Simple and Efficient Bilayer Cross Counting

Wilhelm Barth* Michael Jünger† Petra Mutzel‡

May 21, 2002

Abstract

We consider the problem of counting the interior edge crossings when a bipartite graph $G = (V, E)$ with node set V and edge set E is drawn such that the nodes of the two shores of the bipartition are on two parallel lines and the edges are straight lines. The efficient solution of this problem is important in layered graph drawing. Our main observation is that it can be reduced to counting the inversions of a certain sequence. This leads to an $O(|E| + |C|)$ algorithm, where C denotes the set of pairwise interior edge crossings, as well as to a simple $O(|E| \log |V_{\text{small}}|)$ algorithm, where V_{small} is the smaller cardinality node set in the bipartition of the node set $|V|$ of the graph. We present the algorithms and the results of computational experiments with these and other algorithms on a large collection of instances.

1 Introduction

Let $G = (N, S, E)$ be a bipartite graph with disjoint node sets N and S and let all edges in E have one end node in N and one in S. Furthermore, let $L_N, L_S \in R^2$ be two disjoint parallel lines, a "northern" and a "southern" line. A *bilayer drawing* $BLD(G)$ assigns all nodes $n_i \in N = \{n_0, n_1, \ldots, n_{p-1}\}$ to distinct points $P(n_i)$ on L_N and all nodes $s_j \in S = \{s_0, s_1, \ldots, s_{q-1}\}$ to distinct points $P(s_j)$ on L_S. The edges $e_k = (n_i, s_j) \in E = \{e_0, e_1, \ldots, e_{r-1}\}$ are assigned to straight line segments with end points $P(n_i)$ and $P(s_j)$, see Figure 1 for an example.

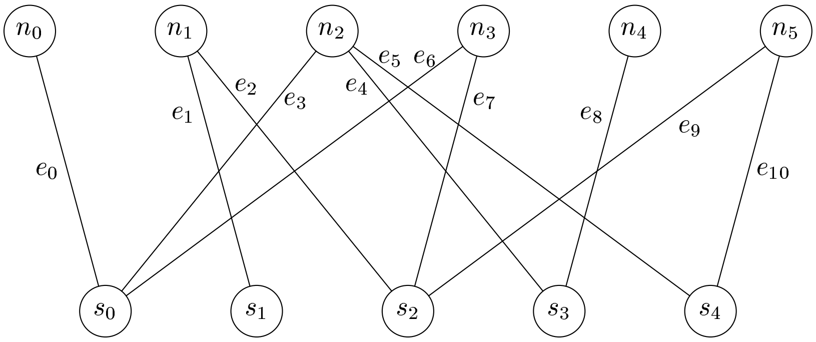

Figure 1: A bilayer drawing

Given a bilayer drawing $BLD(G)$ of a bipartite graph $G = (N, S, E)$, the *bilayer cross count* is the number $BCC(BLD(G))$ of pairwise interior intersections of the line segments corresponding to the edges.

*Institut für Computergraphik und Algorithmen, Technische Universität Wien, Favoritenstraße 9–11, A-1040 Wien, Austria, E-mail: `barth@ads.tuwien.ac.at`

†Institut für Informatik, Universität zu Köln, Pohligstraße 1, D-50969 Köln, Germany, E-mail: `mjuenger@informatik.uni-koeln.de`, partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

‡Institut für Computergraphik und Algorithmen, Technische Universität Wien, Favoritenstraße 9–11, A-1040 Wien, Austria, E-mail: `mutzel@ads.tuwien.ac.at`

The example in Figure 1 has a bilayer cross count of 12. It is a trivial observation that $BCC(BLD(G))$ only depends on the relative positions of the node points on L_N and L_S and not on their exact coordinates. Therefore, $BCC(BLD(G))$ is determined by permutations π_N of N and π_S of S. Given π_N and π_S, we wish to compute $BCC(\pi_N, \pi_S)$ efficiently by a simple algorithm. For ease of exposition, we assume without loss of generality that there are no isolated nodes and that $q \leq p$.

In automatic graph drawing, the most important application of bilayer cross counting occurs in implementations of Sugiyama-style layout algorithms [11]. Such a procedure has three phases. In the first phase, the nodes are assigned to m parallel layers for some $m \in N$ such that all edges join two nodes of different layers. Edges that connect non-adjacent layers are subdivided by artificial nodes for each traversed layer. In the second phase, node permutations on each layer are determined with the goal of achieving a small number of pairwise interior edge crossings. In the third phase, the resulting topological layout is transformed to a geometric one by assigning coordinates to nodes and edge bends. See Figure 2 for a typical Sugiyama-style layout in which an artificial node is assumed wherever an edge crosses a layer. In this example, the artificial nodes coïncide with the edge bends.

