In Search for Efficient Heuristics for Minimum-Width Graph Layering with Consideration of Dummy Nodes

NIKOLA S. NIKOLOV and ALEXANDRE TARASSOV
University of Limerick and IMAGEN Group
and
JÜRGEN BRANKE
University of Karlsruhe

We propose two fast heuristics for solving the NP-hard problem of graph layering with the minimum width and consideration of dummy nodes. Our heuristics can be used at the layer-assignment phase of the Sugiyama method for drawing of directed graphs. We evaluate our heuristics by comparing them to the widely used fast-layering algorithms in an extensive computational study with nearly 6000 input graphs. We also demonstrate how the well-known longest-path and Coffman–Graham algorithms can be used for finding narrow layerings with acceptable aesthetic properties.

Categories and Subject Descriptors: G.1.1 [**Interpolation**]; G.2 [**Approximation**]

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: Layer assignment, layering, layered graphs, hierarchical graph drawing, dummy vertices

1. INTRODUCTION

The rapid development of software engineering and the need to analyze large and complex networks have made *graph drawing* an important area of research in the recent years. The graph drawing techniques find application in visualizing various diagrams, such as call graphs, precedence graphs, data-flow diagrams, ER diagrams, and social and biological networks, etc. In many of those applications it is required to draw a set of objects in a hierarchical relationship. Such sets are modeled by directed acyclic graphs (DAGs), i.e., directed

Work by the first two authors is supported by IRCSET Basic Research Grant SC/2003/161.
Authors' addresses: Nikola S. Nikolov and Alexandre Tarassov, CSIS Department, University of Limerick, Limerick, Ireland and National ICT Australia; email: {alexandre.tarassov,nikola.nikolov}@ul.ie. IMAGEN Group, National ICT Australia, Sydney, Australia. Jürgen Branke, Institute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany; email: branke@aifb-uni-karlsruhe.de.

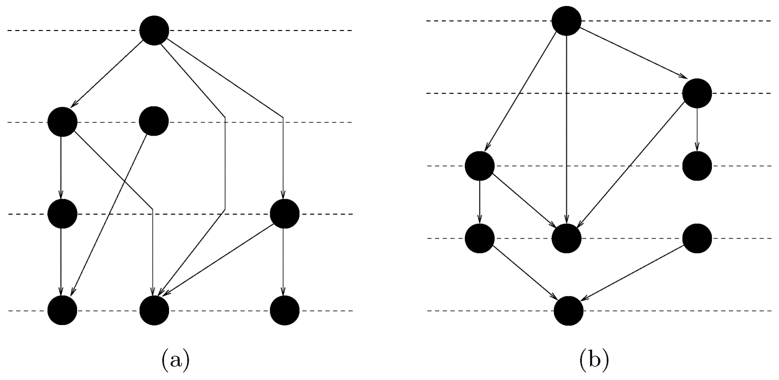


Fig. 1. Two alternative layerings of the same DAG. Each layer occupies a horizontal level marked by a dashed line. All edges point downwards.

graphs without directed cycles and usually drawn by placing the graph nodes on parallel horizontal, concentric, or radial levels with all edges pointing in the same direction.

There have been recognized a few different methods for hierarchical graph drawing. The more recent two are the evolutionary algorithm of Utech et al. [1998] and the magnetic field model introduced by Sugiyama and Misue [1965]. While they are an area of fruitful future research, an earlier method, widely known as the Sugiyama (or STT) method, has received most of the research attention and has become a standard method for hierarchical graph drawing. The STT method is a three-phase algorithmic framework, originally proposed by Sugiyama, Tagawa, and Toda [1981], and also based on work by Warfield [1977] and Carpano [1980]. At its first phase, the nodes of a DAG are placed on horizontal levels and dummy nodes are introduced at places where edges cross levels; at the second phase, the nodes are ordered within each level; and, at the final third phases, the x and y coordinates of all nodes and the eventual edge bends are assigned. The STT method can be employed for drawing any directed graph by reversing the direction of some edges in advance to ensure that there are no directed cycles in the graph and restoring the original directions at the end Eades et al. [1993].

In order to assign DAG nodes to horizontal levels (at the first phase of the STT method), it is necessary to partition the node set into subsets such that nodes connected by a directed path belong to different subsets. In addition, it must be possible to assign integer ranks to the subsets such that for each edge the rank of the subset that contains the target of the edge is less than the rank of the subset that contains its source. Such an ordered partition of the node set of a DAG is known as a *layering* and the corresponding subsets are called *layers*. A DAG with a layering is called a *layered DAG*. Figure 1 gives an example of two alternative layerings of the same DAG with four and five layers, respectively. Algorithms, which partition the node set of a DAG into layers, are known as *layering algorithms*.

If the display area is limited, it is often desired to have a narrow drawing, e.g., that its width is smaller than the width of a computer screen and the user has to navigate only in vertical direction rather than in both directions, in case of a wider drawing. However, layering a graph with minimal width is NP-hard when the contribution of edges crossing a layer is also taken into account [Branke et al. 2002].

This paper extends our previous work on the design of a fast-layering heuristic for layered-graph drawing with minimum width when dummy nodes are taken into account [Tarassov et al. 2004]. In this paper, we extend the evaluation of our previously proposed heuristic `MinWidth` and introduce a new heuristic, `StretchWidth`. We also demonstrate how the layerings found by some well-known fast-layering heuristics can be postprocessed for making them narrower and reducing their dummy-node count. We present an extensive comparison of all discussed layering techniques. To the best of our knowledge, the layering techniques introduced here and in our previous paper are the first successful attempt to solve the minimum-width layering problem by fast heuristics.

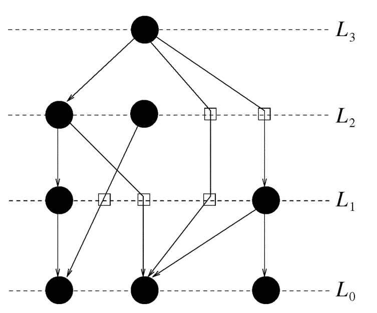
In the next section we formally introduce the terminology related to DAG layering. Then, in Section 3, we briefly present the well-known layering algorithms. The problem of layering with the minimum width is formally defined in Section 4, where we also introduce our heuristics. Section 5 presents the results of the extensive computational study we have conducted to evaluate the performance of the proposed layering heuristics. We draw conclusions from this work in Section 6.

2. MATHEMATICAL PRELIMINARIES

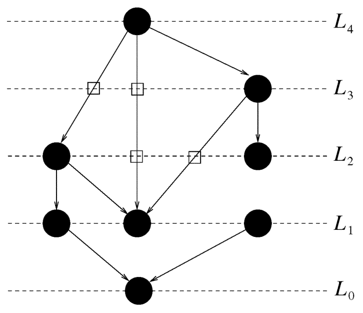
A *directed graph* $G = (V, E)$ consists of a set of nodes V and a set of edges E. Each edge e is associated with an ordered pair of nodes (u, v); u is the *source* of e and v is the *target* of e. We denote this by $e = (u, v)$. We consider only directed graphs where different edges are associated with different node pairs. The *in-degree* $d^-(v)$ of node v is the number of edges with target v and the *out-degree* $d^+(v)$ of v is the number of edges with a source v. We denote the set of all immediate predecessors of node v by $N_G^-(v)$ and the set of all immediate successors of node v by $N_G^+(v)$. That is, $N_G^-(v) = \{u : (u, v) \in E\}$ and $N_G^+(v) = \{u : (v, u) \in E\}$. The k-tuple of edges $p = ((u_1, u_2), (u_2, u_3), \dots, (u_k, u_{k+1}))$ is called a *directed path* from node u_1 to node u_{k+1} with length $k \geq 1$. If $u_1 = u_{k+1}$ then p is a *directed cycle*. In the rest of this work, we consider only directed acyclic graphs (DAGs), i.e., directed graphs without directed cycles.

Let G be a DAG and let $\mathcal{L} = \{L_1, \dots, L_h\}$ be a partition of the node set V into $h \geq 1$ subsets such that if $(u, v) \in E$ with $u \in L_j$ and $v \in L_i$ then $i < j$. $\mathcal{L}$ is called a *layering* of G and the sets $L_1, \dots, L_h$ are called *layers*. A DAG with a layering is called a *layered* DAG. We assume that in a visual representation of a layered DAG all nodes in layer L_i are placed on the horizontal level with a y-coordinate i. Thus, we say that L_j is *above* L_i and L_i is *below* L_j if $i < j$.

Let $l(u, \mathcal{L})$ denote the number of a layer which contains node $u \in V$, i.e., $l(u, \mathcal{L}) = i$ if, and only if, $u \in L_i$. The *span* of edge $e = (u, v)$ in layering $\mathcal{L}$ is then defined as $s(e, \mathcal{L}) = l(u, \mathcal{L}) - l(v, \mathcal{L})$. Clearly, $s(e, \mathcal{L}) \geq 1$ for each $e \in E$; edges with a span greater than 1 are *long edges*. A layering of G is *proper* if $s(e, \mathcal{L}) = 1$ for each $e \in E$, i.e., if there are no long edges. The layering found by a layering algorithm might not be proper because only a small fraction of DAGs



(a) A layering with 4 layers and 5 dummy nodes.



(b) A layering with 5 layers and 4 dummy nodes.

Fig. 2. The drawings from Figure 1 with introduced dummy nodes which subdivide long edges. Dummy nodes are represented by transparent squares.

can be layered properly and also because a proper layering may not satisfy other layering requirements.

In the STT method for drawing DAGs, the node-ordering algorithms applied after the layer-assignment phase assume that their input is a DAG with a proper layering. Thus, if the layering found at the layer-assignment phase is not proper then it must be transformed into a proper one. Normally, this is done by introducing so-called *dummy nodes*, which subdivide long edges (see Figure 2). We refer to the nodes, which are not dummy, as *original nodes*.

It is desirable that the number of dummy nodes is as small as possible because a large number of dummy nodes significantly slows down the node-ordering phase of the STT method. There are also aesthetic reasons for keeping the dummy node count small. A layered DAG with a small dummy node count would also have a small number of undesirable long edges and edge bends.

