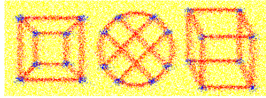


Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 9, no. 3, pp. 305–325 (2005)

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

Markus Eiglsperger

Zühlke Engineering AG
8952 Schlieren, Switzerland

Martin Siebenhaller

WSI für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
siebenha@informatik.uni-tuebingen.de

Michael Kaufmann

WSI für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
mk@informatik.uni-tuebingen.de

Abstract

Sugiyama's algorithm for layered graph drawing is very popular and commonly used in practical software. The extensive use of dummy vertices to break long edges between non-adjacent layers often leads to unsatisfying 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 without increasing the number of crossings. We 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.

Article Type	Communicated by	Submitted	Revised
Regular paper	E. R. Gansner and J. Pach	November 2004	July 2005

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.

1 Introduction

Most approaches for drawing directed graphs used in practice follow the 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. This is commonly done by a layer-by-layer sweep where in each step the number of edge crossings is minimized for a pair of adjacent layers. The fourth phase, called *Horizontal Coordinate Assignment*, calculates an x-coordinate for each vertex. Finally the dummy vertices introduced after the layer assignment are removed and replaced by bends.

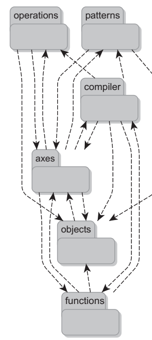
Unfortunately, almost all problems occurring during the phases of this approach are NP-hard: Feedback-arc Set [14], Precedence Constrained Multiprocessor Scheduling [5], 2-layer Crossing Minimization [8], Optimal Linear Arrangement [11], etc. Nevertheless, for all these problems appropriate heuristics 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. Two examples for layered graph drawings are given in Figure 1.

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. In our description we can therefore skip the first step in which the given graph G is made acyclic.

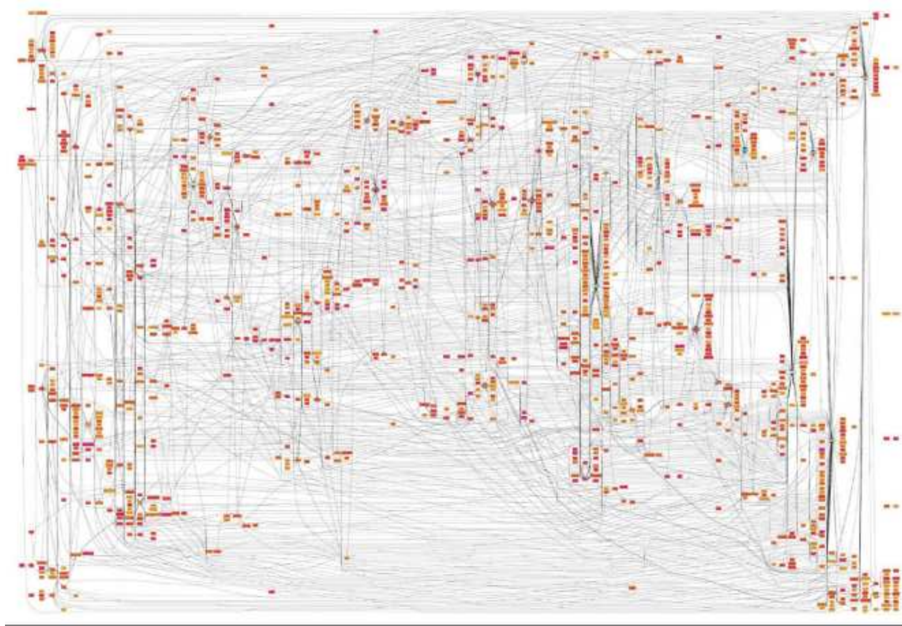
1.1 Layer Assignment and Normalization

We assume that $G = (V, E)$ is a directed acyclic graph. 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 edges $e = (v, w)$ with $v \in L_i$ and $w \in L_j$ holds $i < j$. The span of an edge e is $j - i$. The number of vertices in a layer L_i is denoted with n_i. In a layered drawing, all vertices $v \in L_i$ are drawn on a horizontal line (same y-coordinate). So the layer assignment step assigns each vertex $v \in V$ a 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



(a)



(b)

Figure 1: A small and a large example for a layered drawing of a directed graph. The small example 1(a) shows a UML package diagram. The large example 1(b) is taken from the Atlas of Cyberspaces (http://www.cybergeography.org/atlas/topology.html), and shows social relationships within a textual virtual space. It stems from the cobot project (http://www.cc.gatech.edu/fac/Charles.Isbell/projects/cobot/).