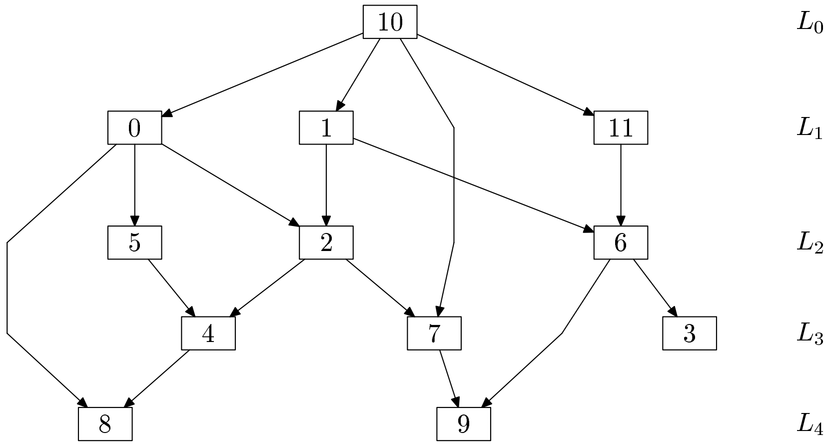

Figure 2: A typical Sugiyama-style layout

In phase two, popular heuristics approximate the minimum number of crossings with a layer by layer sweep. Starting from some initial permutation of the nodes on each layer, such heuristics consider pairs of layers $(L_{\text{fixed}}, L_{\text{free}}) = (L_0, L_1), (L_1, L_2), \ldots, (L_{m-2}, L_{m-1}), (L_{m-1}, L_{m-2}), \ldots, (L_1, L_0), (L_0, L_1), \ldots$ and try to determine a permutation of the nodes in L_{free} that induces a small bilayer cross count for the subgraph induced by the two layers, while keeping L_{fixed} temporarily fixed. These down and up sweeps continue until no improvement is achieved. The bilayer crossing minimization problem is NP-hard [4] yet there are good heuristics and it is even possible to solve this problem very quickly to optimality for instances with up to about 60 nodes per layer [6]. A common property of most algorithmic approaches is that a permutation of the nodes of the free layer is determined by some heuristic and then it must be decided if the new bilayer cross count is lower than the old one. This is the *bilayer cross counting problem* that we address in this paper. It has been observed in [12] that bilayer cross counting can be a bottleneck in the overall computation time of Sugiyama-style algorithms.

Of course, it is easy to determine if two given edges in a bilayer graph with given permutations π_N and π_S cross or not by simple comparisons of the relative orderings of their end nodes on L_N and L_S. This leads to an obvious algorithm with running time $O(|E|^2)$. This algorithm can even output the crossings rather than only count them, and since the number of crossings is $\Theta(|E|^2)$ in the worst case, there can be no asymptotically better algorithm. However, we do not need a list of all crossings, but only their number.

The bilayer cross counting problem is a special case of a core problem in computational geometry, namely counting (rather than reporting) the number of pairwise line crossings for a set of straight lines in the plane. Let C be the set of pairwise line crossings. The best known algorithm for reporting all these crossings is by Chazelle and Edelsbrunner [2] and runs in $O(|E|\log|E|+|C|)$ time and $O(|E|+|C|)$ space; the running time is asymptotically optimum. The best known algorithm for counting the crossings is by Chazelle [1] and runs in $O(|E|^{1.695})$ time and $O(|E|)$ space. For the bilayer cross counting problem, a popular alternative in graph drawing software is a sweep-line algorithm by Sander [10] that runs in $O(|E|+|C|)$ time and $O(|E|)$ space. This algorithm is implemented, e.g., in the VCG tool [9] or the AGD library [5].

A breakthrough in theoretical and practical performance is an algorithm by Waddle and Malhotra [12] that runs in $O(|E|\log|V|)$ time and $O(|E|)$ space, where $V = N \cup S$. The authors report on computational experiments that clearly show that the improvement is not only theoretical but leads to drastic time savings in the overall computation time of a Sugiyama-style algorithm that is implemented in an internal IBM software called NARC (Nodes and ARC) graph toolkit. Their algorithm consists of a sweep-line procedure that sweeps the bilayer graph once, say from west to east, and maintains a data structure called *accumulator tree* that is similar to the *range tree* data structure that is common in computational geometry, e.g., when a finite set of numbers is given and the task is to determine the cardinality of its subset of numbers that lie in a specified interval, see Lueker [8]. The sweep-line procedure involves complicated case distinctions and its description takes several pages of explanation and pseudo-code.

In Section 2 we give a simple proof of the existence of $O(|E|\log|V|)$ algorithms for bilayer cross counting by relating the bilayer cross count to the number of inversions in a certain sequence. This observation immediately leads to a bilayer cross counting algorithm that runs in $O(|E|+|C|)$ time and $O(|E|)$ space like the algorithm by Sander [10] and another algorithm that runs in $O(|E|\log|V|)$ time and $O(|E|)$ space like the algorithm by Waddle and Malhotra [12]. In Section 3, we present an even simpler algorithm that runs in $O(|E|\log|V_{\text{small}}|)$ time and $O(|E|)$ space, where V_{small} is the smaller cardinality set of N and S. This algorithm is very easy to understand and can be implemented in a few lines of code. The question how the old and the new algorithms perform in direct comparison is addressed empirically in Section 4. It turns out that the algorithm presented in detail in Section 3 outperforms the others not only in terms of implementation effort, but in most cases also in terms of running time.