A layering algorithm may also be expected to produce a layering with specified either width and height, or aspect ratio. The *height* of a layering is the number of layers. Normally the nodes of DAGs from real-life applications have text labels and sometimes prespecified shape. We define the *width* of a node to be the width of the rectangle that encloses the node. If the node has no text label and no information about its shape or size is available, we assume that its width is one unit. The *width of a layer* is the sum of the widths of all nodes in that layer (including the dummy nodes) and the *width of a layering* is the maximum width of a layer. Usually the width and the height of a layering are used to approximate the dimensions of the final drawing.

It is NP-hard to find a layering with given upper bounds both on the height and on the width, even without taking into account the contribution of the dummy nodes to the width and even if all original nodes have the same unit width [Eades and Sugiyama 1990]. It is trivial to find a layering with the minimum number of original nodes per layer and no upper bound on the height. Any layering with a single node per layer is an optimal solution. However, the problem becomes NP-hard if the dummy nodes are taken into account. A recent study has shown that it is NP-hard to find a layering with a given upper bound on the width, if the width of the dummy nodes is considered greater than zero [Branke et al. 2002].

Another property of a layering is the *edge density*. The edge density between horizontal levels i and j with $i < j$ is defined as the number of edges (u, v) with $u \in L_j \cup L_{j+1} \cup \cdots \cup L_h$ and $v \in L_1 \cup L_2 \cup \cdots \cup L_i$. The edge density of a layered DAG is the maximum edge density between adjacent layers (horizontal levels). Drawings with low edge density are clear and easier to comprehend.

3. EXISTING LAYERING ALGORITHMS

There are two groups of layering algorithms that find a layering of a DAG subject to *some* of the criteria discussed above. The first group of algorithms are adopted from the area of static precedence-constrained multiprocessor scheduling. They produce layerings with either the minimum height or a specified maximum number of original nodes per layer. The second group of algorithms employ network simplex and branch-and-cut techniques for minimizing the number of dummy nodes. In this section, we briefly outline these two groups of algorithms.

3.1 List-Scheduling Algorithms

The precedence-constrained multiprocessor scheduling problem is the problem of scheduling n causally related tasks (which represent a parallel program) on m processors with the goal of minimizing the completion time of the parallel program. This problem is NP-hard when $m < \infty$ [Ulman 1975]. It is also known as *static-scheduling*, because all the tasks with their causal relationship are given in advance and the schedule must be constructed prior executing any of them. A simplified version of this problem, when all the tasks have the same computational cost and the communication time between tasks is neglected, is equivalent to the problem of finding a layering of a DAG with, at most, m nodes per layer and the minimum number of layers. Thus, the earliest static-scheduling algorithms, which deal with simplified models, have also found an application as DAG layering algorithms.

Most of the static-scheduling algorithms are variations of a generic list-scheduling technique, which consists of two main steps:

1. Build a scheduling list, which contains all the tasks.
2. While the scheduling list is not empty remove the first task from it and schedule it for execution on a processor which allows earliest start-time.

There are two list-scheduling algorithms that have been widely employed as layering algorithms. The first one is the *longest-path algorithm*, which solves the static-scheduling problem for $m = \infty$. Let π be the number of nodes in the longest-directed path in a DAG. The longest-path algorithm builds the scheduling list by assigning priority π to the nodes without outgoing edges. If all immediate successors of a node have been assigned a priority, then that node is assigned the lowest of the priorities of its immediate successors minus one. This is repeated until all nodes are assigned a priority. The nodes with the same priority k form layer $L_{\pi-k+1}$.

Algorithm 1. The Longest-Path Algorithm(G)

```
Requires: DAG $G = (V, E)$

$U \leftarrow \phi$
$Z \leftarrow \phi$
$currentLayer \leftarrow 1$
while $U \neq V$ do
  Select node $v \in V \setminus U$ with $N_G^+(v) \subseteq Z$
  if $v$ has been selected then
    Assign $v$ to the layer with a number $currentLayer$
    $U \leftarrow U \cup \{v\}$
  end if
  if no node has been selected then
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
  end if
end while
```

Alternatively, the longest-path algorithm can be viewed as building the layering layer by layer, starting from the bottom layer L_1. This can be done with the help of two node sets U and Z, which are empty in the beginning, as shown in Algorithm 1. The value of the variable $current\_layer$ is the label of the layer currently being built. As soon as a node gets assigned to a layer, it is also added to the set U. Thus, U is the set of all nodes already assigned to a layer. Z is the set of all nodes assigned to a layer below the current layer. A new node v to be assigned to the current layer is picked among the nodes which have not been already assigned to a layer, i.e., $v \in V \setminus U$, and which have all their immediate successors already assigned to the layers below the current one, i.e., $N_G^+(v) \subseteq Z$.

The advantages of the longest-path algorithm are its simplicity and its linear time complexity [Mehlhorn 1984]. It also finds layerings with the minimum height. However, it performs very poorly in terms of drawing area, number of dummy nodes, and edge density [Healy and Nikolov 2002b]. The longest-path layerings tend to be very wide at the bottom layers.

The second list-scheduling algorithm used for DAG layering is the Coffman-Graham algorithm [Lemke 1994], which is based on an earlier algorithm by Hu [1961]. It approximately solves the NP-hard static-scheduling problem for $m < \infty$. The time complexity of the algorithm is quadratic in terms of the size of the input DAG because of the more complex technique for building the scheduling list compared to the longest-path algorithm. The Coffman–Graham algorithm guarantees a layering with, at most, m original nodes per layer and in the worst case the height of the layering may become close to twice the optimal height [Coffman and Graham 1972]. It has been observed that Coffman-Graham layerings have a large amount of dummy nodes and, when they are taken into account, the area of the layerings can be even worse than the area of the longest-path layerings [Healy aand Nikolov 2002b]. We do not describe the Coffman–Graham algorithm in detail in this paper. It can be found in the original publication of Coffman and Graham [1972], as well as in several scheduling and graph-drawing publications [Eades and Sugiyama 1990; Kwok and Ahmad 1999].

3.2 Integer Linear-Programming Approaches to DAG Layering

The first approach to generating layerings with the minimum number of dummy nodes is the layering technique introduced by Gansner et al. [1993]. They model the problem by the following integer linear program:

$$\begin{aligned} \min \quad & \sum_{(u,v)\in E} l(u,\mathcal{L}) - l(v,\mathcal{L}) \\ \text{subject to:} \quad & l(u,\mathcal{L}) - l(v,\mathcal{L}) \geq 1, \quad \forall (u,v) \in E \\ & l(u,\mathcal{L}) \geq 0, \quad \forall u \in V \\ & \text{all } l(u,\mathcal{L}) \text{ are integer} \end{aligned}$$

The linear-programming relaxation of this integer program has always an integer solution, because its constraint matrix is totally unimodular [Nemhauser and Wolsey 1988]. Thus, the integer program can be solved by the simplex method. Gansner et al. [1993] go further by introducing a network simplex algorithm for solving it. Their algorithm has not been proved polynomial, but reportedly requires a few iterations and runs fast. They also perform a postprocessing balancing step. It consists of moving nodes with equal in- and out-degree up or down, keeping the layering feasible. A node is moved to an alternative layer with the fewest nodes. This is done in a greedy fashion and reportedly works sufficiently well for achieving more even node distribution over the layers while preserving the minimum number of dummy nodes.

Another approach is the branch-and-cut $(\mathrm{B}+\mathrm{C})$ layering algorithm introduced by Healy and Nikolov [2002a]. It finds layerings (if there is a feasible solution) with the minimum number of dummy nodes subject to upper bounds on both the height and the width of the layering. This algorithm is especially designed for producing high-quality layerings when the quality of the drawing has higher priority than the running time. Layered DAGs produced by the $(\mathrm{B}+\mathrm{C})$ algorithm of Healy and Nikolov have a slightly higher number of dummy nodes than the minimum, but, on average lower edge density and smaller area than layerings produced by the algorithm of Gansner et al. [1993].

The longest-path algorithm, the Coffman–Graham algorithm, and the network simplex algorithm of Gansner et al. are the three commonly used fast layering algorithms in the existing graph-drawing systems. None of them takes into account the contribution of the dummy nodes to the layering width, which has been the motivation behind the work presented in this paper. We use these three algorithms as benchmark for assessing the efficiency of the new layering heuristics proposed below.

It has been shown that the network simplex algorithm performs best in terms of various properties of the layerings. In general, layered DAGs with the minimum number of dummy nodes are compact and with low edge density [Healy and Nikolov 2002b]. Layerings found by the list-scheduling algorithms have considerably larger number of dummy nodes. In a recent study, Nikolov and Tarassov have shown that layerings found by the list-scheduling algorithms (specifically by the longest-path algorithm) can be easily improved by a simple node-promotion heuristic [Nikolov and Tarasso, 2006]. Since the node-promotion heuristic plays a significant role in the present study, we describe it in detail in the next section.

3.3 Dummy Node Reduction by Node Promotion

The node-promotion heuristic modifies a given layering $\mathcal{L} = \{L_1, \dots, L_h\}$ of a DAG G by promoting nodes from the layer where they are placed to the layer above. It is applied only to the original DAG nodes and not to the dummy nodes. To *promote* node v with $l(v, \mathcal{L}) = k$ is to move v from L_k to L_{k+1}, which results in a new partition $\mathcal{L}^* = \{L_1, \dots, L_k \setminus \{v\}, L_{k+1} \cup \{v\}, \dots, L_h\}$. If $v \in L_h$ has to be promoted, then a new empty layer L_{h+1} is added to the layering and v is inserted into it. If v has an immediate predecessor placed in layer L_{k+1} then $\mathcal{L}^*$ is *not* a layering of G. To ensure that the result of the promotion of node v to layer L_{k+1} is a layering all immediate predecessors of v in layer L_{k+1} (if there are any) have to be promoted to layer L_{k+2}; the same applies to their immediate predecessors, etc.

The recursive function, which performs the described node promotion, is shown in Algorithm 2. It takes node v as an input parameter and returns *dummydiff*, which is the difference between the number of dummy nodes before and after the promotion v. In the for loop, each immediate predecessor u of v, which lies in the layer above v, gets promoted. The return value of its promotion is added to *dummydiff*. We then promote v, subtract the number of immediate predecessors, of v from *dummydiff*, and add to it the number of immediate successors of v. That is, we promote v one layer up, recursively promoting in advance all its immediate predecessors, which need to be promoted. The time complexity of `PromoteNode` is $O(|E|)$, because, in the worst case, all DAG edges might be traversed while promoting nodes recursively.