L_k. This process is called *normalization* and the result is 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 edge length is minimized. Thus the number of dummy vertices is minimized, too. An estimation of this number has been given by Frick [9]. 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 [13, 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 number of 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 $i = 2, .., h$. After we have processed the bottommost layer, we reverse the sweep direction and go from bottom to top. These steps are repeated until no further crossings can be eliminated for a certain number of iterations.

Different heuristics have been proposed to attack the one-sided two-layer crossing minimization problem [3, 6]:

- **Barycenter Heuristic [17]**
  The position of a vertex v in list L_i depends on the position of its adjacent vertices in list L_{i-1}. First a measure is calculated for each vertex v in L_i. The measure of a vertex v is the barycenter (average) of the positions of its neighbors in the above layer. We get the positions of the vertices in L_i by sorting them according to their measure.

- **Median Heuristic [8]**
  This heuristic is similar to the barycenter heuristic, except that the measure of a vertex v of layer L_i is the median of the positions of its neighbors in layer L_{i-1}.

- **Greedy Switch Heuristic [7]**
  Starting with a certain order of the vertices in L_i, consecutive vertices are

exchanged if the exchange reduces the number of crossings. In contrast to the near-linear time complexity of the above heuristics the time complexity of this heuristic is quadratic in the number of vertices. It is mainly used as a local optimization of the median or barycenter heuristic within a layer sweep iteration [10].

Only a few provable results on the quality of the above heuristics are known:

- The barycenter as well as the median heuristic give a solution without crossings if one exists [6].
- The median heuristic produces at most three times more crossings than necessary [8].

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 and Malhotra [19]. Barth et al. [2] gave a much simpler description of the algorithm with the same running time.

It works as follows: Let L_i and L_{i+1} be two adjacent layers with layer ordering $v_1, .., v_p$ and $w_1, .., w_q$ respectively. The edges between both layers are sorted lexicographically such that $(v_i, w_j) < (v_k, w_l)$ if and only if $i < k$ or $i = k$ and $j < l$. Let $e_1, .., e_r$ be the lexicographically sorted edge sequence, and j_m the index of the target vertex of e_m. An inversion in the sequence $j_1, .., j_r$ is a pair j_k, j_l with $k < l$ and $j_k > j_l$. Each inversion corresponds to an edge crossing between both layers. The number of inversions is counted by means of an efficient data structure, called the accumulator tree T. The data structure can easily be extended to support cross counting with weighted edges, which will be relevant later.

1.3 Horizontal Coordinate Assignment

The horizontal coordinate assignment computes the x-coordinate for each vertex with respect to the layer ordering computed during 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." The failure of the second objective can produce many unnecessary bends which results in a "spaghetti" effect and reduces the readability.

Gansner et al. [10] model the horizontal coordinate assignment problem as a linear program:

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

subject to

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

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. If Ω is chosen carefully the spaghetti effect could be limited. The linear program can be interpreted as a rank assignment problem on a compaction graph $G_a = (V_N, \{(a, b) : a, b \text{ consecutive in } L_i,\ 1 \leq i \leq h\})$ 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].

Another approach is to use the *linear segments model*, 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] is a longest path based heuristic for the horizontal coordinate assignment of directed acyclic graphs. It produces vertical inner segments and nice balanced drawings in linear time. First it tries to align a vertex with either its median upper or its median lower neighbor. Aligned vertices share the same vertex in the compaction graph and thus get the same x-coordinate. There are three kinds of alignment conflicts, type 0 conflicts arise between two non-inner segments (a non-inner segment has at least one non-dummy vertex endpoint), type 1 conflicts arise between a non-inner segment and an inner segment and type 2 conflicts between two inner segments. Type 1 conflicts are always solved in favor of the inner segment.

Regardless if the vertices are aligned with their median upper or median lower neighbor, alignment conflicts can be solved either in a leftmost or a rightmost fashion. So there are four possible combinations for aligning the vertices. For each combination the horizontal coordinates are calculated by a longest-path algorithm. Finally, the four resulting coordinate assignments are combined to get a balanced drawing.