2 Bilayer cross counts and inversion numbers

In a sequence $\pi = \langle a_0, a_1, \ldots, a_{t-1}\rangle$ of pairwise comparable elements a_i $(i = 0, 1, \ldots, t-1)$, a pair (a_i, a_j) is called an *inversion* if $i < j$ and $a_i > a_j$. The *inversion number* $INV(\pi) = |\{(a_i, a_j) \mid i < j \text{ and } a_i > a_j\}|$ is a well known measure of the degree of sortedness of the sequence π.

In a given bilayer graph with northern layer permutation $\pi_N = \langle n_0, n_1, \ldots, n_{p-1}\rangle$ and southern layer permutation $\pi_S = \langle s_0, s_1, \ldots, s_{q-1}\rangle$ let $\pi_E = \langle e_0, e_1, \ldots, e_{r-1}\rangle$ be sorted lexicographically such that $e_k = (n_{i_k}, s_{j_k}) < (n_{i_l}, s_{j_l}) = e_l$ in π_E iff $i_k < i_l$ or $i_k = i_l$ and $j_k < j_l$. In Figure 1, the edges are sorted like this. Let $\pi = \langle j_0, j_1, \ldots, j_{r-1}\rangle$ be the sequence of the positions of the southern end nodes in π_E. In our example, we have $\pi = \langle 0,1,2,0,3,4,0,2,3,2,4\rangle$. Each inversion in π is in a 1-1 correspondence to a pairwise edge crossing in a bilayer graph drawing $BLD(G)$ according to π_N and π_S. Therefore, $BCC(\pi_N, \pi_S)$ is equal to the number of inversions in π.

It is well known that the number of inversions of an r-element sequence π can be determined in $O(r\log r)$ time and $O(r)$ space, e.g., Cormen, Leiserson, and Rivest [3] suggest an obvious modification of the merge sort algorithm in exercise 1-3d. Since the lexicographical ordering that leads to π can be computed in $O(|E|)$ time and space, this implies immediately the existence of an $O(|E|\log|V|)$ time and $O(|E|)$ space algorithm for bilayer cross counting. More precisely, the (modified) merge sorting algorithm can be implemented to run in $O(r\log RUN(\pi))$ time and $O(r)$ space, where $RUN(\pi)$ is the number of *runs*, i.e., the number of sorted subsequences in π. This appears attractive when $RUN(\pi)$ is expected to be small. We will test this empirically in Section 4. The number of inversions of a sequence π can also be determined with the insertion sort algorithm with $O(|E|+INV(\pi))$ time and $O(|E|)$ space consumption, and this immediately gives an $O(|E|+|C|)$ time and $O(|E|)$ space algorithm for bilayer cross counting. We will work out this idea in detail in the following section, and improve the performance to $O(|E|\log|V_{\text{small}}|)$ running time. An algorithm for counting the inversions of an r-element sequence

π with elements in $\{0, \ldots, q-1\}$, $q \leq r$, with running time better than $O(r \log r)$ would immediately improve the bilayer cross counting approaches based on counting inversions. We do not know if such an algorithm exists.

3 A simple $O(|E| \log |V_{\text{small}}|)$ algorithm

Our task is the efficient calculation of the number of inversions of the sequence π coming from a bilayer graph drawing according to π_N and π_S as described in Section 2.

We explain our algorithm in two steps. In step 1, we determine the bilayer cross count by an insertion sort procedure in $O(|E|^2)$ time, and in step 2, we use an accumulator tree to obtain $O(|E| \log |V_{\text{small}}|)$ running time. We use the example of Figure 1 to illustrate the computation. Here is step 1:

(1) Sort the edges lexicographically according to π_N and π_S by radix sort as described in Section 2. This takes $O(|E|)$ time. In Figure 1, this step has already been performed and the edges are indexed in sorted order.

(2) Put the positions of the southern end nodes of the edges into an array in sorted order of (1). In the example, we obtain $\langle 0, 1, 2, 0, 3, 4, 0, 2, 3, 2, 4 \rangle$.

(3) Run the insertion sort algorithm (see, e.g. [3]) on the array and accumulate the bilayer cross count by adding the number of positions each element moves forward. In the illustration on our example in Figure 3 we also show the nodes of N and the edges of E. This additional information is not needed in the algorithm, it just helps visualizing why the procedure indeed counts the crossings. In our example, the answer is $2 + 4 + 2 + 1 + 3 = 12$ crossings.

The correctness of this algorithm follows from the fact that whenever an element is moved, the higher indexed elements are immediately preceding it in the current sequence. This is the important invariant of the insertion sort algorithm. So the total number of positions moved is equal to the number of crossings.