The node-promotion heuristic then consists of two nested loops shown in Algorithm 3, an external repeat-until loop, and an internal for loop. In the internal loop, all nodes in a layered DAG are scanned, in no particular order,

Algorithm 2 `PromoteNode`(v)

Require: A layered DAG $G = (V, E)$ with the layering information stored in a global node array of integers called *layering*; a node $v \in V$.

```
$dummydiff \leftarrow 0$
for all $u \in N_G^-(v)$ do
    if $layering[u] = layering[v] + 1$ then
        $dummydiff \leftarrow dummydiff + \texttt{PromoteNode}(u)$
    end if
end for
$layering[v] \leftarrow layering[v] + 1$
$dummydiff \leftarrow dummydiff - N_G^-(v) + N_G^+(v)$
return $dummydiff$
```

Algorithm 3 `Node-Promotion Heuristic`

Require: $G = (V, E)$ is a layered DAG; a valid layering of G is stored in a global node array called *layering*.

```
$layeringBackUp \leftarrow layering$
repeat
  $promotions \leftarrow 0$
  for all $v \in V$ do
    if $d^-(v) > 0$ then
      if PromoteNode($v$) $< 0$ then
        $promotions \leftarrow promotions + 1$
        $layeringBackUp \leftarrow layering$
      else
        $layering \leftarrow layeringBackUp$
      end if
    end if
  end for
until $promotions = 0$
```

and each node with a positive in-degree gets promoted by `PromoteNode` (see Algorithm 2) if its promotion reduces the total number of dummy nodes. The external loop goes on until the internal loop makes no promotion.

The promotion of a single node in the body of the internal loop takes $O(|E|)$ time. If the promotion does not reduce the total number of dummy nodes, then the layering before the promotion is restored. This is done by making a copy of the layering before the promotion. Making a copy and restoring the layering takes $O(|V|)$ time. Thus, the worst-case time complexity of the internal loop is $O(|V| * (|V| + |E|))$.

The internal loop in Algorithm 3 scans the nodes of the DAG in no particular order. If after scanning all of them, the total number of dummy nodes has been reduced by promoting some nodes, which is an indication for repeating the body of the external loop (the repeat-until loop). In the worst case, the number of iterations of the external loop will be equal to one plus the number of dummy nodes in the initial layering, because each iteration, except the last one, decreases the number of dummy nodes. The best known estimate of the number of dummy nodes in a layered DAG is $O(min\{|V|^3, |E|^2\})$ [Lemke 1994]. Thus, in total the worst-case time complexity of the node-promotion heuristic is $O(min\{|V|^3, |E|^2\} * |V| * (|E| + |V|))$. However, it has been shown that, at most, 80 iterations of the repeat-until loop are usually enough for achieving a significant reduction of the number of dummy nodes for graphs with up to 100 nodes. Thus, if the number of iterations of the repeat-until loop is $O(|V|)$ then the worst-case time complexity of the node-promotion heuristic is $O(|V|^2 * (|E| + |V|))$.

4. HEURISTIC APPROACHES TO MINIMUM-WIDTH LAYERING

In the earlier work on the STT method, the width of a layering is defined as the maximum number of original nodes in a layer [Di Battista et al. 1999]. Given such a definition, the Coffman–Graham algorithm can be used for finding a layering with a prespecified upper bound on the width. However, it has been shown that the Coffman–Graham algorithm generates layerings with a large number of dummy nodes. The resulting drawings become much wider than expected [Healy and Nikolov 2002], because the space occupied by dummy nodes (or long edges, in other words) is not insignificant.

The present study addresses the problem of finding a layering with the minimum width when a more *realistic* definition of width is employed. In the remainder of this paper, we consider the width of a layering to be equal to the maximum number of nodes (both original and dummy) in a layer.

In practice, the network simplex algorithm of Gansner et al. [1993] outperforms the list-scheduling layering algorithms in terms of layering width when dummy nodes are considered. It has to be noted that the node-promotion heuristic applied after the longest-path algorithm improves the quality of the layering significantly, making it virtually as good as a layering with the minimum number of dummy nodes [Nikolov and Tarasso, 2006]. However, layerings with the minimum number of dummy nodes do not necessarily have the minimum width.

The four drawings in Figure 3 demonstrate that neither minimizing the number of original nodes per layer, nor minimizing the number of dummy nodes necessarily lead to layerings with the minimum width when dummy nodes are taken into account. The drawings are made with the graph-drawing system GEOMI [Ahmed et al. 2005]. The two layerings in Figures 3a and 3b have one original node per layer. The first of them has 27 dummy nodes and width 6, while the second has 18 dummy nodes and width 4. The layering in Figure 3c has only two dummy nodes, which is the minimum, and width 5. The layering in Figure 3d has 7 dummy nodes, which is more than three times the minimum, but width 3, which is the minimum width.

Taking the dummy nodes into account is a step forward to the even more precise definition given in Section 2, which considers variable widths of the original and the dummy nodes.

4.1 The `MinWidth` Heuristic

In a related previous paper, we have described the extensive parameter study we conducted to test a heuristic for layering with the minimum width [Tarassov et al. 2004]. We called the heuristic `MinWidth`. To the best of our knowledge, `MinWidth` is the first successful heuristic to address the minimum-width layering problem. An earlier attempt is the heuristic developed by Branke et al. [2001].

In this section, we present the general framework of `MinWidth`. We then propose some alternatives to the particular design decisions suggested in our previous paper.

`MinWidth` is roughly based on the longest-path algorithm, which is shown in detail in Algorithm 1. The longest-path algorithm finds layerings with the

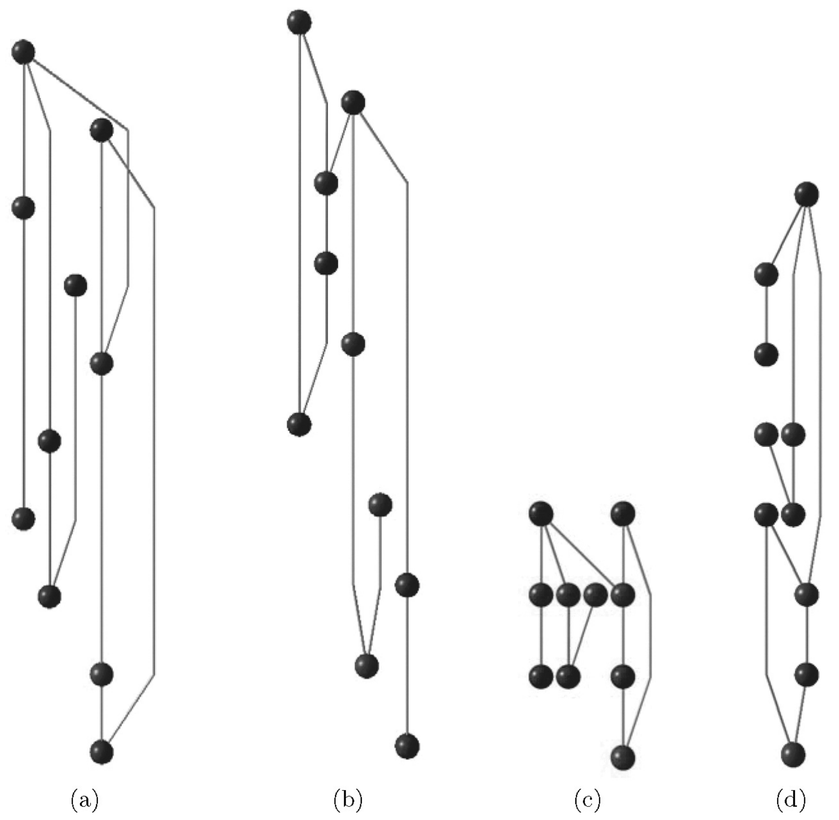


Fig. 3. Four alternative layerings of the same DAG. All edges point downward. The first two layerings (a) and (b) have one original node per layer and width 6 and 4, respectively. The third drawing (c) has the minimum number of dummy nodes and width 5; the fourth drawing (d) has the minimum width 3.

minimum height, but it does not employ any control on the layering width. `MinWidth` is displayed in Algorithm 4. Besides the DAG, G it has another two input parameters, *UBW* and c, which are explained below. We employ two variables *widthCurrent* and *widthUp* to store the width of the current layer and the width of the layers above it, respectively. The width of the current layer, *widthCurrent*, is calculated as the number of original nodes already placed in that layer plus the number of potential dummy nodes along edges with a source in $V \setminus U$ and a target in Z (one dummy node per edge). The variable *widthUp* provides an estimation of the width of *any* layer above the current one. It is the number of potential dummy nodes along edges with a source in $V \setminus U$ and a target in the current layer (one dummy node per edge). It would be more correct if *widthUp* was equal to the number of edges with a source in $V \setminus U$ and a target in U. Surprisingly, we found out by extensive tests that the more simplistic approach to *widthUp* brings better results [Tarassov et al. 2004].

Algorithm 4 `MinWidth`(G, UBW, c)

```
Requires: DAG $G = (V, E)$, integers $UBW$ and $c$
$U \leftarrow \phi$; $Z \leftarrow \phi$
$currentLayer \leftarrow 1$; $widthCurrent \leftarrow 0$; $widthUp \leftarrow 0$
while $U \neq V$ do
  Select node $v \in V \backslash U$ with $N_G^+(v) \subseteq Z$ and ConditionSelect
  if $v$ has been selected then
    Assign $v$ to the layer with a number $currentLayer$
    $U \leftarrow U \cup \{v\}$
    $widthCurrent \leftarrow widthCurrent - d^+(v) + 1$
    $widthUp \leftarrow widthup + d^-(v)$
  end if
  if no node has been selected OR ConditionGoUp then
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
    $widthCurrent \leftarrow widthUp$
    $widthUp \leftarrow 0$
  end if
end while
```

When we select a node to be placed in a layer, we employ an additional condition `ConditionSelect`, which is true if v is a node with the maximum outdegree among the candidates to be placed in the current layer. Such a choice of v brings the maximum reduction to *widthCurrent*.