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 system described in [1]. Unfortunately, no details about the theoretical or practical performance, or the implementation are given, and a comparison with the quality of the approaches commonly used has not been described. However, in this approach, only the proper edges are

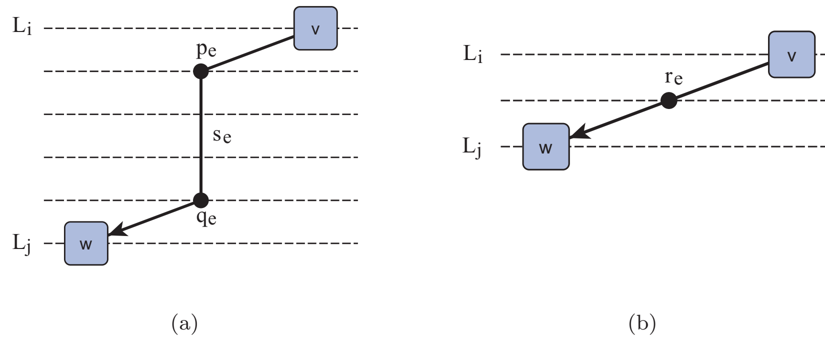
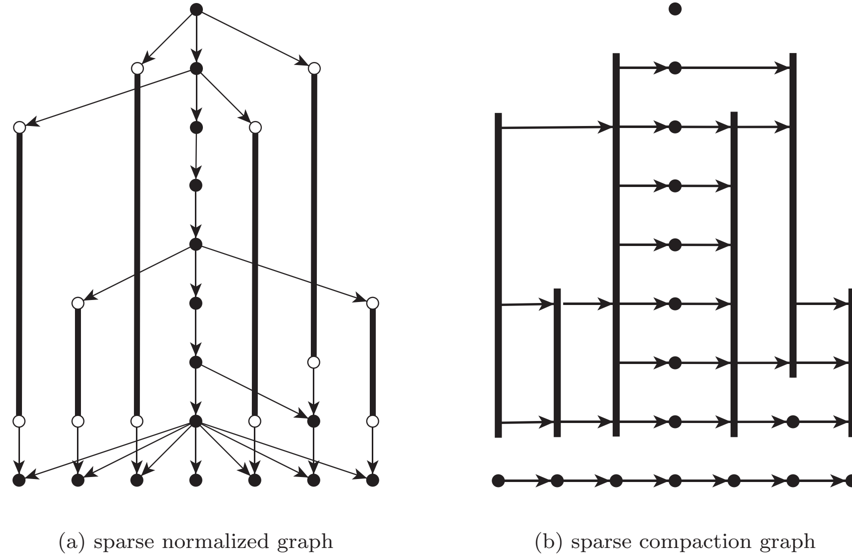


Figure 2: Sparse normalization: In the left figure edge $e = (v, w)$ ($span(e) > 2$) is split into three segments (v, p_e), $s_e = (p_e, q_e)$ and (q_e, w). The first and the last edge are proper, s_e is drawn vertical. In the right figure edge e ($span(e) = 2$) is split into two segments (v, r_e) and (r_e, w).

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 the vertical edge s_e, called the *segment* of e, is not necessarily proper (see Figure 2(a)). If $|j - i| = 2$ we insert a single dummy vertex r_e at layer L_{i+1} as well as two edges (v, r_e) and (r_e, w) (see Figure 2(b)). Single dummy vertices are treated like common vertices later on. We call this transformation *sparse normalization* and the result the *sparse normalized graph* $G_S = (V_S, E_S)$ (see Figure 3(a)). The size of the sparse normalized graph is linear with respect to the size of the input graph. A similar transformation is used in the horizontal coordinate assignment approach of [4], where vertically aligned vertices are combined into blocks.



(a) sparse normalized graph (b) sparse compaction graph

Figure 3: The left figure shows a sparse normalized graph. Thick lines denote the segments. The right figure shows the corresponding sparse compaction 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*. When we draw a graph G in the linear segments model, there is a correlation 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 L_i and are consecutive. 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 Section 1.3. The 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 therefore is that the edges between two segments in L_i can be omitted if no two segments cross.

