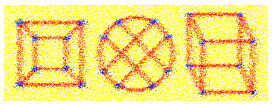
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 12, no. 3, pp. 293–317 (2008)

Using a Significant Spanning Tree to Draw a Directed Graph

Martin Harrigan[1] *Patrick Healy*[1]

[1]Department of Computer Science and Information Systems
University of Limerick

Abstract

A directed graph can model any ordered relationship between objects. However, visualizing such graphs can be a challenging task. If the graph is undirected, a popular strategy is to choose a significant spanning tree, nominate a vertex as the root, for example the vertex whose distance from all other vertices is minimal, hang the significant spanning subtrees from this root and add in the remaining edges in some unobtrusive manner [19, 26, 27, 33]. In the directed case the spanning tree is a tree DAG and not simply a directed tree with one appropriate root. It may have multiple sources that all warrant root status and so the undirected approach must be modified somewhat. In this paper, we present a method of drawing directed graphs that emphasizes a significant spanning tree. It combines two steps of the Sugiyama framework [31] (leveling and crossing minimization) by finding, in linear time, a leveling of the graph that is level planar with respect to some spanning tree and restricting the permutations of the vertices on each level to those that constitute a level planar embedding of this subgraph. The edges of the spanning tree will therefore not cross each other. Using a globally oriented Fiedler vector we choose permutations of the vertices on each level that reduce the number of edge crossings between the remaining edges.

Submitted:	Reviewed:	Revised:	Accepted:	Final:
July 2007	September 2007	January 2008	March 2008	July 2008

Published:
October 2008

Article type: Regular Paper
Communicated by: S.-H. Hong

Supported by the Irish Research Council for Science, Engineering and Technology.

E-mail addresses: martin.harrigan@ul.ie (Martin Harrigan) patrick.healy@ul.ie (Patrick Healy)

1 Introduction

A popular strategy when drawing graphs is to extract a spanning tree or hierarchy, draw it using some simpler algorithm and then handle the remaining edges. When the graph is undirected this spanning tree is a free tree and we can nominate some vertex as the root [3], draw the spanning tree using the simpler tree drawing algorithms [29, 32, 5] and add in the remaining edges. However, in the directed case the spanning tree is not necessarily a directed tree with one single source; it is a tree DAG (a directed graph without any cycles, directed or undirected) with potentially multiple sources. It is inappropriate to choose one of these vertices arbitrarily as the root and so the simpler tree drawing algorithms are not suitable.

We propose finding a significant spanning tree DAG T using either domain-specific (*e.g.* edge weights associated with the graph) or graph-theoretic knowledge. We find a leveling of the graph that is level planar with respect to T by inserting a small number of dummy vertices and restrict the permutations of the vertices on each level to those that constitute a level planar embedding of T. In this way we ensure that any edge crossings in the final drawing do not involve two significant edges. We use a globally oriented Fiedler vector to choose permutations of the vertices on each level that reduce the number of edge crossings between the remaining edges. Our main contributions are: a proof that minimizing the number of dummy vertices, even in a simple 'base case', is NP-Hard, a heuristic method that runs in linear time and a demonstration of how this approach can be used to draw entire directed graphs.

Our approach can be considered a variation of the Sugiyama framework [31]. This framework temporarily removes any directed cycles by reversing a small number of edges, creates a proper leveling, permutes the vertices on each level to reduce the number of edge crossings and balances the layout. An alternative goal when permuting the vertices on each level is to maximize the size of a level planar subgraph. Even if there are only two levels and the vertices on one are fixed, both the crossing minimization and maximum level planar subgraph problems are NP-Hard [11, 10] and so heuristics are generally employed. Once we have computed a Fiedler vector, our method efficiently combines the leveling and crossing minimization steps in linear time. We particularly favor our method when we want a spanning tree of the directed graph to be clear and apparent from the drawing.