In order to further control the width of the layering, we introduce a second modification to the longest-path algorithm, an additional condition for moving up to a new layer, `ConditionGoUp`. The idea is to move to a new layer either if the width of the current layer, or the width of the layers above it, become too large and cannot be improved. In order to be able to check this, we employ the parameter *UBW* against which we would like to compare the width of the current layer. *UBW* stands for upper bound on the width, but we use it as an upper bound in a very broad sense, as it becomes clear from the following paragraphs.

Since *widthUp* represents only an approximation of the width of the layers above the current one, we propose to compare it to $c \times UBW$, where $c \geq 1$, i.e., by employing c, we allow *widthUp* to be larger than *widthCurrent*, because *widthUp* is just an estimation of the width of the upper layers. We set up `ConditionGoUp` to be satisfied when either:

- $widthCurrent \geq UBW$ and $d^+(v) < 1$, or
- $widthUp \geq c \times UBW$.

We require $d^+(v) < 1$ for $widthCurrent \geq UBW$ to be taken into account because initially *widthCurrent* is set equal to the approximate number of dummy nodes in the current layer and it gets smaller (or at least it does not change) when node v with a positive out-degree gets placed in the current layer. In that case, the dummy nodes along edges with a source v are removed from the current layer and get replaced by v. If $d^+(v) \geq 1$, the condition *widthCurrent* $\geq$ *UBW* on its own is not a reason for moving to the upper layer, because there is still a chance to add nodes to the current layer, which will reduce *widthCurrent*. If $d^+(v) = 0$ then the assignment of v to the current layer increases *widthCurrent*, because it does not replace any dummy nodes. This is an indication that *widthCurrent* can not be further reduced.

Note that we assign the selected node to the current layer before checking `ConditionGoUp`. We do this because both *UBW* and c can have any values and the while loop may never end if we assign the selected node to the current layer only if `ConditionGoUp` is not satisfied. In the following section, we introduce a heuristic that does not require the parameters *UBW* and c and employs a more elaborate `ConditionGoUp`, which is checked before assigning the selected node to a layer.

With the use of proper data structures, `MinWidth` can be implemented as an algorithm with $O(|V| \log |V| + |E|)$ time complexity. For this purpose we need a sorted list of nodes L, which is initialized before starting the while loop. Initially L contains all sinks sorted by out-degree. Such a list can be built in $O(|V|)$ time with an efficient implementation of L, e.g., as a Fibonacci heap. When a node is assigned to layer in the body of the while loop it is deleted from L and all its immediate predecessors are scanned one by one. Those of them, which are not already in L, but are ready to be assigned to a layer, are inserted into L while keeping L sorted by out-degree. The deletion of the selected node from L takes $O(\log |V|)$ time and the insertion of a node into L takes $O(1)$ time if L is implemented as a Fibonacci heap. In total, the maintenance of L throughout the algorithm takes $O(|V| \log |V|)$ time. Within the while loop each node and each edge are examined once, which takes $O(|V| + |E|)$ time. Thus the worst-case time complexity of `MinWidth` is $O(|V| \log |V| + |E|)$ or $O(|V| \log |V|)$ for sparse DAGs.

We have preformed an extensive parameter study in order to determine the optimal values for *UBW* and c. It was presented in our previous paper [Tarassov et al. 2004]. The results provide evidence that the narrowest layerings are found for $1 \leq UBW \leq 4$ and $1 \leq c \leq 2$. We conjecture that the optimal value range for c is related to the average out-degree of a node, which is between 1 and 2 in the graph data set we used for experiments. Thus, we proposed to run `MinWidth` for $UBW \in \{1, 2, 3, 4\}$ and $c \in \{1, 2\}$ and then choose the narrowest of the eight layerings. It turned out that this layering technique finds layerings with a much higher number of dummy nodes than the number of dummy nodes in layerings found by the well-known layering algorithms. We suggested that the node-promotion heuristic should be applied after `MinWidth` in order to improve the result. In order not make the layering wider as a result of node promotion, we proposed a promotion of a node to an upper layer to be performed only if it does not increase the width of the layering.

Furthermore, we have, in addition, tested alternative versions of `ConditionSelect`, which considered both the in- and the out-degree of the candidates and their immediate predecessors. However, it turned out that the simplest approach to choose the candidate with the maximum out-degree works best.

Before we proceed with the computational results, we introduce our second heuristic for tackling the minimum-width layering problem. It is a modification of `MinWidth`, which does not require the input parameters *UBW* and c.

4.2 The `StretchWidth` Heuristic

We call our second heuristic `StretchWidth`, because it builds the layering by trying to have its width lower than or equal to an upper bound, which gradually gets bigger. Initially, we set up the upper bound at $\max\{\max\{d^+(v) : v \in V\}, \max\{d^-(v) : v \in V\}\}$. If the algorithm reaches a point where it is impossible to assign a node to a layer without going above the upper bound, then we increment the upper bound by one and start over.

The details of the `StretchWidth` heuristic are shown in Algorithm 5. It builds the layering in a way similar to `MinWidth`, but without the need of the input parameters *UBW* and c.

`ConditionSelect` is slightly more complex than it is in `MinWidth`. For each candidate v to be placed in the current layer, we compute the value $rank(v) = \max\{d^+(v), \max\{d^+(u) : (u, v) \in E\}\}$ and choose the candidate with the biggest $rank(v)$. By making this choice, we allow nodes with high out-degree (which can make big improvement to the width of some of the layers

Algorithm 5 `StretchWidth`(G)

```
Requires: DAG $G = (V, E)$
$U \leftarrow \phi$; $Z \leftarrow \phi$
$currentLayer \leftarrow 1$; $widthCurrent \leftarrow 0$; $widthUp \leftarrow 0$
$maxwidth \leftarrow 0$
while $U \neq V$ do
  Select node $v \in V \backslash U$ with $N_G^+(v) \subseteq Z$ and ConditionSelect
  if no node has been selected or (ConditionGoUp and $U \backslash Z \neq \phi$) then
    /* move to the upper layer */
    $currentLayer \leftarrow currentLayer + 1$
    $Z \leftarrow Z \cup U$
    $widthCurrent \leftarrow widthUp$
    $widthUp \leftarrow 0$
  else
    if ConditionGoUp then
      /* reset the layering */
      $currentLayer \leftarrow 1$; $widthCurrent \leftarrow 0$; $widthUp \leftarrow 0$
      $U \leftarrow \phi$; $Z \leftarrow \phi$
      $maxwidth \leftarrow maxwidth + 1$
    else
      /* add a node to the current layer */
      Assign $v$ to the layer with a number $currentLayer$
      $U \leftarrow U \cup \{v\}$
      $widthCurrent \leftarrow widthCurrent - d^+(v) + 1$
      $widthUp \leftarrow widthup + d^-(v)$
    end if
  end if
end while
```

above the current layer) to be assigned to a layer as soon as possible without being blocked by their successors with low out-degree. Through experiments with thousand of DAGs, we observed that `StretchWidth` gives slightly better results with this more complex `ConditionSelect` than it does with the simpler `ConditionSelect`, employed in `MinWidth`.

The variable $maxwidth$ is the maximally allowed width mentioned above. `ConditionGoUp` is true if the placement of the selected node v in the current layer makes either

- $widthCurrent > maxwidth$ or
- $widthUp > maxwidth *$ average out-degree of a node in G.

We take this as an indication to move to the upper layer only if the current layer is not empty. If the current layer is empty and no node can be assigned to it without satisfying `ConditionGoUp`, then we add 1 to $maxwidth$ and re-start building the layer from the bottom layer. In this case, we also say that the layering is *reset*. If a candidate has been selected and `ConditionGoUp` is false, we assign the candidate to the current layer. In that case, we also update the values of $widthCurrent$ and $widthUp$.

Similar to `MinWidth`, `StretchWidth` can be implemented efficiently by maintaining a list of nodes sorted by rank. It takes $O(|V||E|)$ time to compute the ranks of all nodes in advance. `StretchWidth` will reset the layering no more than $O(|E|)$ times, which makes its total worst-case time complexity $O(|E||V|\log|V| + |E|^2)$ (provided each reset has a precomputed list of sinks sorted by rank). The worst-case time complexity of `StretchWidth` for sparse DAGs is $O(|V|^2 \log|V|)$. However, the computational results in Section 5 show evidence that `StretchWidth` runs even faster than `MinWidth`, in practice.

In the next section, we present an extensive computational study on the properties of layerings found by both `MinWidth` and `StretchWidth`.

5. COMPUTATIONAL RESULTS

In our experimental work, we used 5911 DAGs from the well-known Rome graph dataset introduced by Di Battista et al. [1997] in their experimental studies and available at the GDToolkit website[1]. The copy of the Rome graph set consists of 11,530 graphs in LEDA[2] format. Since, by default, a graph in LEDA format is directed, we accepted the default direction of the edges given by the LEDA format and filtered out the graphs with directed cycles. We also filtered out the unconnected graphs leaving 5911 DAGs. The Rome graphs have node count between 10 and 100 nodes and typically twice as many edges as nodes. The x axis in all the plots below represents the number of original nodes in a graph. We have partitioned all DAGs into groups by node count. Each group covers an interval of size 5 on the x axis, except the last group, which represents only the graphs with 100 nodes. We display the average result for each group. The machine we used for computation has an 1.8 GHz Pentium M processor.

[1]http://www.dia.uniroma3.it/~gdt/

[2]http://www.algorithmic-solutions.de/enleda.htm

We run the longest-path algorithm, the Coffman–Graham algorithm, `MinWidth`, and `StretchWidth` for all the 5911 DAGs. We used QSopt 1.0[3] for solving the linear program proposed by Gansner et al. [1993] for finding layerings with the minimum number of dummy nodes. The Coffman–Graham algorithm takes an upper bound m on the number of original nodes in a layer as an input parameter. Thus, we run it for $m = 1..|V|$, where V is the node set of the DAG, and chose the narrowest layering taking into account the dummy nodes. In the plots below, we denote the layering techniques by LPath (the longest-path algorithm), CG (the Coffman–Graham algorithm), Gans (the integer linear-programming model of Gansner et al. [1993]), `MinWidth`, and `StretchWidth` respectively.

In the remainder of this section, we present two groups of results, without and with the postprocessing node-promotion step, respectively. First we discuss the properties of the layerings found without node promotion and then we study the effect of the node-promotion heuristic to the quality of the layerings.