Given a layer L_i of a sparse normalized graph, 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_{i_k} as $head(S_{i_k})$ and the last element as $tail(S_{i_k})$. Furthermore we denote by $s(v)$ the segment to which $v \in V_S$ 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:*

$$\begin{aligned} N &= V \cup \{r_e \colon r_e \text{ single dummy vertex of } e \in E\} \cup \{s_e \colon s_e \text{ segment of } e \in E\} \\ A &= \{(s(v_{i_{j-1}}), s(v_{i_j})) : 1 \leq i \leq h,\ 1 \leq j \leq n_i - 1,\ S_{i_j} = \emptyset\} \cup \\ &\quad \{(tail(S_{i_j}), s(v_{i_j})) : 1 \leq i \leq h,\ 0 \leq j \leq n_i - 1,\ S_{i_j} \neq \emptyset\} \cup \\ &\quad \{(s(v_{i_{j-1}}), head(S_{i_j})) : 1 \leq i \leq h,\ 1 \leq j \leq n_i,\ S_{i_j} \neq \emptyset\} \end{aligned}$$

An example of a sparse compaction graph is given in Figure 3(b).

If we look at two consecutive layers L_i and L_{i+1} of a sparse normalized graph we have the following properties:

P1: A segment s_e in L_i is either also in L_{i+1} or the corresponding q-vertex q_e is in L_{i+1}.

P2: A segment s_e in L_{i+1} is either also in L_i or the corresponding p-vertex p_e is in L_i.

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.*

Proof: In the sparse compaction graph, each edge is induced by a vertex in V_S. Each vertex in V_S induces at most 2 edges. Therefore the number of edges is at most $2|V_S|$. Since there are at most $2|V_S|$ lists S the number of vertices in the compaction graph is linear with respect to G. We prove that the compaction graph yields a total layer ordering by induction on the layers. In the first layer there are no segments and the compaction graph of the sparse normalized graph is identical to the compaction graph of the normalized graph for this layer, which defines a total layering. Assume that the compaction graph defines a total layer ordering for all layers above L_i. We show that for two consecutive segments s_1 and s_2 in a list S_{i_j} there is a path from s_1 to s_2 using vertices and segments defined in the layers above L_i. From this fact it follows that the compaction graph defines a total ordering for layer L_i. Let $j < i$ be the layer with the largest number such that s_1 and s_2 are no longer consecutive in a list S_{j_k}. We show that there are only vertices of V_S between s_1 and s_2 in L_j. If there was a segment s_e between them, then this segment would end in a layer k with $j < k < i$, otherwise s_e would cross either s_1 or s_2, which is a contradiction to the fact that no pair of segments cross. But then, using property P1 in layer $k+1$, there is a vertex q_e between s_1 and s_2, otherwise there would be again a pair of crossing segments. But this is a contradiction to the definition of j which

is the greatest layer in which s_1 and s_2 are not consecutive. Because there are only vertices of V between s_1 and s_2 in layer L_j, there is a path between s_1 and s_2 according to the definition of the compaction graph. □

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, which is the topic of the next section.

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 two containers S_1 and S_2. All elements less than s are stored 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 of S are stored in S_1 and the remainder in S_2.
- $\mathbf{size}(\mathbf{S})$: Returns the number of elements in container S.

Our algorithm *Crossing_Minimization(L_i, L_{i+1})* is divided into six steps (see Figure 4):

- 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}. The initial containers in L_{i+1} are the resulting containers of the first step. Recall that the measure of a container in L_{i+1} is its position in L_i.
- 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. We use the following operations on these sorted lists:
  - $\mathbf{l} = \mathbf{pop}(\mathbf{L})$: Removes the first element l from list L and returns it.
  - $\mathbf{push}(\mathbf{L}, \mathbf{l})$: Inserts element l at the head of list L.

  We merge both lists in the following way:

