Graph Layering by Promotion of Nodes

Nikola S. Nikolov and Alexandre Tarassov

Department of Computer Science and Information Systems,
University of Limerick, Ireland
{nikola.nikolov,alexandre.tarassov}@ul.ie

Abstract

This work contributes to the wide research area of visualization of hierarchical graphs. We present a new polynomial-time heuristic which can be integrated into the method of Sugiyama, Tagawa and Toda (STT) for drawing hierarchical graphs. Our heuristic, which we call Promote Layering (PL), is applied to the output of the layering phase of the STT method. PL is a simple and easy to implement algorithm which decreases the number of so-called dummy (or virtual) nodes in a layered directed acyclic graph (DAG). In particular, we propose applying PL after the longest path layering algorithm and we present an extensive empirical evaluation of this layering technique. The experimental results suggest that PL applied after the longest path algorithm leads to layered DAGs with aesthetic characteristics often better than the aesthetic characteristics of layered DAGs with the minimum number of dummy nodes.

Key words: graph drawing, layered directed acyclic graph, layering algorithm

1 Introduction

To layer a directed acyclic graph (DAG) is to partition its node set into subsets such that nodes connected by a directed path belong to different subsets. In addition, subsets are assigned integer ranks 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. Each DAG allows at least one layering; a DAG with a given layering is called a layered DAG. Most often layered DAGs are visualized by placing the DAG nodes on parallel horizontal levels such that each layer occupies a single level, different layers occupy different levels, and all edges point in the same direction. Figure 1 gives an example of two alternative ways to layer the same DAG.

Preprint submitted to Elsevier Science

29 November 2002
Fig. 1. Two alternative layerings of the same DAG. Each layer occupies a horizontal level marked by a dashed line. All edges point downwards.

In this work we consider the graph layering problem in the context of DAG visualization. In the research area known as *graph drawing* there have been recognized a few different methods for drawing DAGs. The more recent two are a magnetic field model introduced by Sugiyama and Misue [15] and an evolutionary algorithm by Utech et al. [18]. While potentially these two are an area of fruitful future research, an earlier method, widely known as the Sugiyama (or STT) method [16], has received most of the research attention and has become a standard method for drawing DAGs. The Sugiyama method is a three phase algorithmic framework, originally proposed by Sugiyama et al. [16], and also based on work by Warefield [19] and Carpano [2]. At its first phase the nodes of a DAG are partitioned into layers and each layer is assigned to a horizontal level; at the second phase the nodes are ordered within each layer; and at the final third phase the $x$- and $y$-coordinates of all nodes and the eventual edge bends are precisely tuned. In this paper we propose a heuristic, called Promote Layering (PL), that can be applied at the end of the first phase and before the second phase for improving the characteristics of an already found layering.

If there are no additional requirements it is not hard to find a layering of a DAG. Classical graph algorithms such as breadth-first search, depth-first search and algorithms for finding a minimum spanning tree can be easily modified to partition the node set of a DAG into layers. However, normally it is desirable to take into account a number of additional criteria when computing a layering [3]. It might be desirable to keep the number of layers and the maximum number of nodes per layer within certain bounds. Also, for the clarity of the final drawing, it is preferable to partition the node set into layers so that long edges spanning several layers are kept to a small number. A large number of long edges also significantly slows down the algorithms applied at the next two phases of the Sugiyama method. It might be also a
good idea to find a layering with low edge density between adjacent horizontal levels in the corresponding drawing. The PL heuristic that we propose in this paper can be used for achieving a layering with short edges and, as we have observed experimentally, low edge density.

The paper is organized as follows. The next section introduces the basic terminology required for discussing graph layering further. In Section 3 we briefly describe the known algorithms for partitioning the node set of a DAG into layers. Section 4 introduces our new layering improvement heuristic, PL. Then in Section 5 we report experimental results of applying PL to about 8,000 DAGs taken from various benchmark graph databases and compare it to other layering techniques. In the final Section 6 we draw conclusions from this work and outline directions for further work.

2 Preliminaries

Consider a DAG $G = (V, E)$ with a set of nodes $V$ and a set of edges $E$. The in-degree of node $v$, denoted by $d^-(v)$, is the number of edges with a target $v$, and the out-degree of $v$, denoted by $d^+(v)$, 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 $N^+_G(v) = \{v : (v, u) \in E\}$.

Let $\mathcal{L} = \{L_0, L_1, \ldots, L_h\}$ be a partition of the node set of $G$ 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 $L_0, L_1, \ldots, L_h$ are called layers. A DAG with a layering is called a layered DAG. In the remainder of this paper we assume that in a visual representation of a layered DAG all nodes in layer $L_i$ are placed on the horizontal level with an $y$-coordinate $i$. Thus, we say that $L_j$ is above $L_i$ and $L_i$ is below $L_j$ if $i < j$.