5.1 The Layering Algorithms Compared

First, we present the properties of the layerings before applying the postprocessing node-promotion step. We compared five parameters of the layerings as well as the running time. The five parameters are

- Width—the maximum number of both original and dummy nodes in a layer;
- Simplified width—the maximum number of original nodes in a layer;
- Height—the number of layers;
- Number of dummy nodes divided by the total number of nodes;
- Edge density—the maximum number of edges between adjacent layers divided by the total number of edges.

The results for width and simplified width are shown in Figures 4 and 5b, respectively. On average, the Gans layerings have the minimum width followed closely by `MinWidth`. For large graphs, the width of the Gans and `MinWidth` layerings is about 20% smaller than the width of the layerings found by the other algorithms. As it can be seen in Figure 5, `MinWidth` achieves this by having significantly smaller maximum number of original nodes in a layer than any of the other algorithms. This result suggests that the `MinWidth` layerings have much higher dummy node count than the minimum. Indeed, we can see it in Figure 7. In spite of the higher number of dummy nodes, the width of the `MinWidth` layerings is closer to the width of the Gans layerings, because of the much higher number of layers. The height of the layerings shown in Figure 6. `MinWidth` also achieves the lowest edge density (see Figure 8).

In general, the Coffman–Graham algorithm (CG) is the worst in terms of width and edge density. `StretchWidth` gives slightly better results for width and edge density than the longest-path algorithm. All of the algorithms except CG run within 0.02 s, on average. The running time of CG is within 1.5 s and we excluded it from the running-time plot in Figure 9b to be able to show the

[3]http://www2.isye.gatech.edu/~wcook/qsopt/

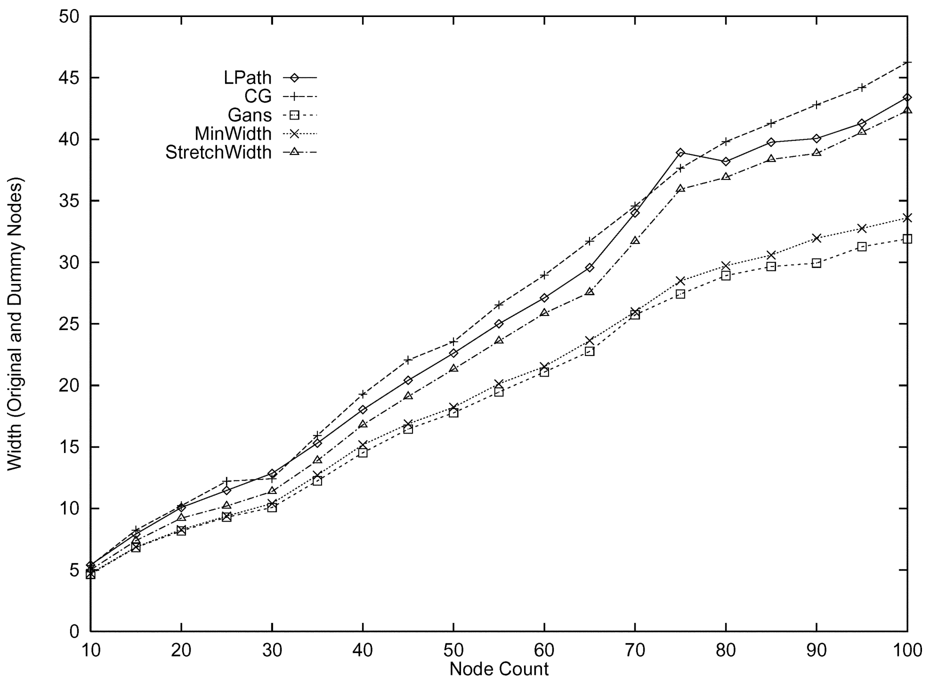


Fig. 4. Layering width before node promotion.

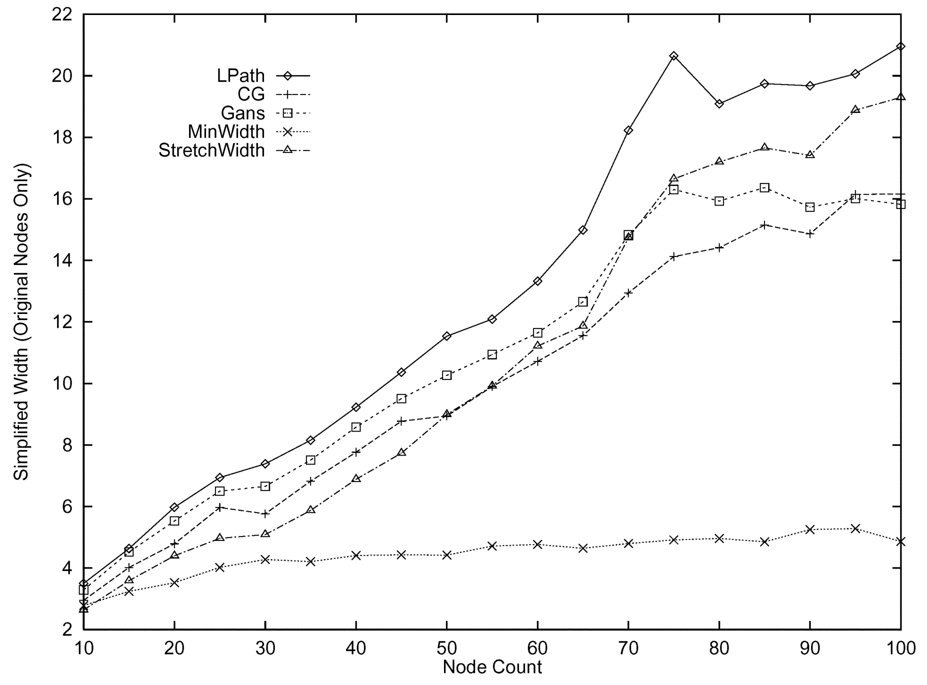


Fig. 5. Simplified layering width before node promotion.

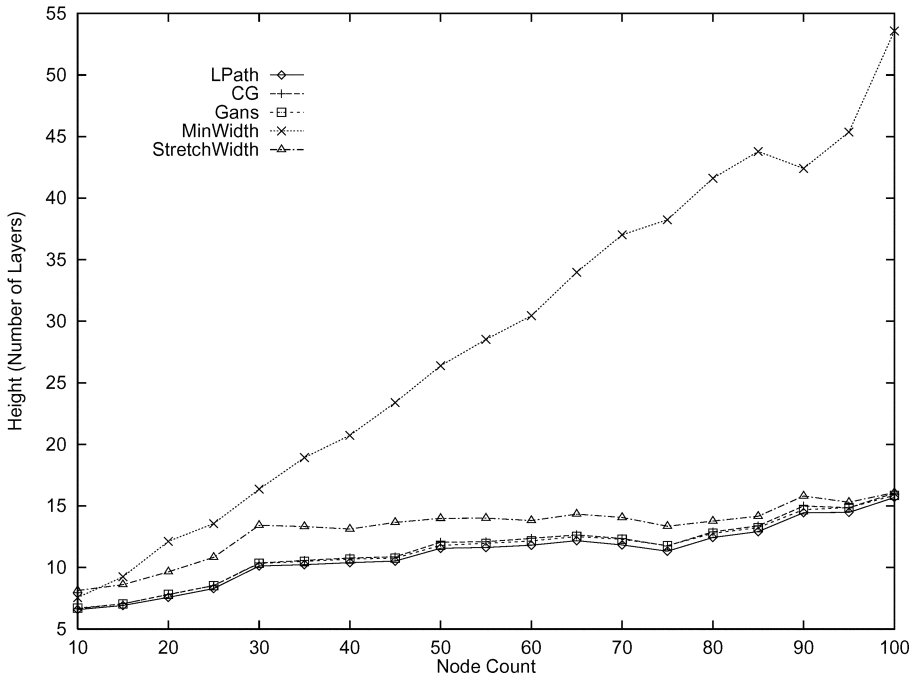


Fig. 6. Layering height before node promotion.

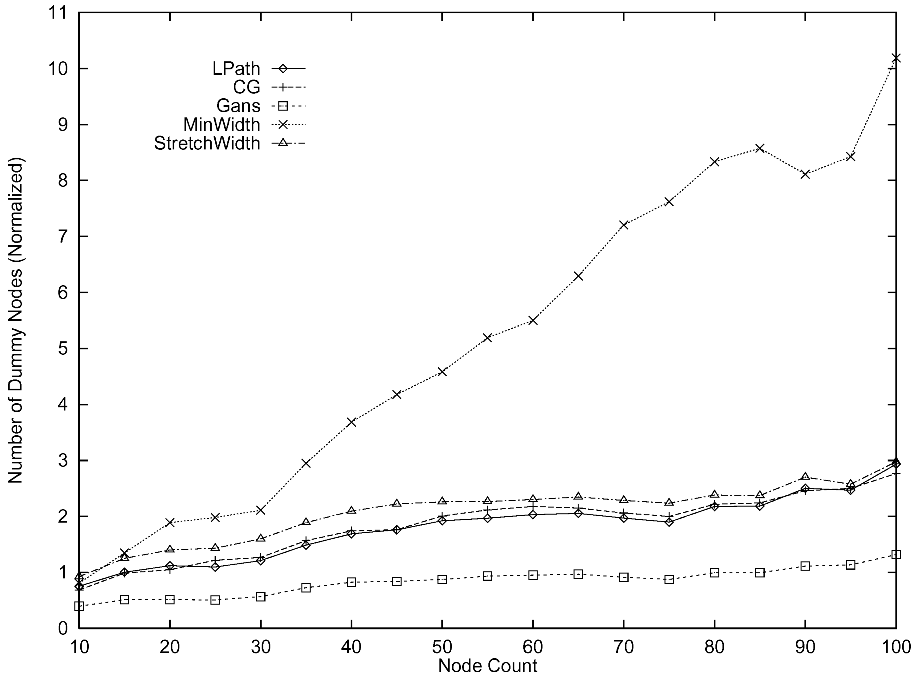


Fig. 7. Normalized number of dummy nodes before node promotion.

