An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing[⋆]

Markus Eiglsperger[1], Martin Siebenhaller[2], and Michael Kaufmann[2]

[1] Universität Konstanz, Fakultät für Informationswissenschaften, 78457 Konstanz, Germany
markus.eiglsperger@uni-konstanz.de

[2] Universität Tübingen, WSI für Informatik, Sand 13, 72076 Tübingen, Germany
{siebenha,mk}@informatik.uni-tuebingen.de

Abstract. Sugiyama's algorithmic framework for layered graph drawing is commonly used in practical software. The extensive use of dummy vertices to break long edges between non-adjacent layers often leads to unsatisfactorial performance. The worst-case running-time of Sugiyama's approach is $O(|V||E|\log|E|)$ requiring $O(|V||E|)$ memory, which makes it unusable for the visualization of large graphs. By a conceptually simple new technique we are able to keep the number of dummy vertices and edges linear in the size of the graph and hence reduce the worst-case time complexity of Sugiyama's approach by an order of magnitude to $O((|V|+|E|)\log|E|)$ requiring $O(|V|+|E|)$ space.

1 Introduction

Most approaches for drawing directed graphs used in practice follow the same framework developed by Sugiyama et al. [17], which produces layered layouts [3]. This framework consists of four phases: In the first phase, called *Cycle Removal*, the directed input graph $G = (V, E)$ is made acyclic by reversing appropriate edges. During the second phase, called *Layer Assignment*, the vertices are assigned to horizontal layers. Before the third phase starts, long edges between vertices of non-adjacent layers are replaced by chains of dummy vertices and edges between the corresponding adjacent layers. Hence in the third phase, called *Crossing Reduction*, an ordering of the vertices within a layer is computed such that the number of edge crossings is reduced. Finally, the fourth phase, called *Horizontal Coordinate Assignment*, calculates an x-coordinate for each vertex. Now the dummy vertices introduced after the layer assignment are removed and replaced by bends.

Unfortunately, almost all problems occuring during the single phases of this approach are NP-hard: Feedback-arc set [12], Precedence Constrained Multiprocessor Scheduling [5], 2-layer crossing minimization [8], etc. Nevertheless, for

[⋆] This work has partially been supported by the DFG-grant Ka512/8-2. It has been performed when the first author was with the Universität Tübingen.

all these problems appropriate heuristics have been developed and nearly all practical graph drawing software use this approach, mostly enriched by modifications required in practice like large vertices, same-layer-edges, clustering, etc.

In the following, we review Sugiyama's framework for drawing directed graphs in more detail and give the necessary definitions and results. Then we use this as basis for our new approach. In the rest of this work we assume that the input graph is already acyclic.

1.1 Layer Assignment and Normalization

Let L_1,..,L_h be a partition of V with $L_i \subset V$, $1 \leq i \leq h$ and $\bigcup_{i=1}^{h} L_i = V$ (h denotes the number of layers). Such a partition is called a layering of G if for all $e = (v, w)$ with $v \in L_i$ and $w \in L_j$ holds $i < j$. The number of vertices in a layer L_i is denoted with n_i. The span of edge e is $j - i$. In a layered drawing, all vertices $v \in L_i$ are drawn on a horizontal line (same y-coordinate). We call the layering proper if $span(e) = 1$ for all edges $e \in E$. In most applications the layers of the vertices can be assigned arbitrarily and, in some cases, the layer assignment is even part of the input.

For edges $e = (u, v)$ with $span(e) > 1$ and for which the endpoints u and v lie on layers L_i and L_j, we replace edge e by a chain of dummy vertices $u = d_i, d_{i-1}, \ldots, d_{j+1}, d_j = v$ where any two consecutive dummy vertices are connected by a dummy edge. Vertex d_k for $i \leq k \leq j$ is placed on layer L_k. This process is called *normalization* and the result the *normalized graph* $G_N = (V_N, E_N)$. With this construction, the next phase starts with a proper layering.

