a preceding row of tiles.

We measured the execution time in milliseconds of the layout algorithm with the naive, "Simple" $O(e^2)$ crossing algorithm, and for the LevelCross algorithm on two series of randomly generated graphs. In each series of graphs, the number of target nodes in each graph were doubled from the previous graph. In the first series (shown in Table 1), the graphs were generated from a library of tiles in which each tile had 4 levels, and the in-edges were connected to a previous row selected from a uniformly distributed range of 1 to 4. (A "1" connects to a tile in the previous row, "2" the row before that, etc. This process creates long edges acording to the chosen distribution.)

As shown in Table 1, the graphs ranged in size from 168 nodes and 281 edges to 40,444 nodes and 74,189 edges in the input graph. The columns "Lay nodes" and "Lay edges" give the internal size of the graph after long edges have been replaced by a sequence of edges to virtual nodes, and the "density" columns give the ratio of edges/nodes for their respective columns. Initially, the layout times are the same for both crossing algorithms, but diverge with increasing graph sizes. (A 40K node graph was as large as the 64 Meg. memory on our test machine could handle.) On all but the last graph, the crossing reduction phase accounted for between 92 and 95 percent of the total layout time when using the simple algoritm, and accounted for 77 percent of the total layout time when level cross was implemented. In the last graph, the cost of creating the level structure and long edges increased by an order of magnitude. Since the number of nodes and edges only doubled, we assume this is due to exhausting memory.

To explore the effect of increased edge density, we generated a second series of graphs based on tiles with two levels, raised the in-degree of the nodes, and prevented generation of long edges, with the results shown in Table 2. (Both

Table 1

In Nodes	In edges	Lay nodes	Lay edges	In density	Lay density	Simple time(ms)	LevelCross time(ms)
168	281	261	374	1.67	1.43	16	16
341	569	640	868	1.67	1.36	47	39
680	1,193	1,509	2,002	1.75	1.34	148	109
1,297	2,314	3,340	3,341	1.78	1.3	414	250
2,611	4,710	7,857	9,956	1.8	1.27	1,383	625
5,121	9,350	16,945	21,174	1.83	1.25	3,961	1,344
10,174	18,713	33,065	41,064	1.79	1.24	10,039	2,672
20,117	37,001	69,654	86,538	1.84	1.24	29,219	5,914
40,444	74,189	139,220	172,965	1.83	1.24	69,024	15,563

Table 2

In Nodes	In edges	Density	Simple time(ms)	LevelCross time(ms)
157	499	3.18	23	16
312	1,145	3.27	31	23
633	2,191	3.46	94	54
1,275	4,387	3.4	218	109
2,524	8,869	3.51	734	274
5,076	18,138	3.57	2,000	586
10,044	35,485	3.53	5,375	1,242
20,003	71,885	3.59	15,000	2,477
40,081	143,806	3.59	42,063	5,632

long edges and tiles with more levels limit the edge density.) Since these graphs have no long edges, there are the same number of edges in the layout as in the input graph, and we omit separate columns for the layout graph. If we compare the results inTable 2 with those in Table 1, the simple algorithm degrades faster with a higher edge density. For example, for the last graph in Table 2, the layout with the simple algorithm takes over 7 times as long as LevelCross, where it only took 4.4 times as long for 40K node input graph in Table 1. The times are longer in Table 1 because the size of the layout graph is larger due to long edges.