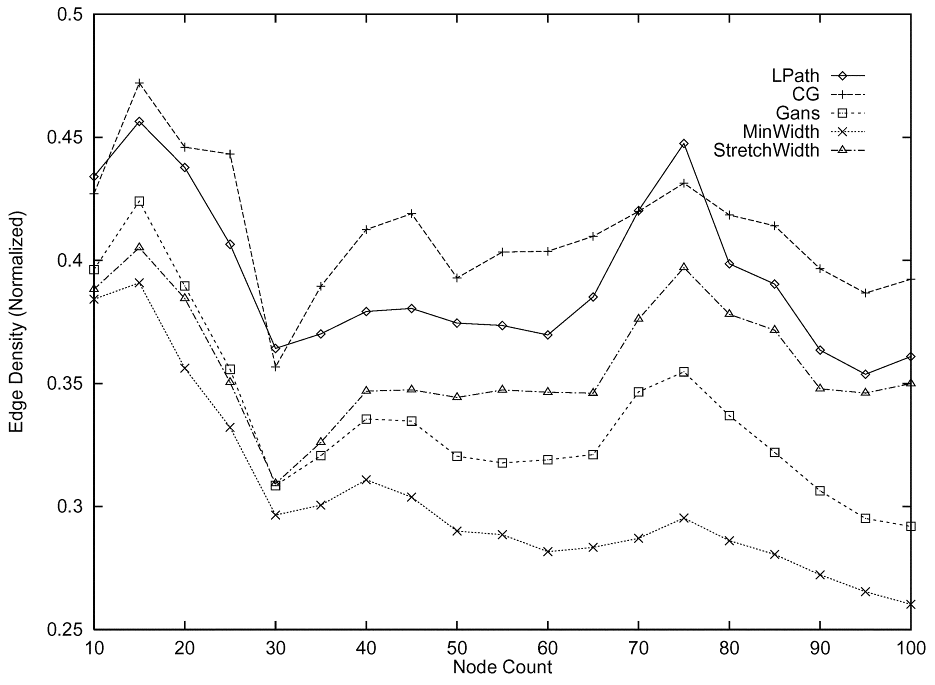


Fig. 8. Normalized edge density before node promotion.

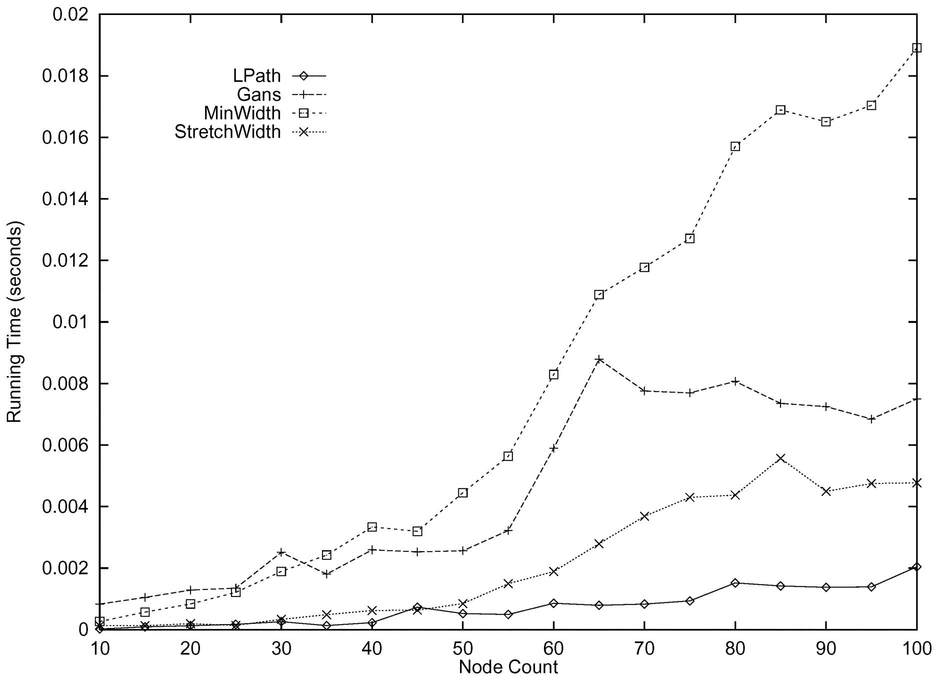


Fig. 9. Running times before node promotion.

difference between the running times of the other four algorithms. CG is much slower than the rest of the algorithms because we run it in a loop $|V|$ times and choose the narrowest layering that it finds.

The good width and edge-density properties of `MinWidth` with such a high number of dummy nodes suggest that it might be fruitful to postprocess its layerings with the node-promotion heuristic. We study the effect of the node-promotion heuristic to `MinWidth`, `StretchWidth`, LPath, and CG in the following section.

5.2 Effect of Promotion

The node-promotion technique described in Section 3.3 can be applied to any layering that does not have the minimum number of dummy nodes in order to reduce its dummy node count. In this section, we present the properties of the layerings found by `MinWidth`, `StretchWidth`, LPath, and CG after applying the node-promotion heuristic to them. We used a slightly modified version of the heuristic in which a promotion of a node to an upper layer is performed *only* if it does not increase the width of the layering. We allowed, at most, $|V|/2$ iterations of the repeat-until loop. The results are shown in Figures 10–15.

As we expected, node promotion reduces the height significantly (approximately by one-half) and the dummy node count (approximately by one-third) of the `MinWidth` layerings. After node promotion the `MinWidth` layerings have smaller width than the previously best Gans layerings and still the lowest edge density (see Figures 10 and 14). The LPath and `StretchWidth` layerings have slightly lower edge density than Gans and the `StretchWidth` are slightly narrower than the Gans layerings.

The Coffman–Graham algorithm is still the worst in terms of width and edge density with height values virtually the same as the height values of LPath, Gans, and `StretchWidth` (see Figure 12). The `MinWidth` layerings are the tallest, which is expected for narrow layerings with low edge density. The large number of layers of the `MinWidth` layerings is correlated to the relatively large number of dummy nodes. If the time constraints allow it, more iterations of the repeat-until loop of the node-promotion heuristic may further reduce the number of dummy nodes in the `MinWidth` layerings.

The running times are shown in Figure 15. `MinWidth` runs within 1.75 s on average, for the DAGs with less than 100 nodes and just above 2 s, on average, for the small group of DAGs with 100 nodes. `StretchWidth` runs within 0.75 s, on average. It has to be noted that we did not specifically implement the algorithms in the most efficient way in terms of running time. Better implementations are possible. Thus, the running times we present serve only to show that the proposed heuristics run in reasonable time. The very fast performance of Gans is probably because of the fact that it consists of calling the highly efficient QSopt for computing the layer assignment.

5.3 Example Drawings

Before we summarize the conclusions of our work, we show two examples in Figures 16 and 17. All the hierarchical drawings were made with the graph

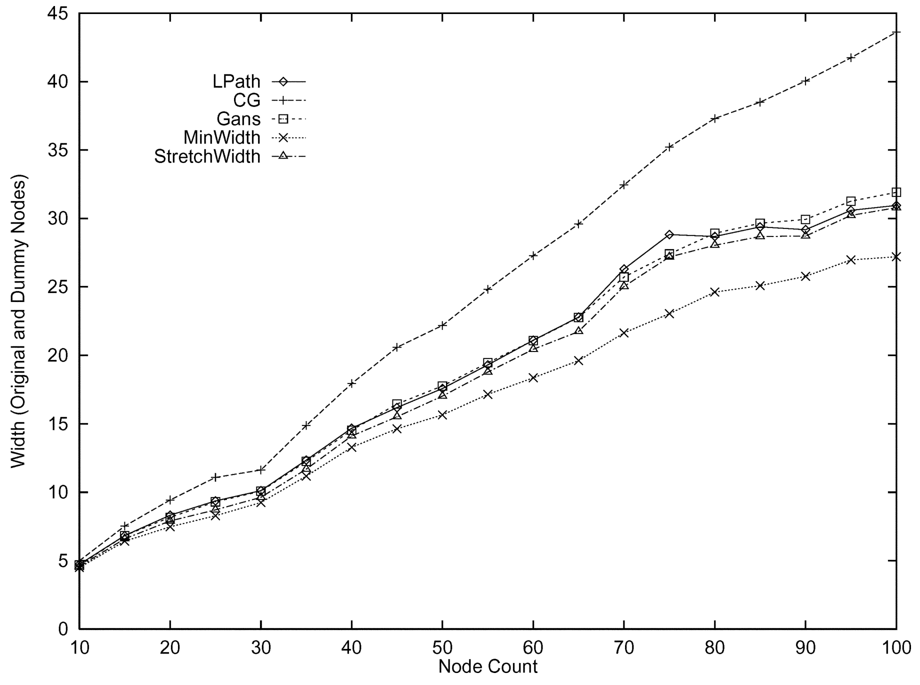


Fig. 10. Layering width after node promotion.

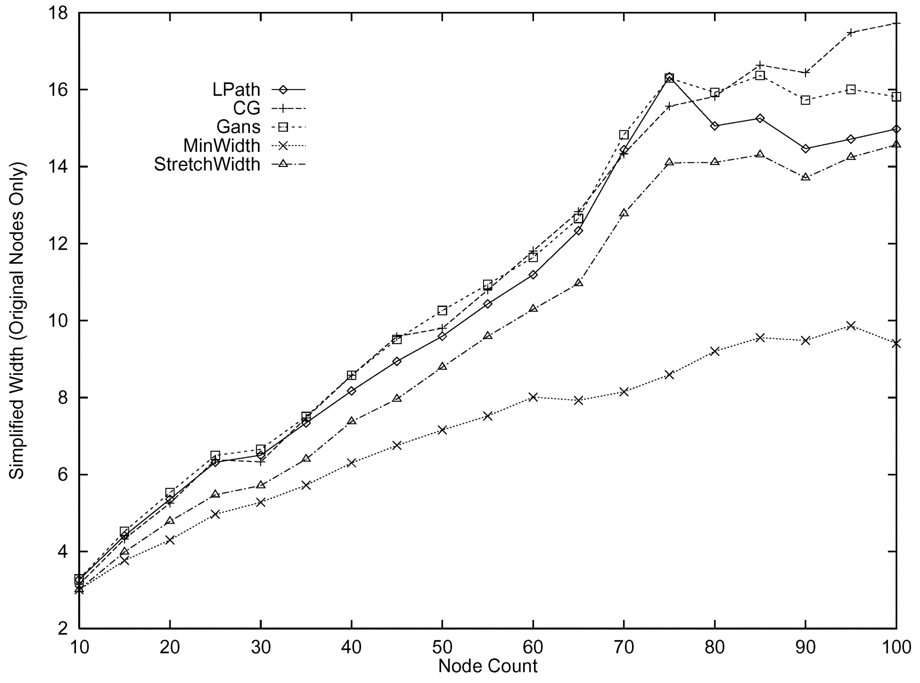


Fig. 11. Simplified layering width after node promotion.