Gansner et al. [10] presented an algorithm, which calculates a layer assignment of the vertices such that the total number of dummy vertices is minimized. The algorithm for minimizing the number of dummy vertices is a network simplex method and no polynomial time bound has been proven for it, but several linear time heuristics for this problem work well in practice [14, 15]. In the worst case $|V_N| = O(|V||E|)$ and $|E_N| = O(|V||E|)$.

After the final layout of the modified graph, we replace the chains of dummy edges by polygonal chains in which the former dummy vertices become bends.

1.2 Crossing Reduction

The vertices within each layer L_i are stored in an ordered list, which gives the left-to-right order of the vertices on the corresponding horizontal line. Such an ordering is called a *layer ordering*. We will often identify the layer with the corresponding list L_i. The ordering of the vertices within adjacent layers L_{i-1} and L_i determines the edge crossings with endpoints on both layers.

Crossing reduction is usually done by a layer-by-layer sweep where each step minimizes the number of edge crossings for a pair of adjacent layers. This layer-by-layer sweep is performed as follows: We start by choosing an arbitrary vertex order for the first layer L_1 (we number the layers from top to bottom). Then iteratively, while the vertex ordering of layer L_{i-1} is kept fixed, the vertices of

L_i are put in an order that minimizes crossings. This step is called one-sided two-layer crossing minimization and is repeated for $i = 2, .., h$. Then the sweep direction is reversed and repeated until no further crossings can be saved.

Many heuristics have been proposed to attack the one-sided two-layer crossing minimization problem [3, 6]. Most important are the *median* and the *barycenter heuristic*, where the new position of each vertex v in list L_i is chosen relative to the position of the adjacent vertices from list L_{i-1}.

To decide whether we improved the number of crossings by a sweep, we have to count this number. This important subproblem, called the *bilayer cross counting* problem, has to be solved in each of the steps. The naive sweep-line algorithm needs time $O(|E'| + |C'|)$ where $|E'|$ is the number of edges between the two layers and $|C'|$ the number of crossings between these edges [15]. It has recently been improved to $O(|E'| \log |V'|)$ by Waddle [19] and Barth et al. [2].

The algorithm reduces the bilayer cross counting problem to the problem of counting the inversions in the vertex sequences of layers L_{i+1} and L_i respectively. The number of inversions are counted by means of an efficient data structure, called the accumulator tree T.

1.3 Horizontal Coordinate Assignment

The horizontal coordinate assignment computes the x-coordinate for each vertex with respect to the layer ordering computed by the crossing reduction phase. There are two objectives to consider to get nice drawings. First the drawings should be compact and second the edges should be as vertical as possible.

Gansner et al. [10] model this problem as a linear program:

$$\min \sum_{(v,w)\in E} \Omega(v,w) \cdot |x(v) - x(w)|$$

$$s.t. \quad x(b) - x(a) \geq \delta(a,b) \quad a, b \text{ consecutive in } L_i,\ 1 \leq i \leq h$$

where $\Omega(v,w)$ denotes the priority to draw edge (v,w) vertical and $\delta(a,b)$ denotes the minimum distance of consecutive vertices a and b. This linear program can be interpreted as a rank assignment problem on a compaction graph $G_a = (V, \{(a,b) : a, b \text{ consecutive in } L_i,\ 1 \leq i \leq h\})$ with length function δ. Each valid rank assignment corresponds to a valid drawing. The above objective function can be modeled by adding vertices and edges to G_a [10].

The drawback of the above approach is, that edges can have as many bends as dummy vertices. This creates sometimes a "spaghetti" effect which reduces the readability. To avoid this negative behaviour the *linear segments model* was proposed, where each edge is drawn as polyline with at most three segments. The middle segment is always drawn vertical. In general, linear segment drawings have less bends but need more area than drawings in other models. There have been a number of algorithms proposed for this model [4, 15]. The approach of Brandes and Köpf [4] produces pleasing results in linear time.