Insertion sort takes linear time in the number of inversions, and since there are $\binom{|E|}{2}$ inversions in the worst case, we have described an $O(|E|^2)$ algorithm for bilayer cross counting. We use an accumulator tree as in [12] in order to obtain an $O(|E| \log |V_{\text{small}}|)$ algorithm. Namely, let $c \in N$ be defined by $2^{c-1} < q = |S| \leq 2^c$, and let T be a perfectly balanced binary tree with 2^c leaves whose first q are associated with the southern node positions.

We store the accumulator tree T in an array with $2^{c+1} - 1$ entries in which the root is in position 0 and the node in position i has its parent in position $\lfloor \frac{i-1}{2} \rfloor$. All array entries are initialized to 0. The occupied leaves will contain the degrees of their associated southern nodes and the internal nodes will contain the sum of the entries of their children. We build up this information by processing the southern end node positions in the order given by π. For each such position, we start at its corresponding leaf and go up to the root and increment each visited tree position (including the root) by 1. In this process, whenever we visit a left child (odd position in the tree), we add the entry in its right sibling to the number of crossings (which is initialized to 0). In Figure 4, we demonstrate this for our example: Inside each tree node, we give its corresponding tree index, and to the right of it, we give the sequence of entries as they evolve over time. An entry $\substack{v \\ j}$ indicates that value v is reached when the j-th element of the sequence π is inserted. The bilayer cross count becomes 2, 6, 8, 9, and 12, when the southern end node positions of e_3, e_6, e_7, e_8, and e_9, respectively, are inserted.

By our reasoning above, the correctness of the algorithm is obvious and, if we assume without loss of generality that $|S| \leq |N|$, i.e., $V_{\text{small}} = S$, we have a running time of $O(|E| \log |V_{\text{small}}|)$. Figure 5 displays a C-program fragment that implements the algorithm. The identifier names correspond to the notation we have used above, or are explained in comments, respectively. The identifier `southsequence` points to an array corresponding to the sequence π of the southern end node positions after the radix sorting (not shown here) has taken place.