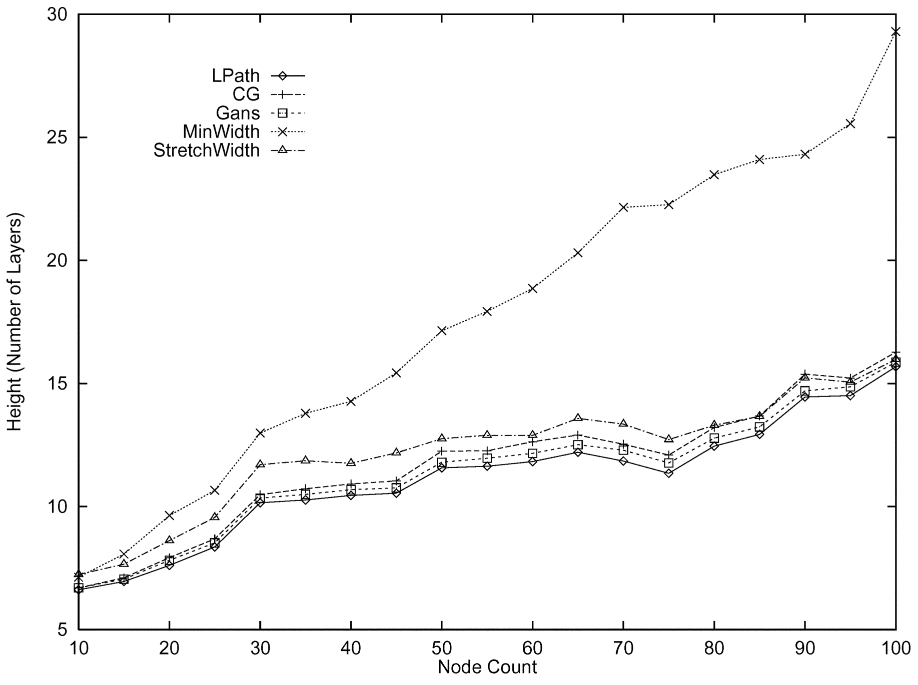


Fig. 12. Layering height after node promotion.

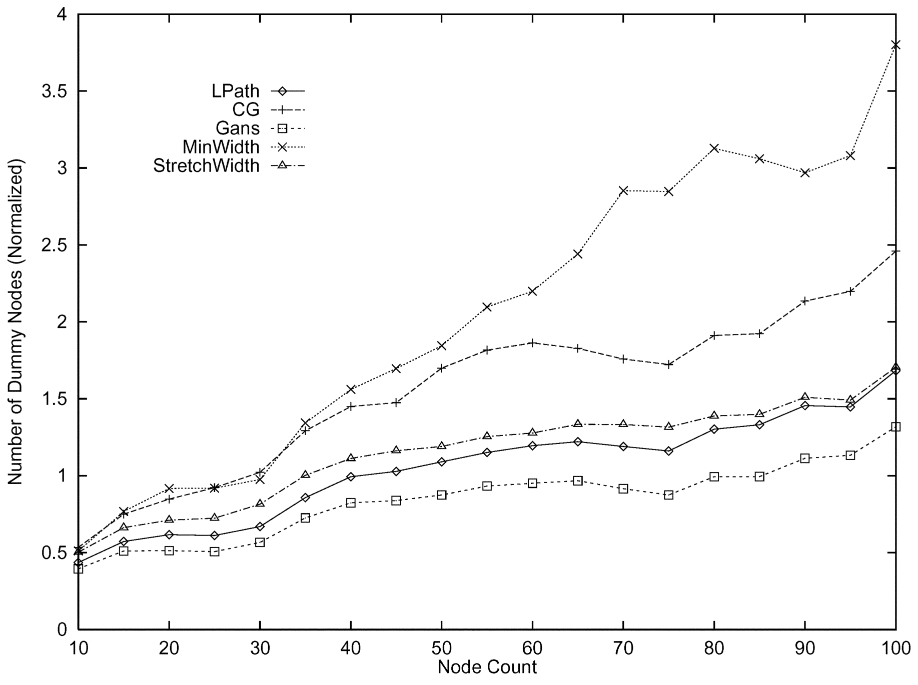


Fig. 13. Normalized number of dummy nodes after node promotion.

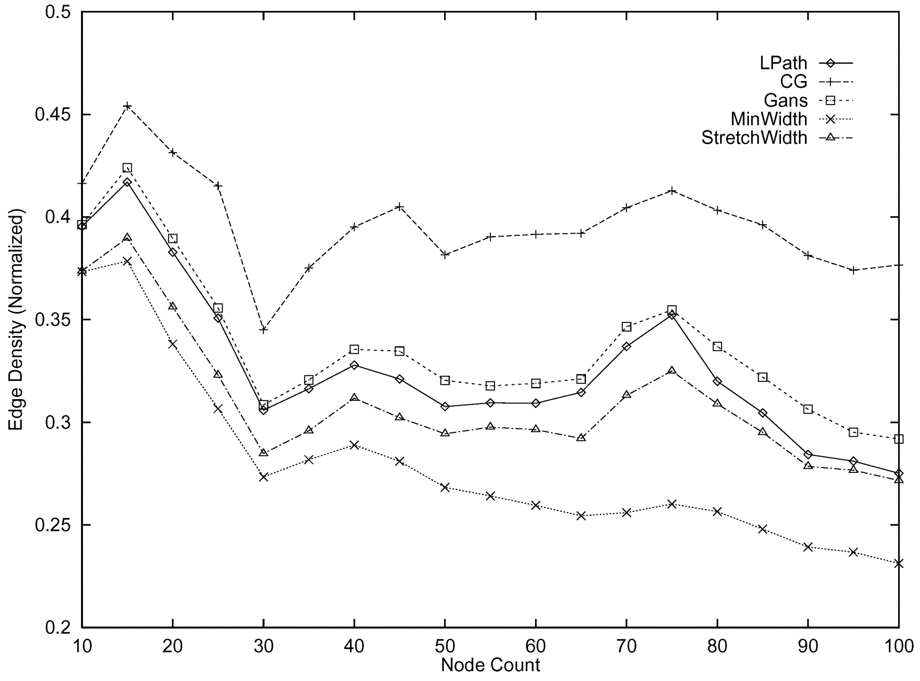


Fig. 14. Normalized edge density after node promotion.

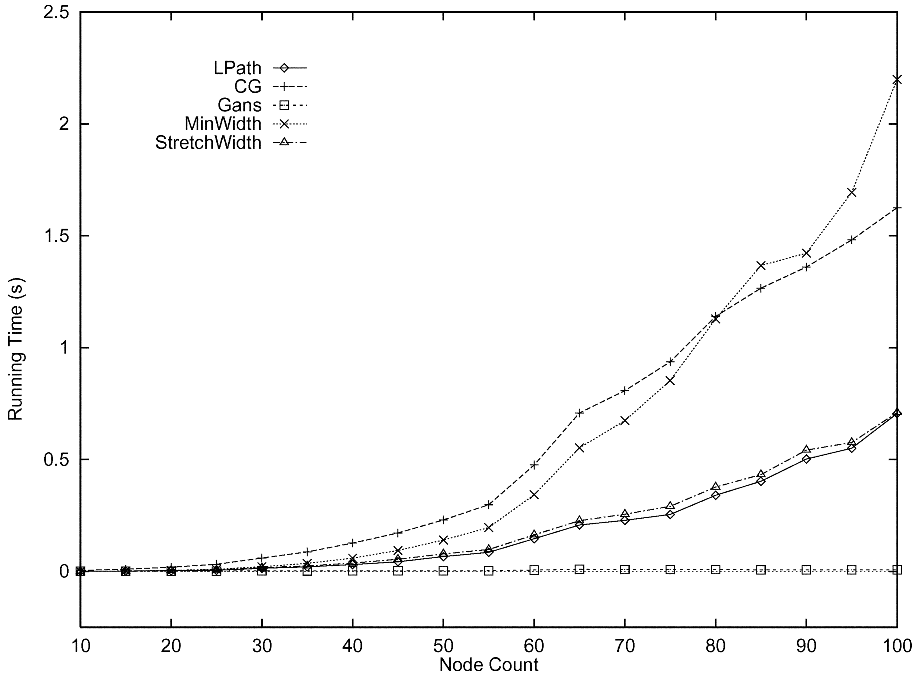


Fig. 15. Running times after node promotion.

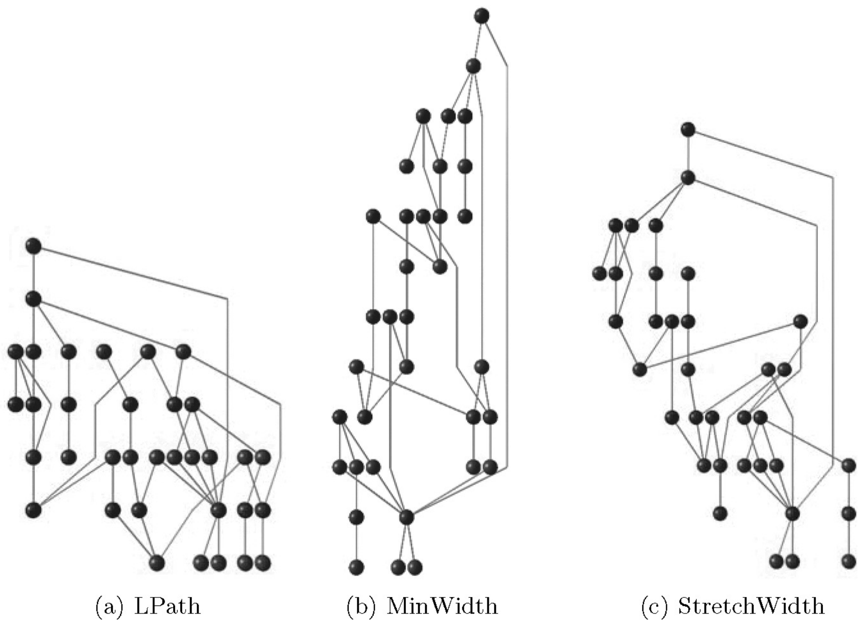


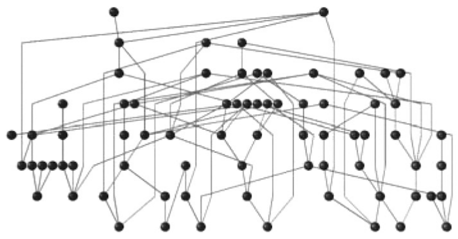
Fig. 16. Three alternative layerings of the same DAG, grafo11330, taken from the Rome graph set. The graph has 35 nodes. All edges point downward.

visualization system GEOMI [Ahmed et al. 2005] by applying the same algorithms at each phase of the STT method except the layer-assignment phase. We show the `MinWidth`, the `StretchWidth`, and the LPath layerings for two DAGs after postprocessing with the node-promotion heuristic.