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

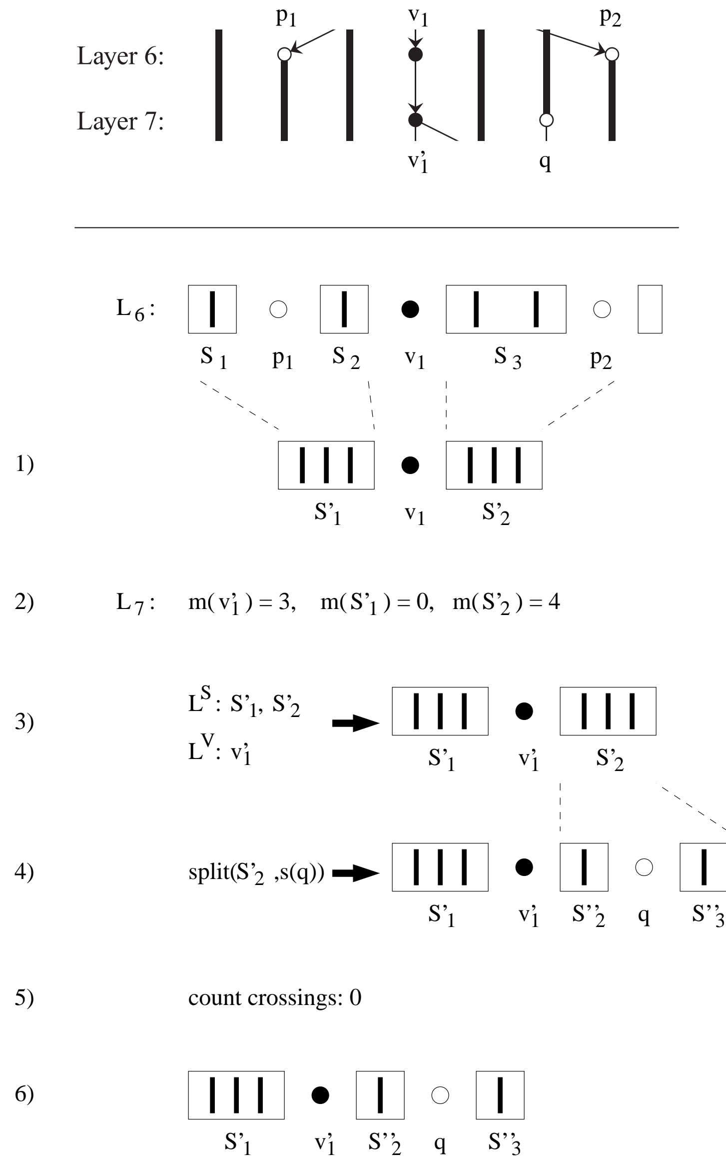


Figure 4: The six steps applied to layers 6 and 7 from Figure 3(a).

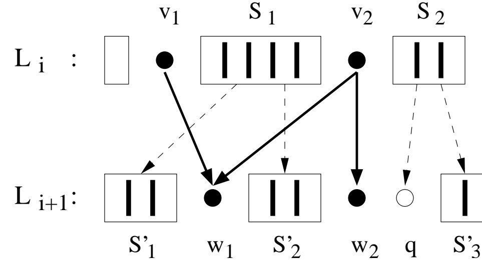


Figure 5: Example for our modified cross counting. Container S_1 is split into containers S'_1 and S'_2, container S_2 into a q-vertex and container S'_3. Dashed edges represent virtual edges. Besides the common edges (v_1, w_1), (v_2, w_1) and (v_2, w_2) with $weight = 1$ we have the virtual edges (S_1, S'_1) and (S_1, S'_2) both with $weight = 2$ as well as (S_2, q) and (S_2, S'_3) both with $weight = 1$. So the crossing between (v_1, w_1) and (S_1, S'_1) as well as the crossing between (v_2, w_1) and (S_1, S'_2) counts as 2 crossings.

- In the fourth step we place each q-vertex v of L_{i+1} according to the position of its corresponding segment $s(v)$. We do this by calling $split(S, s(v))$ for each q-vertex v in layer L_{i+1} and placing v between the resulting containers (S denotes the container that includes $s(v)$).

- In the fifth step we perform cross counting according to the scheme proposed by Barth et al (see Section 1.2). During the cross counting step between layer L_i and L_{i+1} we therefore consider all layer elements as vertices. Beside the common edges between both layers, we also have to handle *virtual edges*, which are imaginary edges between a container element in L_i and the resulting container elements or q-vertices in L_{i+1} (see Figure 5). In terms of the common approach each virtual edge represents at least one edge between two dummy vertices. The number of represented edges is equal to the size of the container element in L_{i+1}. We have to consider this fact to get the right number of edge crossings. We therefore introduce edge weights. The *weight* of a virtual edge ending with a container element S is equal to $size(S)$. The *weight* of the other edges is one. So a crossing between two edges e_1 and e_2 counts as $weight(e_1) \cdot weight(e_2)$ crossings.

- 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

In the crossing reduction phase we perform a layer-by-layer sweep on the sparse normalization and apply the 2-layer crossing minimization described in this section. During a reverse sweep we simply have to take the former p-vertices as q-vertices and vice versa. 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.

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.