Let $l(u, \mathcal{L})$ denotes the number of a layer which contains node $u \in V$, i.e. $l(u, \mathcal{L}) = i$ if and only if $u \in L_i$. Then the span of edge $e = (u, v)$ in layering $\mathcal{L}$ is 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 can be layered properly and also because a proper layering may not satisfy other layering requirements.

In the Sugiyama method for drawing DAGs the node ordering algorithms applied after the layering phase assume that their input is a DAG with a proper layering. Thus, if the layering found at the layering phase is not proper then
Fig. 2. The drawings from Figure 1 with introduced dummy nodes which subdivide long edges. Dummy nodes are represented by transparent squares.

It must be transformed to a proper one. Normally, this is done by introducing so-called dummy nodes which subdivide long edges (see the illustration in Figure 2). Formally, let \( e = (u, v) \) be an edge with \( l(u, \mathcal{L}) = j \) and \( l(v, \mathcal{L}) = i \) and \( s(e, \mathcal{L}) = j - i > 1 \). Then we add dummy nodes \( d_{e}^{i+1}, d_{e}^{i+2}, \ldots, d_{e}^{j-1} \) to layers \( L_{i+1}, L_{i+2}, \ldots, L_{j-1} \) respectively and we replace edge \( e \) by the path \( (u, d_{e}^{i+1}, \ldots, d_{e}^{j-1}, v) \). To distinguish the original nodes of a DAG from the dummy nodes we refer to the former as regular nodes. We also denote the set of all dummy nodes introduced to a layered DAG \( G \) with a layering \( \mathcal{L} \) by \( \mathcal{D}(G, \mathcal{L}) \). Clearly,

\[
|\mathcal{D}(G, \mathcal{L})| = \sum_{e \in E} s(e, \mathcal{L}) - |E|.
\]

It is desirable that \( |\mathcal{D}(G, \mathcal{L})| \) is as small as possible because a large number of dummy nodes significantly slows down the node ordering phase of the Sugiyama method. Thus, one of the goals of a layering algorithm should be to find a layering with as few as possible dummy nodes. There are also aesthetic reasons for keeping the number of dummy nodes small. A layered DAG with a small number of dummy nodes would also have a small number of undesirable long edges and edge bends.

Other parameters of a layering which reflect on the quality of the drawing are the width and the height of a layering and the edge density between adjacent horizontal levels. The height of a layering is the number of layers, and the width is the maximum number of nodes in a layer. Usually these two parameters are used to approximate the dimensions of the final drawing. When measuring the width of a layering the contribution of the dummy nodes may or may not be taken into account. A more precise definition of the layering width takes
into account both variable node widths and the contribution of the dummy nodes \cite{1,8}. The area of a layering, used to approximate the area of the final drawing, is defined as the product of the layering width and the layering height.

The edge density between horizontal levels \( i \) and \( j \) with \( i < j \) is defined as the number of edges \((u, v)\) with \( u \in L_j \cup L_{j+1} \cup \ldots \cup L_h \) and \( v \in L_0 \cup L_1 \cup \ldots \cup L_k \). The edge density of a layered DAG is the maximum edge density between adjacent layers (horizontal levels). Naturally, drawings with low maximum and average edge density are clear and easier to comprehend.

In the next section we review the known layering algorithms and discuss the quality of layerings found by them.

3 Existing Layering Algorithms

At present there are two groups of layering algorithms which find a layering of a DAG subject to some of the above criteria. 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 nodes per layer. The second group of algorithms employ network simplex and branch-and-cut techniques respectively for minimizing the number of dummy nodes.

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 also known as static scheduling because all the tasks are given in advance and the schedule must be constructed prior executing any of them. A simplified version of this problem, where 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 \( \pi \) be the number of nodes in the longest directed path in a DAG. The longest path algorithm builds the scheduling list by assigning priority \( \pi \) 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 \( L_{\pi-k} \). The advantages of the longest path algorithm are its simplicity and its linear time complexity. It also produces layerings with the minimum height. However, it performs very poorly in terms of drawing area, number of dummy nodes and edge density [9].

The second list scheduling algorithm used for DAG layering is the Coffman-Graham algorithm [3] which is based on an earlier algorithm by Hu [10]. It approximately solves the static scheduling problem for \( m < \infty \) which is NP-hard [17]. The technique used for building the scheduling list is more complex than the one used by the longest path algorithm. The worst-case time complexity of the Coffman-Graham algorithm is \( O(V^2) \). It guarantees a layering with at most \( m \) nodes per layer and in the worst case the height of the layering may become close to twice the optimal height [3]. 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 [9]. 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 [3] as well as in several scheduling and graph drawing publications [5,11].