1.4 Drawbacks

The complexity of algorithms in the Sugiyama framework heavily depends on the number of dummy vertices inserted. Although this number can be minimized efficiently, it may still be in the order of $O(|V||E|)$ [9]. Assume we use an algorithm based on the Sugiyama framework which uses the fastest available algorithms for each phase. Then this algorithm has running time $O(|V||E| \log |E|)$ and uses $O(|V||E|)$ memory.

To improve the running time and space complexity we avoid introducing dummy vertices for each layer that an edge spans. We rather split edges only in a limited number of segments. As a result, there may be edges which traverse layers without having a dummy vertex in it. We will extend the existing crossing reduction and coordinate assignment algorithms to handle this case.

A similar idea is used in the Tulip-software described in [1]. Unfortunately, no details are given. However, in this approach, only the proper edges are considered in the crossing reduction phase and the long edges are ignored. This leads to drawings which have many more crossings than drawings using the traditional Sugiyama approach. In contrast, we will show that our approach yields the same results as the methods traditionally used in practice.

2 The New Approach

The basic idea of our new approach is the following: Since in the linear segments model each edge consists of at most two bends, all corresponding dummy vertices in the middle layers have the same x-coordinate. We combine them into one *segment* and therefore reduce the size of the normalized graph dramatically. More precisely, if edge $e = (v, w)$ spans between layers L_i and L_j with $|j - i| > 2$, we introduce only two dummy vertices: p_e at layer L_{i+1} (called p-vertex) and q_e at layer L_{j-1} (called q-vertex), as well as three edges: (v, p_e), $s_e = (p_e, q_e)$, and (q_e, w). The first and the last edge are proper while s_e, called the *segment* of e, is not necessarily proper. If $|j - i| = 2$ we insert a single dummy vertex r_e. We call this transformation *sparse normalization* and the result the *sparse normalized graph* $G_S = (V_S, E_S)$. The size of the sparse normalized graph is linear with respect to the size of the input graph.

A layer L of a sparse normalized graph contains vertices and segments. A layer ordering of a sparse normalized graph is a linear ordering of the vertices and segments in a layer and is called a *sparse layer ordering.* For a graph G, there is a one-to-one correspondence between layer orderings of the normalized graph G_N and sparse layer orderings of the sparse normalized graph G_S.

Let us look at the layer orderings of normalized graphs: instead of storing the layer ordering in lists, we can store it in a directed graph D. This graph has an edge between vertices v and w if and only if these two vertices are in the same layer i and are consecutive in L_i. The ordering $<$ defined as $v < w$ if and only if there is a directed path from v to w in D, is a complete ordering for the vertices of a layer, i.e., either $v < w$ or $w < v$ for $v, w \in L_i$. In fact D is the compaction graph G_a mentioned in the preceding section. The

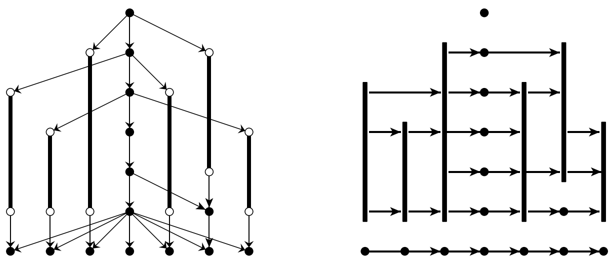

Fig. 1. In the left figure a sparse normalized graph is shown. Thick lines denote the segments. The right figure shows the corresponding compaction graph.

graph D has $|V_N|$ vertices and $O(|V_N|)$ edges, which results in a worst case size of $O(|V||E|)$.

We want to reduce the size of D to $O(|V| + |E|)$ without losing the property that $<$ defines a total layer ordering. The key observation therefor is that the edges between two segments in D can be omitted if no two segments cross.