This paper is organized as follows. We begin with some preliminaries in Sect. 2. In Sect. 3 we show how to choose a significant spanning tree DAG T. In Sect. 4 we present a recursive algorithm that ensures the level planarity of T by inserting a small number of dummy vertices. This also implies a simple embedding algorithm. In Sect. 5 we describe a crossing minimization heuristic based on a Fiedler vector and show how it can be combined with the previous step. Section 6 brings all the parts together to draw the dependency relationship between packages related to the Python programming language and Sect. 7 gives some concluding remarks.

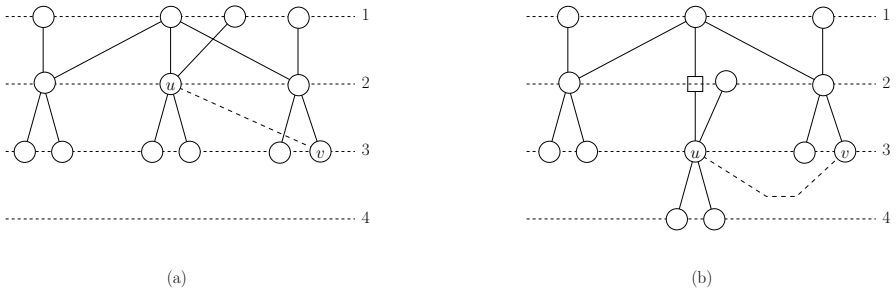

Figure 1: A non-level planar and level planar drawing of spanning tree DAGs (solid edges). All edges point downwards.

2 Preliminaries

A *leveling* of a directed graph $G = (V, E)$ is a surjective mapping $\phi : V \to \{1, 2, \ldots, k\}$ such that $\phi(v) \geq \phi(u) + 1, \forall (u, v) \in E$. A leveling is *proper* if the relation is strictly equality. If a leveling is not proper then we can make it so by inserting dummy vertices along edges than span more than one level. In the following we assume all levelings to be proper. A *level drawing* of G with leveling ϕ is a drawing in which the vertices in $\phi^{-1}(j), 1 \leq j \leq k$, are placed on a horizontal level l_j and the edges are drawn as straight-line segments between the vertices. A level drawing of G with leveling ϕ is *level planar* if no two edges cross except at common endpoints. G with leveling ϕ is level planar if it has a level planar drawing. A *level embedding* of G with leveling ϕ consists of total orders $<_j$ of the vertices in $\phi^{-1}(j), 1 \leq j \leq k$. A level embedding is level planar if any level drawing of the graph in which the order of the vertices along l_j satisfy $<_j, 1 \leq j \leq k$, is level planar.

Every tree DAG T (a directed graph with no cycles, directed or undirected) has a leveling ϕ such that all edges are directed uniformly. If T with leveling ϕ is not level planar then we can make it so by inserting dummy vertices along appropriate edges (see the spanning tree DAGs in Fig. 1). Accomplishing this with a small number of dummy vertices is the main concern of Sect. 4.

The *Laplacian* $\mathcal{L}(G)$ of a graph G is a square matrix defined by

$$\mathcal{L}(G)_{u,v} = \begin{cases} \mathtt{deg}(v) & \text{if } u = v, \\ -1 & \text{if } (u, v) \in E \vee (v, u) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

$\mathcal{L}(G)$ is positive semi-definite and so its eigenvalues, $\lambda_1 \leq \ldots \leq \lambda_n$, are all nonnegative. An eigenvector corresponding to the second smallest eigenvalue is known as a *Fiedler vector* and can be calculated using power iteration [20] on the matrix $\lambda_n I - \mathcal{L}(G)$ or more efficiently using multi-scale methods [13].

3 Choosing a Significant Spanning Tree DAG

Let $G = (V, E)$ be a directed graph. We assume the graph is connected; otherwise we handle each connected component separately. We seek a significant spanning tree DAG $T = (V, E_T)$ of G, *i.e.* one in which the edges in E_T are deemed more significant than those in $E \setminus E_T$. This can be done using domain-specific knowledge or any of the numerous methods for measuring significance (*e.g.* spanning tree algorithms [22, chap. 2] and structural indices [24]). Recently, [25] combined a number of measures to find a spanning tree or 'backbone' with a very low average spanner property for the non-tree edges and used it as input to an undirected tree drawing algorithm. The method we choose in this paper is to weight each edge $(u, v) \in E$ by $(\vec{x}_u - \vec{x}_v)^2$ where $\vec{x}$ is a Fiedler vector of $\mathcal{L}(G)$ and $\vec{x}_u$ denotes the entry associated with vertex u. T is then a minimum weighted spanning tree DAG of G.