It can be noted that since the introduction of the Coffman-Graham algorithm a large number of alternative list scheduling algorithms have been proposed for solving variations of the static scheduling problem. In addition, there have been proposed alternatives to list scheduling. Most of them assume a more complex static scheduling model which is closer to real scheduling problems and is not directly equivalent to a DAG layering problem. However, it might prove fruitful translating the various static scheduling techniques into DAG layering techniques. A recent survey by Kwok and Ahmad introduces a taxonomy of the multiprocessor scheduling problems and covers a big variety of static scheduling techniques [11].
3.2 Integer Linear Programming Approaches

The first algorithm that generates layerings with the minimum number of dummy nodes is the algorithm introduced by Gansner, Koutsofios, North and Vo [7]. They model the problem by the following integer linear program:

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

The linear programming relaxation of this integer program has always an integer solution because its constraint matrix is totally unimodular [14]. Thus, the integer program can be solved by the simplex method. Gansner et al. go further by introducing a network simplex based algorithm for solving this problem [7]. Their algorithm has not been proven polynomial but reportedly requires a few iterations and runs fast. They also perform a balancing step after the layers have been determined. In the balancing step nodes with equal in- and out-degree and which can be moved up or down without destroying the layering are 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 and improved aspect ratio of the drawing.

The layering algorithm of Gansner et al. (even without the balancing step) performs significantly better than the list scheduling layering algorithms. Layered DAGs with the minimum number of dummy nodes have also much lower edge density and considerably smaller layering area even without any explicit control on the layering dimensions [9]. However, there are some exceptions. For instance, the drawings in Figure 3 show that a layered DAG with the minimum number of dummy nodes may become much wider than required.

The branch-and-cut layering algorithm introduced by Healy and Nikolov [8] solves such cases by minimizing the number of dummy nodes subject to upper bounds on the height and the width of the layering and taking into account variable node dimensions as well as the dummy node contribution to the layering width. 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 branch-and-cut 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. [8].
Fig. 3. Two alternative layerings of the same DAG which show that a layering with the minimum number of dummy nodes may become too wide.

4 The Promote Layering Heuristic

The fact that the integer linear program proposed by Gansner et al. can be solved by a linear programming solver shows that the problem of finding a layering with the minimum number of dummy nodes has a polynomial time complexity. The motivation behind the work we present here was to develop a simple and easy to implement layering method for decreasing the number of dummy nodes in a DAG layered by some list scheduling algorithm. Such a layering method should be considerably easier to implement than the network simplex algorithm of Gansner et al. and would prove useful when a commercial linear programming solver is not available.

We found out that a very simple improvement heuristic, applied after the longest path layering algorithm leads to layerings which do not have the minimum number of dummy nodes but do have very similar characteristics to layerings with the minimum number of dummy nodes. A surprising result from applying the new heuristic was that in some cases it leads to layerings with lower edge density and smaller area than those of the layerings with the minimum number of dummy nodes. We call our improvement heuristic Promote Layering or PL and we refer to the layering method that consists of applying PL to the output of the longest path algorithm as LPath+PL. We evaluate LPath+PL in the next section and in the remainder of this section we introduce PL in detail.

4.1 Layering-Preserving Promotion

PL modifies a given layering \( \mathcal{L} = \{L_0, L_1, \ldots, L_h\} \) of a DAG \( G \) by promoting regular nodes from the layer where they are placed to the layer above. 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 \( \mathcal{L}^* = \{L_0, \ldots, L_k \setminus \{v\}, L_{k+1} \cup \{v\}, \ldots, L_h\} \). 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 promoted to 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 is any) have to be promoted to layer \( L_{k+2} \); the same applies to their immediate predecessors and so on. This is illustrated in Figure 4. In the initial layering in Figure 4(a) node \( d \) is placed in layer \( L_0 \). If we promote it to layer \( L_1 \) (see Figure 4(b)) the layering is destroyed because edge \((c,d)\) does not point downwards. Thus, it is necessary to promote node \( c \) to layer \( L_2 \) in order to preserve the layering (see Figure 4(c)). We call this recursive mechanism of promotion a layering-preserving promotion.