Given a layer L_i, we partition the layer in the following way:

$$S_{i_0}, v_{i_0}, S_{i_1}, v_{i_1}, S_{i_2}, v_{i_2}, \ldots, S_{i_{n_{i-1}}}, v_{i_{n_{i-1}}}, S_{i_{n_i}}.$$

The list S_{i_k} contains the segments which are between vertices $v_{i_{k-1}}$ and v_{i_k} for $1 \leq k \leq n_i - 1$, S_{i_0} contains the segments before v_{i_0} and $S_{i_{n_i}}$ the segments after $v_{i_{n_{i-1}}}$. We denote the first element of a non-empty list S as $head(S)$ and the last element as $tail(S)$. Furthermore, let v be a vertex in V_S. We denote with $s(v)$ the segment to which v is incident if v is a p- or q-vertex, otherwise $s(v) = v$.

Definition 1. *Given a directed acyclic graph $G = (V, E)$ and a sparse layer ordering in which no two segments cross. The sparse compaction graph (N, A) of the sparse normalized graph $G_S = (V_S, E_S)$ of G is defined as:*

$$\begin{aligned} N &= \{V_S \setminus \{v : v \textit{ is p- or q-vertex}\}\} \cup \{s_e : s_e \textit{ is segment of } e \in E\} \\ A &= \{(s(v_{i_{j-1}}), s(v_{i_j})) : 1 \leq i \leq h,\ 1 \leq j \leq n_i - 1,\ S_{i_j} = \emptyset\} \cup \\ &\quad \{(s(tail(S_{i_j})), s(v_{i_j})) : 1 \leq i \leq h,\ 0 \leq j \leq n_i - 1,\ S_{i_j} \neq \emptyset\} \cup \\ &\quad \{(s(v_{i_{j-1}}), s(head(S_{i_j}))) : 1 \leq i \leq h,\ 1 \leq j \leq n_i,\ S_{i_j} \neq \emptyset\} \end{aligned}$$

If we look at two consecutive layers L_n and L_s of a sparse normalized graph we have the following properties:

P1: A segment s_e in L_n is either also in L_s or the adjacent q-vertex q_e is in L_s.
P2: A segment s_e in L_s is either also in L_n or the adjacent p-vertex p_e is in L_n.

Theorem 1. *The ordering $<$ induced by the sparse compaction graph (N, A) of a sparse normalized graph $G_S = (V_S, E_S)$ defines a sparse layer ordering. The compaction graph (N, A) has linear size with respect to G.*

Our new approach is now as follows: In the first phase we create a sparse normalization of the input graph. In the second phase we perform crossing minimization on the sparse normalization. In the third phase we take the resulting sparse compaction graph and perform a coordinate assignment in linear time using an approach similar to the one described in [4]. It remains to show how we can perform crossing minimization on a sparse normalization efficiently.

3 Efficient Crossing Reduction

In this section we present an algorithm which performs crossing minimization using the barycenter or median heuristic on a sparse normalization. The output is a sparse compaction graph which induces a sparse layer ordering with the same number of crossings as these heuristics would produce for a normalization. For our algorithm it is not important which strategy we choose as long as it conforms to some rules.

Definition 2. *A measure m defines for each vertex v in a layer L_{i+1} a non-negative value $m(v)$. If v has only one neighbor w in L_i, then $m(v) = pos(w)$, where $pos(w)$ is the position of w in layer L_i.*

Clearly the barycenter and median heuristic define such a measure.

Lemma 1. *Using such a measure m there are no segments crossing each other.*

Proof. A segment represents a chain of dummy vertices. Each dummy vertex v on a layer L_i has exactly one neighbor w in layer L_{i-1}. Hence when we use a measure m then $m(v) = pos(w)$. Thus two segments never change their relative ordering and thus never produce a crossing with each other. □

3.1 2-Layer Crossing Minimization

The input of our two-layer crossing minimization algorithm is an *alternating layer* L_i and the sparse compaction graph for the layers $L_1, \ldots, L_i$. An alternating layer consists of an alternating sequence of vertices and containers, where each container represents a maximal sequence of segments. The output is an alternating layer L_{i+1} and the sparse compaction graph for $L_1, \ldots, L_{i+1}$, in which the vertices and segments are ordered by some measure. Note that the representation of layer L_i will be lost, since the containers are reused for layer L_{i+1}.