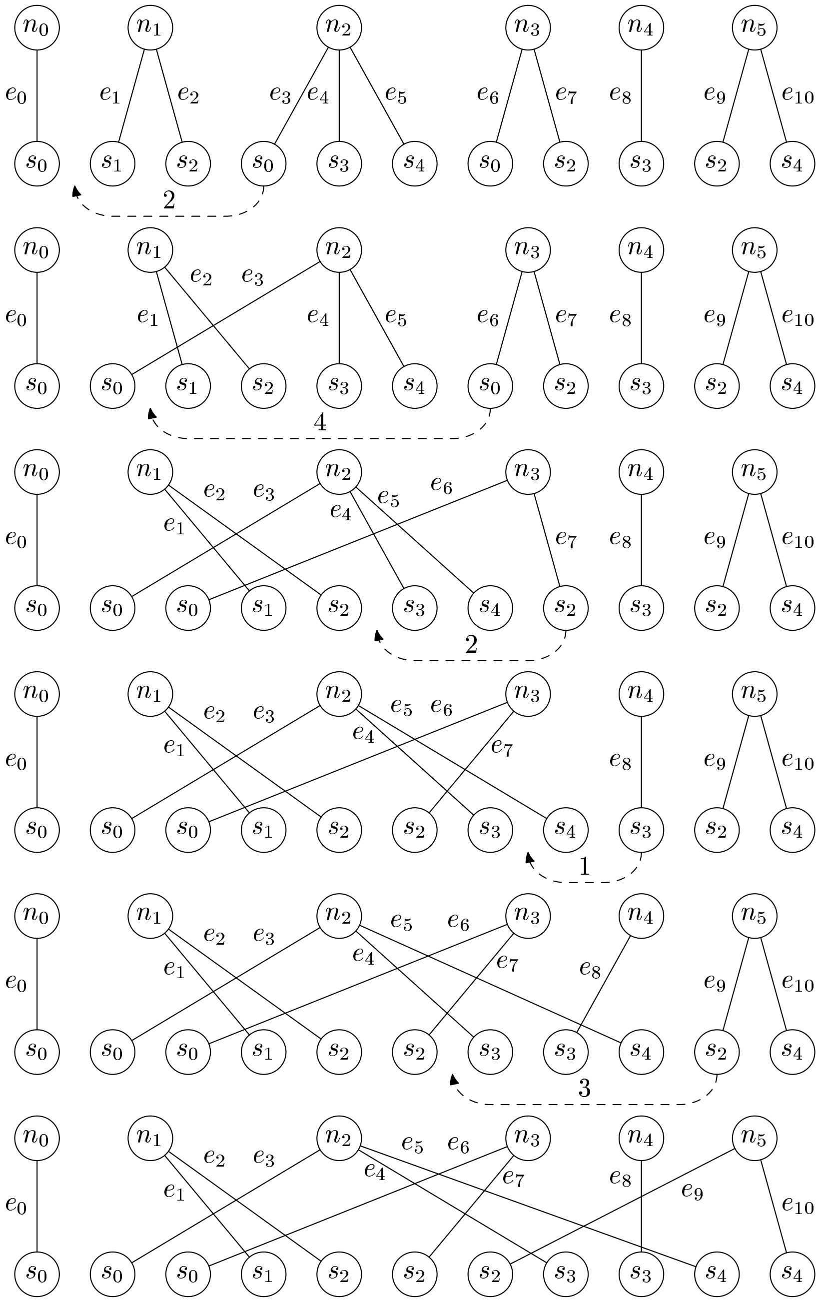

Figure 3: Counting crossings via insertion sort

4 Computational Experiments

In order to obtain an impression of how old and the new algorithms for bilayer cross counting perform in direct comparison, we made an empirical study.

We implemented the following algorithms in the C programming language as functions and used them in various computational experiments:

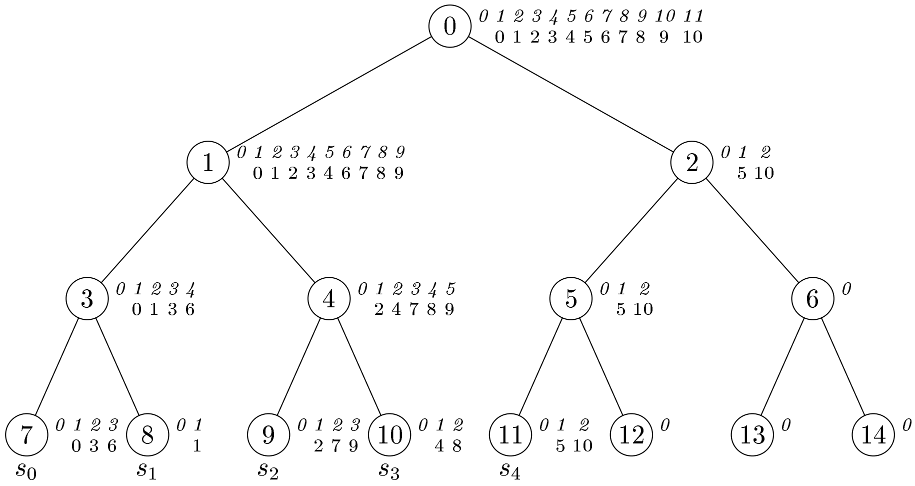

Figure 4: Building the accumulator tree and counting the crossings

```
/* build the accumulator tree */

firstindex = 1;
while (firstindex<q) firstindex *= 2;
treesize = 2*firstindex - 1; /* number of tree nodes */
firstindex -= 1; /* index of leftmost leaf */
tree = (int *) malloc(treesize*sizeof(int));
for (t=0; t<treesize; t++) tree[t] = 0;

/* count the crossings */

crosscount = 0; /* number of crossings */
for (k=0; k<r; k++) { /* insert edge k */
  index = southsequence[k] + firstindex;
  tree[index]++;
  while (index>0) {
    if (index%2) crosscount += tree[index+1];
    index = (index - 1)/2;
    tree[index]++;
  }
}
printf("Number of crossings: %d\n",crosscount);
free(tree);
```

Figure 5: C program fragment for simple bilayer cross counting

SAN is the algorithm by Sander [10] that runs in $O(|E| + |C|)$ time and $O(|E|)$ space,

WAM is the algorithm by Waddle and Malhotra [12] that runs in $O(|E| \log |V|)$ time and $O(|E|)$ space,

MER is a merge sorting algorithm (Section 2) that runs in $O(|E| \log RUN(\pi))$ time and $O(|E|)$ space,

INS is a plain insertion sorting algorithm (Section 2) that runs in $O(|E| + |C|)$ time and $O(|E|)$ space,

BJM is the algorithm described in Section 3 that runs in $O(|E| \log |V_{\text{small}}|)$ time and $O(|E|)$ space.

In order to make the comparison as fair as possible, all C-functions have the same parameters:

int p: p is the number of nodes in the northern layer,

int q: q is the number of nodes in the southern layer ($q \leq p$),

int r: r is the number of edges,

int NorthNodePos*: $NorthNodePos[k] \in \{0, 1, \ldots, p-1\}$ is the position of the northern end node of edge $k \in \{0, 1, \ldots, r-1\}$ in the northern permutation π_N,

int SouthNodePos*: $SouthNodePos[k] \in \{0, 1, \ldots, q-1\}$ is the position of the southern end node of edge $k \in \{0, 1, \ldots, r-1\}$ in the southern permutation π_S.

No assumption is made about the ordering of the edges, e.g., MER and BJM start by computing *southsequence* by a two phase radix sort. Likewise, the other algorithms compute the internally needed information from the given data that should be readily available in any reasonable implementation of a Sugiyama-style layout algorithm. Furthermore, the functions are responsible for allocating and freeing temporarily needed space. We made an effort in implementing all five algorithms as well as we could.