Each layering-preserving promotion of a node changes the total number of dummy nodes. In order to express the change consider node \( v \) which we promote from layer \( L_k \) to layer \( L_{k+1} \) in a given initial layering \( \mathcal{L} \). Let \( v \) have \( s \) immediate successors, \( p \) immediate successors and let \( u_1, u_2, \ldots, u_p \) \((0 \leq p \leq p)\) be all its immediate predecessors placed in layer \( L_{k+1} \). Then the change in the number of dummy nodes after the layering-preserving promotion of \( v \) can be recursively defined as

\[
dummydiff(v, \mathcal{L}) = s - p + \sum_{i=1}^{p} dummydiff(u_i, \mathcal{L})
\]

Note that everywhere in the expression above \( \mathcal{L} \) refers to the initial layering.
As an example consider again the promotion of node $d$ illustrated in Figure 4. 

\[ \text{dummydiff}(c, \mathcal{L}) = 2 - 1 = 1, \]
\[ \text{dummydiff}(d, \mathcal{L}) = 0 - 3 + \text{dummydiff}(c, \mathcal{L}) \]
\[ = 0 - 3 + 1 = -2. \]

That is, by promoting node $d$ in the layering shown in Figure 4(a) we reduce the number of dummy nodes by two. Indeed, the number of dummy nodes in Figure 4(a) is 4 and the number of dummy nodes in Figure 4(c), after the recursive promotion of $d$, is 2.

The recursive function \texttt{PromoteNode}, shown in Algorithm 1, performs a layerings-preserving promotion of node $v$ from layer $L_k$ to layer $L_{k+1}$ in layering $\mathcal{L}$. It returns \texttt{dummydiff} which represents \texttt{dummydiff}($v, \mathcal{L}$). 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 \texttt{dummydiff}. Then we promote $v$, subtract from \texttt{dummydiff} the number of immediate predecessors of $v$, 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 \texttt{PromoteNode} is $O(|E|)$ because in the worst case all DAG edges might be traversed while promoting nodes recursively.

**Algorithm 1 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 $v \in V$.

\[ \text{dummydiff} \leftarrow 0 \]

\textbf{for all} $u \in N_\mathcal{G}^-(v)$ \textbf{do}

\[ \text{if} \ \text{layering}[u] = \text{layering}[v] + 1 \ \text{then} \]

\[ \text{dummydiff} \leftarrow \text{dummydiff} + \text{PromoteNode}(u) \]

\[ \text{layering}[v] \leftarrow \text{layering}[v] + 1 \]

\[ \text{dummydiff} \leftarrow \text{dummydiff} - N_\mathcal{G}^-(v) + N_\mathcal{G}^+(v) \]

\textbf{return} \text{dummydiff}

4.2 The Heuristic

\texttt{PL} consists of two nested loops shown in Algorithm 2, an external \texttt{repeat-until} loop and an internal \texttt{for} loop. In the internal loop all nodes in a layered DAG are scanned in no particular order and each node with a positive in-degree gets promoted by \texttt{PromoteNode} (see Algorithm 1) if its layerings-preserving promotion reduces the total number of dummy nodes. The external loop goes on until the internal loop makes no promotion.
Algorithm 2 PL

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 for
until $promotions = 0$

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 $O(|V| \times (|V| + |E|))$.

The internal loop in Algorithm 2 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, this 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. Thus in total the worst-case time complexity of the PL is $O(|D(G, L)| \times |V| \times (|E| + |V|))$. The best known estimate of the number of dummy nodes in a layered DAG is $O(min\{|V|^3, |E|^2\})$ [12]. A tighter upper bound is known only for layered DAGs with the minimum height [6].

In the next section we compare the performance of LPath+PL to the longest path layering algorithm and to the algorithm of Gansner et al.

5 Experimental Results

For the evaluation of LPath+PL (the longest path layering algorithm followed by PL) we used three benchmark graph sets: the Rome graphs introduced
by Di Battista et al. [4], the set of AT&T graphs\(^1\) and a set of randomly generated DAGs available at [http://www.graphdrawing.org](http://www.graphdrawing.org) which we refer to as GDorg DAGs.

The Rome graph set consists of 11,530 graphs in LEDA [13] 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 a directed cycle. We also filtered out the unconnected graphs leaving 5911 DAGs. The AT&T DAG set and the GDorg DAG set consist of 1277 and 909 DAGs respectively. All the three sets contain graphs with up to 100 nodes. A typical DAG from each of the three DAG sets with \(n\) nodes has 1.6\(n\) edges, however, there are a few much denser DAGs in the AT&T DAG set. All of the GDorg DAGs are biconnected while among the Rome and the AT&T DAGs there are only a few biconnected DAGs. Since the computational results we obtained for the three DAG sets lead to the same conclusions about LPath+PL, for brevity, here we present only the results for the Rome DAGs which are the biggest of the three DAG sets.