The containers correspond to the lists S of the previous section. The segments in the container are ordered. The data structure implementing the container must support the following operations:

- $\mathbf{S} = \mathbf{create}()$: Creates an empty container S.
- $\mathbf{append}(\mathbf{S}, \mathbf{s})$: Appends segment s to the end of container S.
- $\mathbf{join}(\mathbf{S_1}, \mathbf{S_2})$: Appends all elements of container S_2 to container S_1.
- $(\mathbf{S_1}, \mathbf{S_2}) = \mathbf{split}(\mathbf{S}, \mathbf{s})$: Split container S at segment s into containers S_1 and S_2. All elements less than s are in container S_1 and those who are greater than s in S_2. Element s is neither in S_1 nor S_2.
- $(\mathbf{S_1}, \mathbf{S_2}) = \mathbf{split}(\mathbf{S}, \mathbf{k})$: Split container S at position k. The first k elements in container S are in S_1 and the remainder in S_2.
- $\mathbf{size}(\mathbf{S})$: Returns the number of elements in container S.

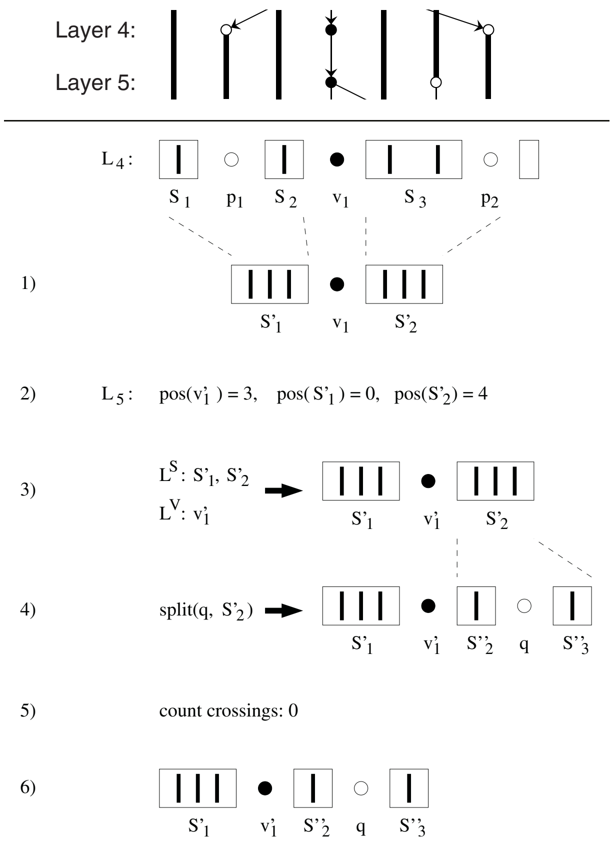

Fig. 2. The six steps applied to layers 4 and 5 from figure 1.

Our algorithm *Crossing_Minimization*(L_i, L_{i+1}) consists of six steps:

- In the first step we append the segment $s(v)$ for each p-vertex v in layer L_i to the container preceding v. Then we join this container with the succeeding container. The result is again an alternating layer (p-vertices are omitted).
- In the second step we compute the measure values for the elements in L_{i+1}. First we assign a position value $pos(v_{i_j})$ to all vertices v_{i_j} in L_i. $pos(v_{i_0}) = size(S_{i_0})$ and $pos(v_{i_j}) = pos(v_{i_{j-1}}) + size(S_{i_j}) + 1$. Note that the pos values are the same as they would be in the median or barycenter heuristic if each segment was represented as dummy vertex. Each non-empty container S_{i_j} has pos value $pos(v_{i_j - 1}) + 1$. If container S_{i_0} is non-empty it has pos value 0. Now we assign the measure to all non-q-vertices and containers in L_{i+1}. Recall that the measure of a container is its old position.
- In the third step we calculate an initial ordering of L_{i+1}. We sort all non-q-vertices in L_{i+1} according to their measure in a list L^V. We do the same for the containers and store them in a list L^S. Then we merge these two sorted lists in the following way:

```
if $m(head(L^V)) \leq pos(head(L^S))$ then $v = pop(L^V)$, $append(L_{i+1}, v)$
if $m(head(L^V)) \geq (pos(head(L^S)) + size(head(L^S)) - 1)$
    then $S = pop(L^S)$, $append(L_{i+1}, S)$
else $S = pop(L^S)$, $v = pop(L^V)$, $k = \lceil m(v) - pos(S) \rceil$, $(S_1, S_2) = split(S, k)$,
    $append(L_{i+1}, S_1)$, $append(L_{i+1}, v)$, $pos(S_2) = pos(S) + k$, $push(L^S, S_2)$.
```