For the horizontal coordinate assignment we need just minor modifications to the approach of Brandes and Köpf (see Section 1.3). In our model there are no type 2 conflicts because we do not have crossing inner segments. We mark type 1 conflicts during the cross counting step. Since the conflicts are resolved in favor of the inner segments, we simply have to mark edges that cross a virtual edge. Type 0 conflicts can be resolved as described in [4]. There are no further changes.

We summarize this section in the following theorem:

Theorem 2 *Using a measure m, the approach described above gives the same result as the traditional crossing reduction.*

4 An Efficient Data Structure

When we use doubly linked lists to represent the containers, we are able to perform the append, size and join operation in time $O(1)$. Although we store a pointer to the split element, we can not perform the split operation in time $O(1)$, since we have to update the size of the resulting containers. So we need $O(n)$ for splitting, where n denotes 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 $O(\log n)$. A standard binary search tree (not balanced) also requires $O(n)$. Thus we use splay trees, a data structure developed by Sleator and 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.

- $\mathbf{append(S, s)}$: 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 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.

- **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 containers. The split operation is performed once for each q-vertex.

- **size**$(\mathbf{S})$: When we perform a split operation we want to update the size of the resulting containers in $O(1)$. Therefore each node knows the size of the subtree rooted by it. While performing the rotations we can update the size information at no extra cost.

- **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 container. This split operation is performed at most once for each common vertex.

All the above operations except of the size operation are based on splaying. In [16] the following theorem is proved:

Theorem 3 *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. The append operation 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 Complexity and Practical Behavior

We have given a new technique that leads to a noticeable reduction of the complexity of the important algorithm of Sugiyama for layered graph drawing. We close with some remarks on the complexity of the algorithm. We first normalize 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. [2], 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 the algorithm of Brandes and Köpf [4]. The memory bound remains linear in the size of the input graph. Our approach favorably compares to the previous implementations of Sugiyama's algorithm where the complexity might be quadratic in the size of the input 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 $1 \leq i \leq n/2 - 1$. Furthermore there are edges (v_i, h_j) for $1 \leq i, j \leq n/2$.
- **random graphs:** These graphs are connected and have n vertices and $2.5n$ random edges.

We ran the experiments for VCG [18], Dot [12] and our new approach. We also added an algorithm called *Traditional*, which uses the same code as our new approach but inserts the traditional dummy vertices. Tables 1 and 2 show the time taken by the crossing minimization 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. The shown results are averaged over 10 passes.

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 also relevant in practice, even if the number of dummy vertices is usually far less than $|V| \cdot |E|$ there. Only the new approach was able to layout the random graphs with 3000 vertices while the other algorithms ran out of memory. The number of crossings in our new approach is comparable with the number computed by the other tools (see Tables 3 and 4). The slight differences are based on

Size (n)	Time (ms/iter)				#Dummy vertices	
(long edges)	VCG	Dot	Traditional	New	Common	New
40	30	59	29	7	3800	740
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
160	6341	228387	3194	249	252800	12560

Table 1: Experimental results for the long edge graphs.

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
750	1265	22338	1228	44	123074	2259
1000	2978	60783	2643	72	233486	3001
2000	14419	n/a*	n/a*	190	796653	6019
3000	n/a*	n/a*	n/a*	230	1995096	9017

* not enough memory

Table 2: Experimental results for the random graphs.

the fact that each implementation has its own refinements (e.g., how to handle vertices having the same median weight) or uses randomized steps. Only Dot has noticeably fewer crossings, but is therefore very slow. This is possibly due to an additional optimization method. The results of VCG suggests that its implementation is similar to algorithm Traditional.

Our improvements made it possible to layout graphs for which this was formerly not possible because of the high memory consumption of Sugiyama's algorithm. Our approach has instead linear memory consumption.