Our choice of T is primarily due to the fact that we will be re-using $\vec{x}$ later. However, we give two reasons to justify T as being a somewhat significant spanning tree DAG. Firstly, consider the problem of embedding G in the line so that all edge lengths are kept short. If the location of $v \in V$ in the line is $\vec{x'}_v$ then we want to minimize $\sum_{(u,v)\in E} (\vec{x'}_u - \vec{x'}_v)^2$ subject to $\vec{x'}\vec{x'}^T = \vec{1}$ (to avoid the trivial solution of setting $\vec{x'} = \vec{0}$). It turns out that a solution is precisely $\vec{x}$. Secondly, $\vec{x}_u$ can be interpreted as a measure of u's 'degree-normalized' *eigenvector centrality* [2] with E_T comprising the edges that propagate most of this centrality through G. Eigenvector centrality is a measure of the significance of a vertex in a graph. It assigns relative values to all vertices based on the principle that incident edges from high-value vertices contribute more to the value of the vertex in question than incident edges from low-value vertices [24].

However, since the choice of a meaningful spanning tree is crucial, our method is more suited to graphs whose edges are weighted *a priori* by domain-specific knowledge.

4 Ensuring Level Planarity

Level planarity can be tested in $\mathcal{O}(|V|)$ time using the PQ-tree data structure [18, 21] or in $\mathcal{O}(|V|^2)$ time using the simpler Vertex-Exchange Graph [16, 15]. However, extending either method to ensure level planarity for the special case of tree DAGs by inserting a small number of dummy vertices is not obvious.

Our algorithm `makeTreeDAGLevelPlanar` is based on the recursive nature of a tree DAG. It processes a tree DAG T with leveling ϕ by recursively decomposing T into smaller tree DAGs with fewer vertices of in-degree greater than one. It computes a matrix $M(T)$ at each step where each column of $M(T)$ represents the restrictions imposed on the level planarity of T by each smaller tree DAG.

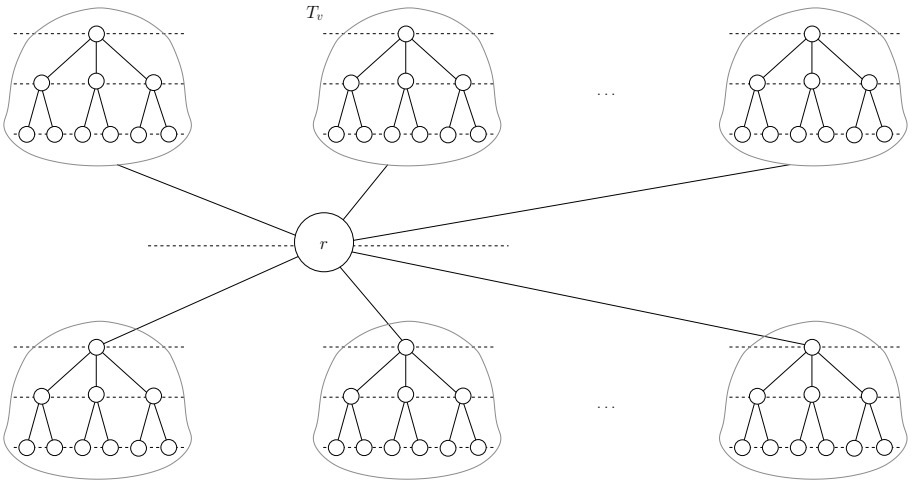

Figure 2: The base case has exactly one vertex with in-degree greater than one.