- In the fourth step we place the q-vertices according to the position of their segment. We do this by calling $split(s(v))$ for all q vertices v in layer L_{i+1}.
- In the fifth step we perform cross counting according to the scheme proposed by Barth et al. Using the $size(S)$ operation, we put appropriate weights on the container S, such that the number of segments in the container can be taken into account without any loss of performance.
- In the sixth step we perform a scan on L_{i+1} and insert empty containers between two consecutive vertices, and call $join(S_1, S_2)$ on two consecutive containers in the list. This ensures that L_{i+1} is an alternating layer.

Finally we create the edges in the sparse compaction graph for layer L_{i+1}.

3.2 The Overall Algorithm

The first and the last layer never contain segments because of property P1 and P2. Therefore when we perform a sweep or reverse sweep it is easy to create the initial alternating layer. During the reverse sweeps we simply have to take the former p-vertices as q-vertices and vice versa and apply the 2-layer crossing minimization algorithm of the previous section.

There are no other changes to the original Sugiyama approach except for the different calculation of the measure m for all vertices in a layer, the normalization of the layer lists such that the lists are alternating, and the modified counting scheme for crossings. We summarize this section in the following theorem.

Theorem 2. *The approach described above is equivalent to traditional crossing reduction.*

4 An Efficient Data Structure

Let n denote the maximal number of elements in a container. To be competitive, we need a data structure that supports append, split, join and size operations in $O(\log n)$. Thus we use splay trees, a data structure developed by Sleator und Tarjan [16]. Splay trees are self-adjusting binary search trees, which are easy to implement because the tree is allowed to become unbalanced and we need not keep balance information. Nevertheless we can perform all required operations in $O(\log n)$ amortized time. A single operation might cost $O(n)$ but k consecutive operations starting from an empty tree take $O(k \log n)$ time.

The basic operation on a splay tree is called a ‘splay’. Splaying node x makes x the root of the tree by a series of special rotations. We use splay trees to represent containers. So we have to implement the container operations.

- **append$(\mathbf{S}, \mathbf{s})$:** We search the rightmost element in the tree (last element in the container) by going from the root down taking always the right child. Now, we insert s as the right child of the rightmost element and then splay s. The append operation is performed once for each p-vertex.
- **join$(\mathbf{S_1}, \mathbf{S_2})$:** To join two containers, we search the rightmost element of S_1, splay it and then make S_2 to the right child of it. This operation can only be invoked by an append operation or during the normalization of a layer list. Thus, it is invoked $O(|V| + |E|)$ times.
- **size$(\mathbf{S})$:** While performing the rotations we have to update the size information. Therefore each node knows the size of the subtree rooted by it. So we can maintain the correct size at no extra cost.
- **split$(\mathbf{S}, \mathbf{s})$:** First we have to search s in the container. We can not perform a conventional tree search because the elements have only an implicit ordering (their container position) which is not stored by the element. To avoid a search operation, we store a pointer to s in the corresponding p-vertex (this split operation is only used when we are processing the q-vertex layer and the q-vertex knows its corresponding p-vertex). So we just have to splay s and then take its left and its right child as root for the resulting lists. The split operation is performed once for each q-vertex.
- **split$(\mathbf{S}, \mathbf{k})$:** First we have to search the element at position k. We use a conventional binary tree search. Let $p(x)$ denote the parent of x and $l(x)$ ($r(x)$) the left (right) child of x. The positions are computed by the following formula: $pos(x) = pos(p(x)) + size(l(x)) + 1$, if x is a right child and $pos(x) = pos(p(x)) - size(r(x)) - 1$ if x is a left child. If x is the root then $pos(x) = size(l(x)) + 1$. After we have found the element at position k, we just splay it and then take its right child as root for the second list. This split operation is performed at most once for each common vertex.