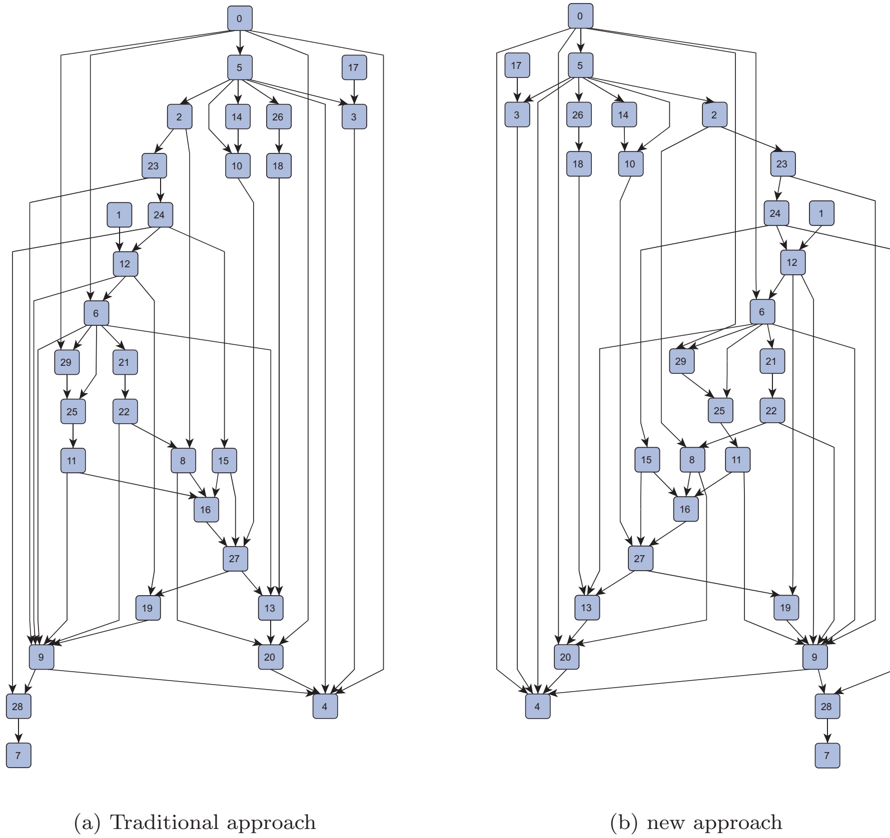
Size (n) (long edges)	#Crossings			
	VCG	Dot	Traditional	New
40	18887	17100	18809	18801
60	97506	91350	97578	97552
80	311969	296400	311242	311319
100	764819	735000	764801	764502
120	1591922	1539900	1590856	1591042
140	2956819	2873850	2956013	2956007
160	5054767	4929600	5053631	5052549

Table 3: Number of crossings for the long edge graphs.

Size (n) (random)	#Crossings			
	VCG	Dot	Traditional	New
100	2264	1296	2002	1998
200	8314	6390	7504	7457
500	49945	39207	46523	47001
750	118485	103340	110045	109537
1000	202371	185680	189517	190455
2000	800221	n/a*	n/a*	735429
3000	n/a*	n/a*	n/a*	1716657

* not enough memory

Table 4: Number of crossings for the random graphs.



(a) Traditional approach

(b) new approach

Figure 6: Layout example: The left figure was drawn with algorithm Traditional using the simplex method for the horizontal coordinate assignment. The right figure was drawn with the new approach which uses the linear segments model. Both layout algorithms used the same layering as well as the median heuristic for crossing minimization.

References

[1] D. Auber. Tulip - a huge graph visualization framework. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 105–126. Springer-Verlag, 2003.

[2] W. Barth, M. Jünger, and P. Mutzel. Simple and efficient bilayer cross counting. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, volume 2528 of *LNCS*, pages 130–141. Springer-Verlag, 2002.

[3] O. Bastert and C. Matuszewski. Layered drawings of digraphs. In M. Kaufmann and D. Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *LNCS*, pages 87–120. Springer-Verlag, 2001.

[4] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 31–44. Springer-Verlag, 2002.

[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 *GD '96: Proceedings of the Symposium on Graph Drawing*, volume 1190 of *LNCS*, pages 169–183. Springer-Verlag, 1997.

[10] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, 1993.

[11] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63. ACM Press, 1974.

[12] Graphviz - open source graph drawing software. `http://www.graphviz.org`.

[13] P. Healy and N. Nikolov. How to layer a directed acyclic graph. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 16–30. Springer-Verlag, 2002.

[14] R. Karp. Reducibility among combinatorical problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[15] G. Sander. Graph layout for applications in compiler construction. *Theor. Comput. Sci.*, 217(2):175–214, 1999.

[16] D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[17] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, February 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 *GD '99: Proceedings of the 7th International Symposium on Graph Drawing*, volume 1731 of *LNCS*, pages 59–71. Springer-Verlag, 1999.

[20] yFiles - a java graph layout and visualization library. `http://www.yworks.com`.