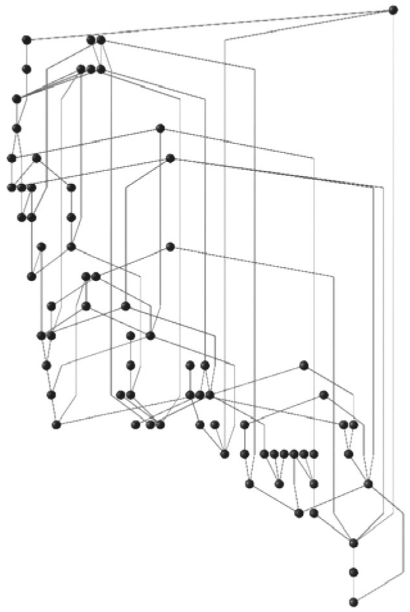
Note that the actual width of the drawings also depends on the x coordinate assignment in the STT method. Typical objectives of an x coordinate assignment algorithm are to minimize the number of edge bends and to ensure that as many edges as possible are drawn as straight lines. At present there are no x coordinate assignment methods, which achieve these objectives, while keeping the width of the drawing as close to the layering width as possible. Thus, some of the drawings we present here are drawn wider that their layering width allows.

The smaller DAG in Figure 16 has 35 nodes. Its `MinWidth` layering is the narrowest (width 7), but also the tallest (height 12), and it has the lowest edge density. The `StretchWidth` layering has width 8 with height 10 and the LPath layering has width 13 with height 7. For the same DAG, the CG layering has width 15 with height 9, and the Gans layering has width 13 with height 7.

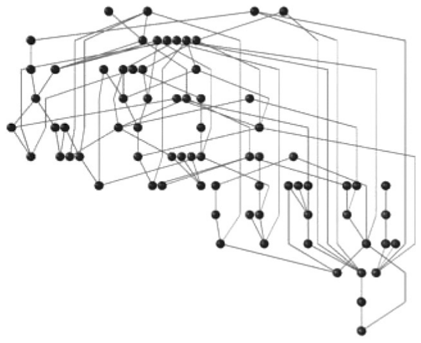
The DAG in Figure 17 has 73 nodes. Again, the `MinWidth` layering is the narrowest (width 19), but the tallest (height 21), and it has the lowest edge density. The `StretchWidth` layering has width 24 with height 12 and the LPath layering has width 32 with height 8. For the same DAG, the CG layering has width 39 with height 10 and the Gans layering has width 31 with height 9.



(a) LPath



(b) MinWidth



(c) StretchWidth

Fig. 17. Three alternative layerings of the same DAG, grafo5074, taken from the Rome graph set. The graph has 73 nodes. All edges point downward.

The examples suggest that it might be fruitful to apply an edge concentration technique after both `MinWidth` and `StretchWidth`, i.e., long parallel edges to be represented by a single edge.

6. CONCLUSIONS

We propose two layering algorithms, `MinWidth` and `StretchWidth`, which can be used at the layer-assignment phase of the STT method for drawing directed graphs. Our heuristics are specifically designed to solve the NP-hard problem of minimum-width DAG layering approximately when the contribution of the so-called dummy nodes to the drawing width is taken into account.

We have conducted an extensive computational study in order to evaluate our heuristics by comparing them to the well-known fast-layering algorithms. In conclusion, `MinWidth`, followed by the previously introduced node-promotion heuristic, is the winner in our effort to design a technique for minimum-width layering. The `MinWidth` layerings have also lower edge density than other layerings. They are quite tall which, combined with the low edge density, leads to a higher dummy node count. If this is not acceptable, then `StretchWidth` followed by node promotion can be an alternative.

We have found out that `StretchWidth` behaves very similar to the longest-path layering algorithm both before and after applying the node-promotion heuristic. This is an interesting result that needs further research to be fully understood. Even if it may not result in a new layering technique, our feeling is that a further study of this relation may provide better understanding of the properties of hierarchical graphs.

The work presented in this paper is, together with the preceding workshop paper [Tarassov et al. 2004], the first successful attempt to design a heuristic for the NP-hard problem of minimum-width DAG layering with consideration of dummy nodes. Further research may extend our approach to solving the minimum-width DAG layering problem with consideration of variable node widths and to attempt to minimize the height given a constraint on the width.

REFERENCES

AHMED, A., DWYER, T., FORSTER, M., FU, X., HO, J., HONG, S., KOSCHÜTZKI, D., MURRAY, C., NIKOLOV, N. S., TAIB, R., TARASSOV, A., AND XU, K. 2005. GEOMI: GEOmetry for Maximum Insight. In *Graph Drawing: Proceedings of 13th International Symposium, GD 2005*, vol. 3843 of *Lecture Notes in Computer Science*. Springer-Verlag, New York. 468–479.

BRANKE, J., EADES, P., LEPPERT, S., AND MIDDENDORF, M. 2001. Width restricted layering of acyclic digraphs with consideration of dummy nodes. Tech. Rep. No. 403, Intitute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany.

BRANKE, J., LEPPERT, S., MIDDENDORF, M., AND EADES, P. 2002. Width-restricted layering of acyclic digraphs with consideration of dummy nodes. *Information Processing Letters 81*, 2, 59–63.

CARPANO, M. J. 1980. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Systems, Man and Cybernetics 10*, 11, 705–715.

COFFMAN, E. G. AND GRAHAM, R. L. 1972. Optimal scheduling for two processor systems. *Acta Informatica 1*, 200–213.

DI BATTISTA, G., ENGLEWOOD CLIFFS, N. J., GARG, A., LIOTTA, G., TAMASSIA, R., TASSINARI, E., AND VARGIU, F. 1997. An experimental comparison of four graph drawing algorithms. *Computational Geometry: Theory and Applications 7*, 303–316.

DI BATTISTA, G., EADES, P., TAMASSIA, R., AND TOLLIS, I. G. 1999. *Graph drawing*. Prentice Hall.

EADES, P., LIN, X., AND SMYTH, W. F. 1993. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters 47*, 319–323.

EADES, P. AND SUGIYAMA, K. 1990. How to draw a directed graph. *Journal of Information Processing 13*, 4, 424–437.

GANSNER, E. R., KOUTSOFIOS, E., NORTH, S. C., AND VO, K.-P. 1993. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering 19*, 3, 214–230.

HEALY, P. AND NIKOLOV, N. S. 2002a. A branch-and-cut approach to the directed acyclic graph layering problem. In *Graph Drawing: Proceedings of 10th International Symposium, GD 2002*, vol. 2528 of *Lecture Notes in Computer Science*. Springer-Verlag, New York. 98–109.

HEALY, P. AND NIKOLOV, N. S. 2002b. How to layer a directed acyclic graph. In *Graph Drawing: Proceedings of 9th International Symposium, GD 2001*, vol. 2265 of *Lecture Notes in Computer Science*. Springer-Verlag, New York. 16–30.

HU, T. 1961. Parallel sequencing and assembly line problems. *Operations Research 9*, 841–848.

KWOK, Y.-K. AND AHMAD, I. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys 31*, 4, 406–471, 1999.

LEMKE, I. 1994. Entwicklung and Implementierung eines Visualisierungswerkzeuges für Anwendungen im Übersetzerbau. Diplomarbeit, Universität des Saarlandes, FB 14 Informatik.

MEHLHORN, K. 1984. *Data Structures and Algorithms, Volume 2: Graph Algorithms and NP-Completeness*. Springer-Verlag, New York.

NEMHAUSER, G. L. AND WOLSEY, L. A. 1988. *Integer and Combinatorial Optimization*. Wiley, New York.

NIKOLOV, N. S. AND TARASSOV, A. 2006. Graph layering by promotion of nodes. *Discrete Applied Mathematics, Special Issue Associated with the IV ALIO/EURO Workshop on Applied Combinatorial Optimization. 154*, 5, 848–860.

SUGIYAMA, K. AND MISUE, K. 1995. Graph drawing by the magneting spring model. *Journal of Visual Languages and Computing 6*, 3, 217–231.

SUGIYAMA, K., TAGAWA, S., AND TODA, M. 1981. Methods for visual understanding of hierarchical system structures. *IEEE Transaction on Systems, Man, and Cybernetics 11*, 2, 109–125.

TARASSOV, A., NIKOLOV, N. S., AND BRANKE, J. 2004. A heuristic for minimum-width of graph layering with consideration of dummy nodes. In *Experimental and Efficient Algorithms, Third International Workshop, WEA 2004*, vol. 3059 of *Lecture Notes in Computer Science*. Springer-Verlag, New York. 570–583.

ULMAN, J. 1975. NP-complete scheduling problems. *Journal of Computer and System Sciences 10*, 384–393.

UTECH, J., BRANKE, J., SCHMECK, H., AND EADES, P. 1998. An evolutionary algorithm for drawing directed graphs. In *Proceedings of the 1998 International Conference on Imaging Science, Systems, and Technology (CISST' 98)*. 154–160.

WARFIELD, J. N. 1977. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man and Cybernetics 7*, 7, 502–523.

Received February 2005; revised December 2005; accepted December 2006