Theorem 3. *[16] A sequence of k arbitrary update operations on a collection of initially empty splay trees takes $O(k + \sum_{j=1}^{k} \log n_j)$ time, where n_j is the number of items in the tree or trees involved in operation j.*

The update operations include insert, join and split operations; 'append' is a special case of the insert operation and the size operation does not change the data structure. Each new iteration starts with empty containers and there are at most $O(|E|)$ elements. Thus we have an overall cost of $O((|V| + |E|) \log |E|)$.

5 Conclusion: Complexity and Practical Behaviour

We have given a new technique that leads to a drastic reduction of the complexity of the important algorithm of Sugiyama for automatic graph drawing. We close with some remarks on the complexity of the algorithm. We first do the normalization of the graph by introducing at most $O(|E|)$ new vertices and edges. Then we perform the layer-by-layer sweep with the modified two-layer crossing minimization procedure. Using the splay-tree data structure as well as the cross-counting scheme by Barth et al., we can ensure that each crossing minimization step can be executed in time $O(n \log n)$ where n denotes the number of vertices and edges involved in this step. Summed up over all layers, the complexity remains $O((|V| + |E|) \log |E|)$. The coordinate assignment is performed in time $O(|V| + |E|)$ using a variant of the algorithm of Brandes and Köpf [4]. Our approach favourably compares to the previous implementations of Sugiyama's algorithm where the complexity might be quadratic in the size of the graph.

We implemented our approach in Java using the yFiles library[20]. We made some preliminary tests and compared our approach to the results achieved with other layout tools using Sugiyama's algorithm. All experiments have been performed on a Pentium IV System with 1.5 GHz and 512 MB main memory running Redhat Linux 9. For our measurement we used the following types of graphs:

- **Long Edge Graphs:** These graphs have many long edges. They have $n/2$ vertical vertices $v_1, \ldots, v_{n/2}$ and $n/2$ horizontal vertices $h_1, \ldots, h_{n/2}$. The vertical vertices are connected by edges (v_i, v_{i+1}) for $1 \le i \le n/2 - 1$. The graph also have edges (v_i, h_j) for $1 \le i, j \le n/2$.
- **Random Graphs:** They have n vertices and $2.5n$ random edges.

We run the experiments for VCG [18], Dot [11] and our new approach. We also added an algorithm 'Traditional' which uses the same code as our new approach but insert the traditional dummy vertices. Table 1 shows the time taken by the cross counting step, which is given in milliseconds/iteration as well as the number of dummy vertices in the normalized graph, when applying the network simplex for layer assignment. The network simplex gives a solution which minimize the edge length. So the results for other methods are even worse.

Our approach achieved significant improvements in running time for both graph types. This is due to the enormous increase of the number of dummy vertices in the common approach. The results show that our improvements are

Table 1. Experimental results for the long edge graphs and the random graphs.

Size (n)	Time (ms/iter)*				#Dummy vertices	
(long edges)	VCG	Dot	Traditional	New	Common	New
60	146	499	116	19	13050	1710
80	455	2852	306	42	31200	3080
100	1040	13346	658	69	61250	4850
120	2060	42414	1219	98	106200	7020
140	3702	103327	2020	158	169050	9590

Size (n)	Time (ms/iter)*				#Dummy vertices	
(random)	VCG	Dot	Traditional	New	Common	New
100	11	33	16	4	2725	295
200	40	275	60	9	9486	596
500	311	4404	416	29	49203	1485
1000	2978	60783	2643	72	233486	3001
2000	14419	n/a**	n/a**	190	796653	6019

* results are averaged over 10 passes ** not enough memory

also relevant for practice, even if the number of dummy vertices is usually far less than $|V| \cdot |E|$ there. The number of crossings in our new approach is comparable with the number computed by the other tools. The slight differences are based on the fact, that each implementation has its own refinements (e.g. how to handle nodes having the same median weight). Only Dot has noticeable less crossings but is therefor very slow. This is possibly due to an additional optimization method. Our improvements made it possible to layout graphs for which this was formerly not possible because of the enormous memory consumption of Sugiyama's algorithm. Our approach has just a linear memory consumption.