All experiments were performed under Linux on a SONY VAIO PCG-R600HEP notebook with an 850 MHz INTEL Mobile Pentium III processor and 256 MB of main memory. The software was compiled by the GNU `gcc` compiler with optimization option `O3`. All uniformly distributed random numbers needed in our experiments were generated by the C-function `gb_unif_rand` of Donald Knuth's Stanford GraphBase [7]. In all subsequent plots, data points are averages for 100 instances each.

The crossing minimization phase of a Sugiyama-style layout algorithm typically starts with random permutations of the nodes on each layer, and in the course of the computation, the edges become more and more untangled. This means that a bilayer cross counting algorithm is likely to be initially confronted with random permutations π_N and π_S and later with permutations that induce significantly less crossings. In our experiments, we take this phenomenon into account by running each layer pair twice – first with random permutations and then with permutations generated by a crossing minimization algorithm. The fastest method with good practical results we know is the so-called MEDIAN crossing minimization algorithm [4]. While the node permutation in one of the two layers is temporarily fixed, the nodes of the other layer are reordered according to the median positions of their neighbors in the fixed layer. The MEDIAN heuristic can be implemented to run in $O(|E|)$ time and space. After some experimentation, we decided that four iterations (reorder southern, then northern, then southern, and finally northern layer) give reasonable results. The second run is performed after such a reordering.

In our first experiment, we consider sparse graphs with 1,000 to 30,000 nodes on each layer and 2,000 to 60,000 randomly drawn edges. The average running times are plotted in Figure 6. Here and in the subsequent figures, the suffix "RAN" indicates that the running times are for the instances with random permutations of the two layers and the suffix "MED" indicates that the running times are for the MEDIAN-ordered instances.

The first observation is that SAN and INS are unpractical for large instances while all other procedures have very reasonable running times. This behavior extends to very large graphs as can be seen in Figure 7 for instances up to 500,000 nodes on each layer and 1,000,000 randomly drawn edges. BJM dominates all other methods for up to about 50,000 nodes both for the "RAN" and for the "MED" instances, for larger instances BJM leads in the "MED" case and MER leads in the "RAN" case. However, the differences are so small that they can possibly be attributed to system or implementation peculiarities, just like the slight peak for 35,000 nodes in Figure 7.

Now we study the behavior of the algorithms for instances of increasing density with 1,000 nodes on each layer. The number of edges grows from 1,000 to 100,000.

As before, SAN and INS are not competitive. Up to about 30,000 edges, BJM is the best method and beyond, WAM is slightly better.

Finally, we ran the algorithms on a selection of real-world graphs compiled from the AT&T directed graph collection by Michael Krüger of the Max-Planck-Institut für Informatik in Saarbrücken. We used