These results show that the line crossing algorithm is a performance bottleneck for Sugiyama-style layout algorithms for large graphs, and our algorithm substantially improves its performance. Beyond this, we feel there are some lessons beyond a better crossing algorithm. First, Sugiyama-style algorithms are fast enough to lay out graphs of tens, or even hundreds of thousands of nodes. The 200 MHz Pentium Pro machine is now several years old, and ran out of memory, or we could have experimented with larger graphs. Thus, it is not necessary to restrict the layout algorithm to a spanning tree as was done in [Munzer] to handle very large graphs. Second, we found to our surprise that graphs of tens of thousands of nodes were "readable" on a 1600 by 1200 pixel display in the sense we could still see individual nodes and edges. The commonly held "folk theorem" seems to be that it is pointless to display graphs of more than a few hundred nodes without (possibly heavy) filtering, because they would be completely unreadable. In fact, we discovered that if the graph is not too dense, and the nodes not distributed too unevenly on the screen, very large graphs can be viewed with current technology. Further, display technology is improving, as exemplified by IBM's recent announcement [13] of a prototype LCD display with a resolution of 2048 by 2560 pixels. This display has almost 3 times the number of pixels in our display, and could usefully display even larger graphs.

Finally, we feel the entire area of graph drawing would benefit from the development of algorithms to generate graphs on demand for testing layout algorithms. While it is always better to have real-life examples rich with interesting semantics, these graphs are often valuable because they are rare. They may also not have the desired size or other properties needed to exercise a particular aspect of a layout algorithm. We found it very convenient to generate graphs by specifying some descriptive parameters, and believe this is a promising area for future research.

7 Conclusions

We have described an $O(e \log e)$ algorithm for counting the number of edge crossings in a leveled graph, and verified its performance by implementation. We have also shown that the line crossing algorithm can be a performance bottleneck in Sugiyama-style algorithms, and the new algorithm significantly improves their performance. The algorithm achieves its performance by taking advantage of the geometry of leveled graphs to classify the edges in the graph into types based their relationship to the current sweep position, and from a scheme to efficiently compute range queries on the end nodes of edges.

References

1. Bentley, J.L. and Ottman, T. Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.* C-28, (1979), pp 643-647.
2. Chazelle, B. Reporting and Counting Segment Intersections, *Journal of Computer and System Sciences*, Vol. 32, pp. 156-182, 1986.
3. Chazelle, B., and Edelsbrunner, H. An Optimal Algorithm for Intersecting Line Segments in the Plane, *JACM*, 39(1), pp. 1-54, Jan 1992.
4. Di Battista, G., Eades, P., Tamassia, R., and Tollis, I.G. *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999.
5. Edelsbrunner, H., and Welzl, E. Halfplanar Range Search in Linear Space and $O(n^{0.695})$ Query Time," *Tech. Univ. Graz*, IIG Report 111, 1983.
6. Gansner, E. R., North, S. C., and Ko, K. P. DAG: A program that draws directed graphs, *Software Practice and Experience*, 18(11), pp. 1047-1062, Nov 1988.
7. Garey, M. R., and Johnson, D.S. Crossing number is NP-Complete, *SIAM Journal on Algebraic and Discrete Methods*, 4, 3 (September 1983), pp. 312-316.

8. Huang, M. L., Eades, P., and Wang, J., On-line Animated Visualization of Huge Graphs using a Modified Spring Algorithm, *Journal of Visual Languages and Computing*, Vol 9., pp. 623-645 (1998).
9. Knuth, D. E., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Second Ed., pp. 400-401, Addison-Wesley, 1973.
10. Munzer, T., Exploring Large Graphs in 3D Hyperbolic Space, *IEEE Computer Graphics and Applications*, Vol. 18 No. 4, 1998.
11. Preparata, F. P., and Shamos, M. I., *Computational geometry - An introduction.* Springer-Verlag, New York, 1985.
12. Rowe, L., et. al., A Browser for Directed Graphs, *Software Practice and Experience*, Vol. 17(1), pp. 61-76, Jan. 1987.
13. Schleupen, K., P. Alt, et. al., "High Information Content Color 16.3 inch Desktop AMLCD with 15.7 Million a-Si:H TFTs," *Asia Display* '98, Digest, Seoul Korea, (1998) pp. 187-191.
14. Sugiyama, K., Tagawa, S., and M. Toda, Methods for Visual Understanding of Hierarchical Structures, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-11, No. 2. Feb. 1981.