References

1. D. Auber: *Tulip – A Huge Graph Visualization Framework.* In: Jünger, Mutzel (eds.): Graph Drawing Software, Springer-Verlag, pp. 105–126, 2003.
2. W. Barth, M. Jünger and P. Mutzel: *Simple and Efficient Bilayer Cross Counting.* In: Proceedings of Graph Drawing 2002, Springer LNCS 2528, pp. 130–141, 2002.
3. O. Bastert and C. Matuszewski: *Layered drawings of digraphs.* In: Kaufmann, Wagner (eds.): Drawing Graphs: Methods and Models, Springer LNCS 2025, pp. 104–139, 2001.
4. U. Brandes and B. Köpf: *Fast and Simple Horizontal Coordinate Assignment.* In: Proceedings of Graph Drawing 2001, Springer LNCS 2265, pp. 31–44, 2001.
5. E. Coffman and R. Graham: *Optimal scheduling for two processor systems.* Acta Informatica, 1: 200–213, 1972.
6. G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis: *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.
7. P. Eades and D. Kelly: *Heuristics for Reducing Crossings in 2-Layered Networks.* Ars Combin., 21.A: 89–98, 1986.
8. P. Eades and N. Wormald: *Edge crossings in drawings of bipartite graphs.* Algorithmica, 11(4): 379–403, 1994.
9. A. Frick: *Upper bounds on the number of hidden nodes in Sugiyama's algorithm.* In: Proceedings of Graph Drawing 1996, Springer LNCS 1190, pp. 169–183, 1996.
10. E. Gansner, E. Koutsofios, S. North and K. Vo: *A technique for drawing directed graphs.* In: IEEE Transactions on Software Engineering, 19(3): 214–229, 1993.
11. Graphviz – open source graph drawing software: *http://www.research.att.com/sw/tools/graphviz/.*
12. R. M. Karp: *Reducibility among Combinatorial Problems.* In: Miller R. E., Thatcher J. W. (eds.): Complexity of Computer Computations, Plenum Press, New York, pp. 85–103, 1972.
13. C. Matuszewski, R. Schönfeld and P. Molitor: *Using sifting of k-layer straightline crossing minimization.* In: Proceedings of the 7th Symposium on Graph Drawing (GD'99), Springer LNCS 1731, pp. 217–224, 1999.
14. N. Nikolov and P. Healy: *How to layer a directed Acyclic Graph.* In: Proceedings of the 9th Symposium on Graph Drawing (GD'01), Springer LNCS 2265, pp. 16–30, 2002.
15. G. Sander: *Graph layout through the VCG tool.* In: Proceedings of Graph Drawing 1994, Springer LNCS 894, pp. 194–205, 1995.
16. D. Sleator and R. E. Tarjan: *Self-Adjusting Binary Search Trees.* In: Journal of the ACM, 3: 652–686, 1985.

17. K. Sugiyama, S. Tagawa and M. Toda: *Methods for visual understanding of hierarchical system structures*. In: IEEE Transactions on Systems, Man and Cybernetics, SMC-11(2): 109–125, 1981.
18. VCG – Visualization of Compiler Graphs: *http://rw4.cs.uni-sb.de/users/sander/ html/gsvcg1.html*.
19. V. Waddle and A. Malhotra: *An E log E line crossing algorithm for levelled graphs*. In: Proceedings of the 7th Symposium on Graph Drawing (GD'99), Springer LNCS 1731, pp. 59–70, 1999.
20. yFiles – a Java Graph Layout and Visualization Library: *http://www.yworks.com*.