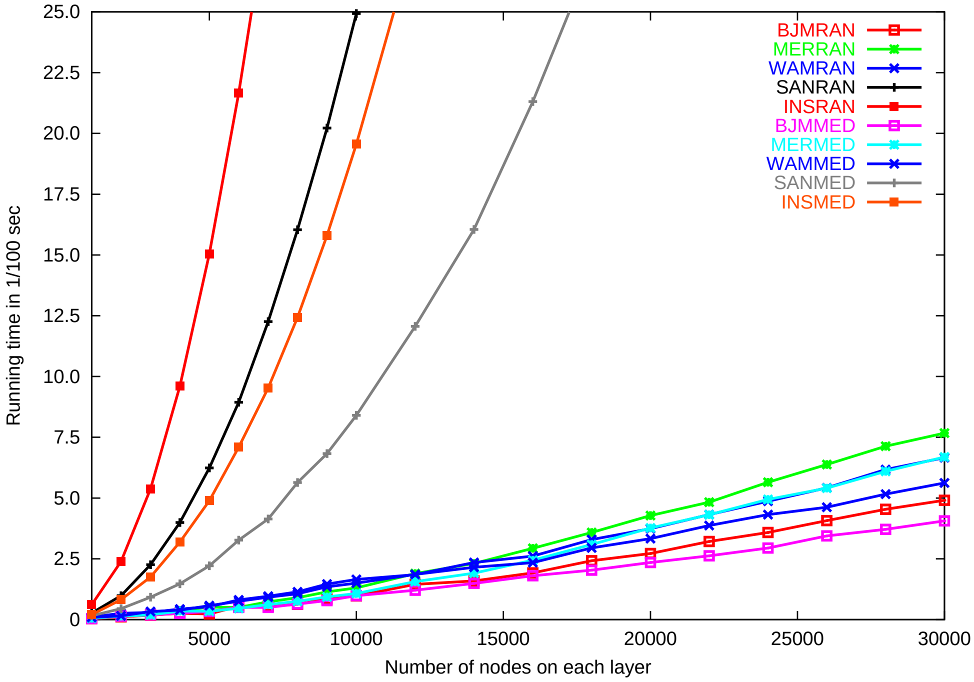

Figure 6: Running time for sparse graphs

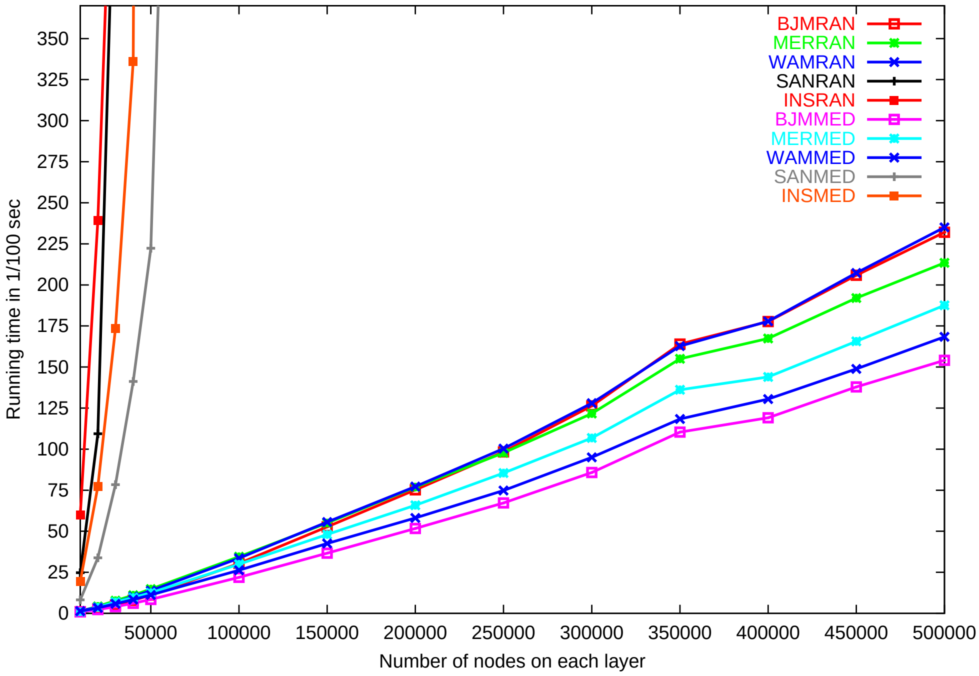

Figure 7: Running time for large sparse graphs

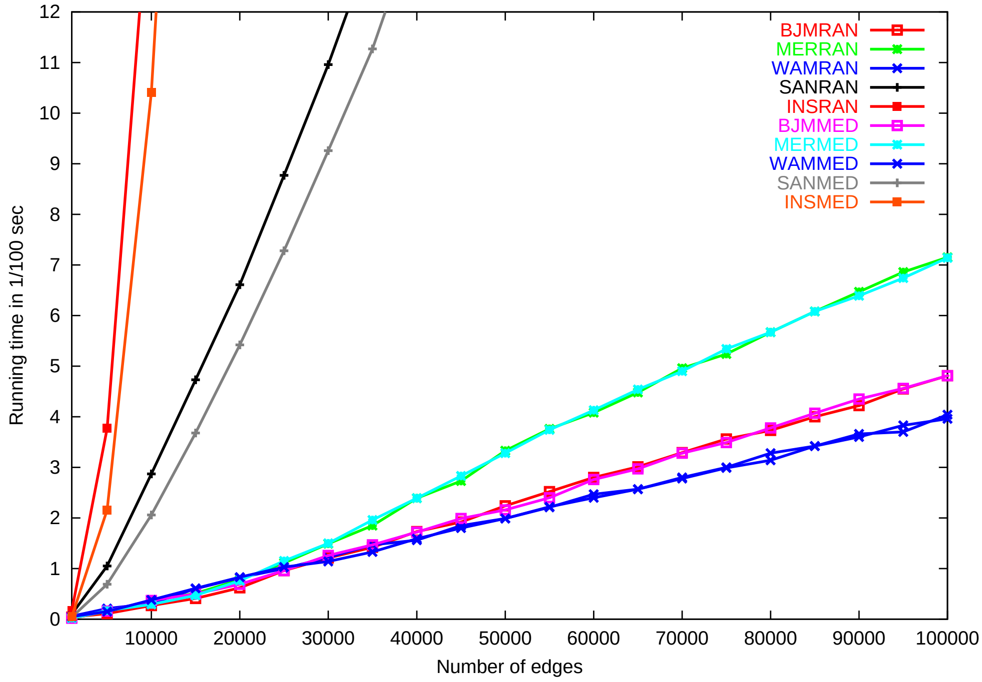

Figure 8: Running time for graphs with increasing density

the first phase of the AGD Sugiyama implementation in order to obtain layerings with the Longest-Path and Coffman-Graham options from which we extracted the resulting layer pairs as test instances. Thus, we compiled two collections of 30,061 instances and 57,300 instances, respectively. For each instance, we applied 10 random shuffles of the northern and southern layers, each followed by a MEDIAN-ordered run as explained above. So we ran a total of 601,220 and 1,146,000 instances of the Longest-Path generated layer pairs and the Coffman-Graham generated layer pairs, respectively.