In the remainder of this section we compare the quality of the layerings produced by LPath+PL to those produced by the longest path layering algorithm and by the algorithm of Gansner et al. We refer below to the longest path algorithm as LPath and to the algorithm of Gansner et al as Gans. Our implementation of Gans directly uses CPLEX 7.0 for solving the integer layering program discussed in Section 3.2 and we do not perform the post-layering balancing step. Note that the balancing step does not change the number of dummy nodes and the edge density. It may reduce the maximum number of regular nodes per layer and thus it may impact on the layering area and aspect ratio.

For comparing the performance of the three layering algorithms we separated the DAGs into "buckets" according to their node count, putting a DAG of \(n\) nodes into bucket \([n/5]\). The values in Figures 5 and 7 are the average for a bucket.

The first plot in Figure 5(a) compares the number of dummy nodes in layerings generated by the three layering methods divided by the total number of regular nodes in a DAG. We observe a significant reduction of the number of dummy nodes achieved by LPath+PL. It is easy to see why PL does not minimize the number of dummy nodes. Figure 6 shows an example of a longest path layering which cannot be improved by PL. The longest path layering in Figure 6(a) can be improved by moving \(v\) together with all of its successors upwards until the number of dummy nodes becomes zero. However, PL will reject both the layering-reserving promotion of \(v\) (illustrated in Figure 6(b)) and the layering

\(^1\) The AT&T graphs are available at [http://www.research.att.com](http://www.research.att.com).
(a) Distribution of the number of dummy nodes by node count.  
(b) Distribution of the edge density by node count.

Fig. 5. Rome DAGs: Number of dummy nodes and edge density.

(a) Longest path layering.  
(b) Layering-preserving promotion of $v$ which is rejected by PL.  
(c) Layering-preserving promotion of $u$ which is rejected by PL.

Fig. 6. Longest path layering which cannot be improved by PL.

preserving promotion of a successor of $v$ (illustrated in Figure 6(c)) because neither of them reduces the total number of dummy nodes on its own.

The slightly higher than the minimum number of dummy nodes is compensated by lower edge density observed in LPath+PL layerings in Figure 5(b). (The edge density values in Figure 5(b) are divided by the total number of regular nodes in a DAG.) The edge density results for the AT&T DAGs and for the GDorg DAGs show slightly better performance of Gans but there is no clear winner.

Figures 7(a) and 7(b) show the results for area (the product of the layering width and the layering height) without and with taking into account the contribution of the dummy nodes to the layering width respectively. Since the test DAGs do not have node labels (except node numbers) we have assumed
that all regular nodes have unit width. The width of the dummy nodes in Figure 7(b) is 1, that is, a dummy node makes the same contribution to the width as a regular node - a charge that is at the upper limit of what seems reasonable. In both cases LPath+PL performs best. With the same assumption about the node widths, Figures 7(c) and 7(d) show results for the aspect ratio (width/height) with and without taking into account the dummy nodes respectively. On the basis of the area and aspect ratio results it can be concluded that Gans layerings tend to be slightly taller and narrower than LPath+PL layerings. It has to be noted that a balancing step, applied after Gans, may slightly reduce the area and bring the aspect ratio closer to the aesthetically most desirable golden mean when dummy nodes are not taken into account. However, it is unlikely to change the picture when the contribution of dummy nodes is considered.

We have performed all the tests on a PC with a 600 MHz Intel Pentium III processor. Although slower than Gans, LPath+PL generates a layering of a DAG (from any of the three data sets) within 0.04 seconds on average.
6 Conclusions

The proposed improvement heuristic, PL, is a very simple polynomial-time algorithm which is much easier to implement than the algorithm of Gansner et al. Although PL does not always minimize the number of dummy nodes in a layered DAG, it does lead to layerings with high aesthetic qualities and low number of dummy nodes when applied after the longest path layering algorithm. The worst-case time complexity of PL is $O(|D(G, \mathcal{L})| \times |V| \times (|E| + |V|))$ where $|D(G, \mathcal{L})|$ is the number of dummy nodes in a layering $\mathcal{L}$ of a DAG $G$. Although slower than the algorithm of Gansner et al., the longest path algorithm followed by PL performs fast enough when applied to DAGs with up to 100 nodes taken from practical applications (the Rome and the AT&T graph sets).

As a further step, it might be fruitful to adopt some of the more recent static scheduling algorithms for DAG layering and to experiment with applying PL for improving the quality of their output. Also, it would be interesting to study the impact of an improvement technique as PL from the static scheduling point of view.

References