In the Longest-Path case, the number of northern nodes varies between 1 and 6,566, with 63 on the average, the number of southern nodes varies between 1 and 5,755, with 57 on the average, and the number of edges varies between 1 and 6,566, with 64 on the average. For the random shuffles, the number of crossings varies between 0 and 10,155,835, with 24,472 on the average and for the MEDIAN ordered layers, the number of crossings varies between 0 and 780,017, with 182 on the average.

In the Coffman-Graham case, the number of northern nodes varies between 1 and 3,278, with 142 on the average, the number of southern nodes varies between 1 and 3,278, with 137 on the average, and the number of edges varies between 1 and 3,276, with 141 on the average. For the random shuffles, the number of crossings varies between 0 and 2,760,466, with 47,559 on the average and for the MEDIAN ordered layers, the number of crossings varies between 0 and 2,872, with 4 on the average.

The total running times are reported in Figure 9. The low crossing numbers in the MEDIAN case explain why INS and MER are the clear winners. With very few inversions and very few runs, INS and MER have almost nothing to do while the other methods do not profit much from this fact. We doubt, however, that we picked “typical real world data”. Further experiments may lead to hybrid methods that choose the algorithm according to the problem characteristics. Nevertheless, our experiments indicate that WAM and BJM are safe choices. It should be kept in mind that the running time of a MEDIAN step is of the same order of magnitude as a bilayer cross counting step with one of the fast algorithms. Therefore, bilayer cross counting may dominate the work in the second phase of a Sugiyama-style layout algorithm significantly, unless WAM, MER, BJM, or a method of comparable performance is used.

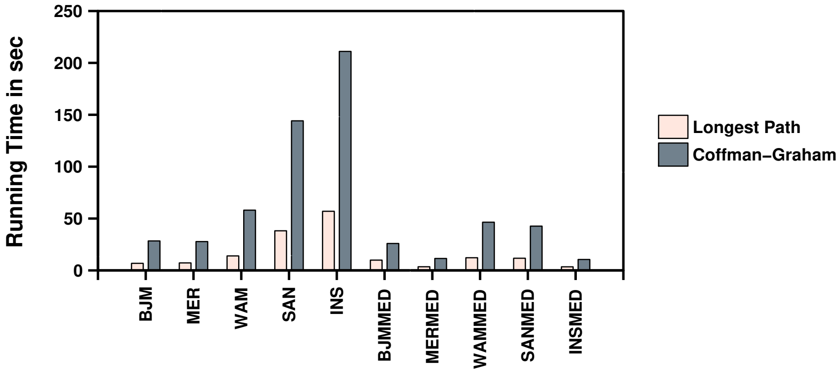

Figure 9: Running time for AT&T graphs

References

[1] B. Chazelle, Reporting and counting segment intersections. Journal of Computer and System Sciences **32** (1986) 156–182.

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

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to algorithms. MIT Press, Cambridge, MA, 1990.

[4] P. Eades and N. Wormald, Edge crossings in drawings of bipartite graphs. Algorithmica **11** (1994) 379–403.

[5] C. Gutwenger, M. Jünger, G. W. Klau, S. Leipert, and P. Mutzel, Graph Drawing Algorithm Engineering with AGD. in: S. Diehl (ed.), Software Visualization, International Dagstuhl Seminar on Software Visualization 2001, Lecture Notes in Computer Science 2269, Springer, 2002, pp. 307–323, see also: `http://www.mpi-sb.mpg.de/AGD/`

[6] M. Jünger and P. Mutzel, 2-layer straight line crossing minimization: performance of exact and heuristic algorithms. Journal of Graph Algorithms and Applications **1** (1997) 1–25.

[7] D. E. Knuth, The Stanford GraphBase: A platform for combinatorial computing. Addison-Wesley, Reading, Massachusetts, 1993

[8] G. S. Lueker, A data structure for orthogonal range queries. Proceedings of the 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 28–34.

[9] G. Sander, Graph Layout through the VCG Tool. in: R. Tamassia and I. G. Tollis (eds): Graph Drawing 1994, Lecture Notes in Computer Science 894, Springer, 1995, pp. 194-205, see also: `http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html`

[10] G. Sander, Visualisierungstechniken für den Compilerbau. Pirrot Verlag & Druck, Saarbrücken, 1996.

[11] K. Sugiyama, S. Tagawa, and M. Toda, Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics **11** (1981) 109–125.

[12] V. Waddle and A. Malhotra, An E log E line crossing algorithm for levelled graphs. in: J. Kratochvíl (ed.) Graph Drawing 1999, Lecture Notes in Computer Science 1731, Springer, 1999, pp. 59–70.