4.1 The Base Case

Let $T = (V, E_T)$ be a tree DAG with leveling ϕ and $\mathcal{V} = \{v \in V : \texttt{inDeg}(v) > 1\}$. We assume $|\mathcal{V}| > 0$ since otherwise T with ϕ is trivially level planar. In the base case $|\mathcal{V}| = 1$ and we let this vertex be r. The outgoing and incoming neighbors of r, $\mathcal{N}^+(r)$ and $\mathcal{N}^-(r)$, join r to the roots and vertices of distinct directed trees respectively (see Fig. 2). We use the notation $T_v = (V_v, E_v)$ to refer to the maximal subtree that is joined to r through some $v \in \mathcal{N}^+(r) \cup \mathcal{N}^-(r)$. To ensure level planarity, we need only insert dummy vertices along edges joining $\mathcal{N}^-(r)$ to r.

We proceed by populating a matrix $M(T)$. For each $v \in \mathcal{N}^-(r)$ we add a column to $M(T)$ as follows. We visit each vertex v' along the (undirected) path from v to the root of its respective tree T_v. If there exists some vertex w that is a descendant of v' in T_v such that $\phi(w) \geq \phi(r)$ then we set the entry in row $\phi(v')$ of the new column to $\phi(w)$. Otherwise we leave the entry empty. We note the edge (v, r) that corresponds to each column.

$M(T)$ is a $k \times |\mathcal{N}^-(r)|$ matrix (where k is the number of levels) consisting of partially populated columns of consecutive non-increasing entries. Entries containing 0 are considered 'unpopulated'. The rows with populated entries lie in the range 1 to $\phi(r) - 1$. Each row represents the restrictions imposed on the level planarity of T by directed subtrees whose roots lie on the corresponding level. Each column represents the restrictions imposed on the level planarity of T by the corresponding T_v. From here on, by referring to, say, the first entry in a column, we mean the first populated entry. We use $\texttt{first}(M(T), j)$ and $\texttt{last}(M(T), j)$ to denote the index of the first and last entries in column j respectively.

In the following lemma we show that if there are at most two entries in any

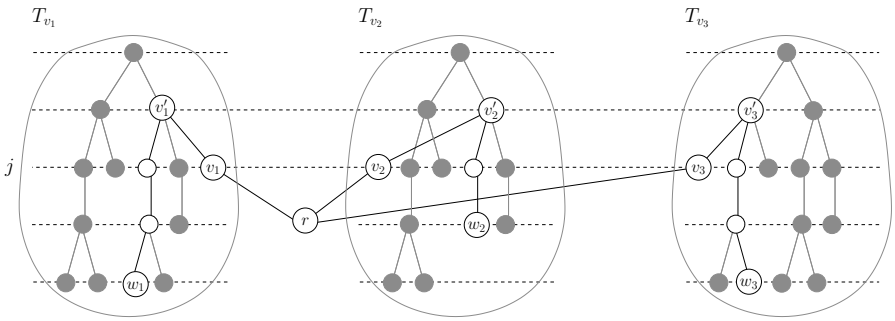

Figure 3: If row j of $M(T)$ has more than two (populated) entries then T with leveling ϕ is not level planar.

one row of $M(T)$, each T_v can 'hang' over one side of r.

Lemma 1. *If T is a tree DAG with leveling ϕ and has one vertex of in-degree greater than one then T is level planar if and only if $M(T)$ has at most two (populated) entries in any one row.*

Proof: Suppose row j of $M(T)$ has three entries. Let $\{v_i\} \subseteq \mathcal{N}^-(r)$ and $T_{v_i} = (V_{v_i}, E_{v_i})$, $1 \leq i \leq 3$, such that T_{v_i} is a directed tree whose column in $M(T)$ contains an entry in row j. Let $v_i', w_i \in V_{v_i}$ such that $\phi(v_i') = j$ and $\phi(w_i) \geq \phi(r)$, $1 \leq i \leq 3$. For any total order $<_j$ of the vertices in $\phi^{-1}(j)$ there must be at least one crossing in any level drawing of T (see Fig. 3). However, the crossings are avoidable if there are at most two populated entries in row j. □

To insert a dummy vertex along an edge between some $v \in \mathcal{N}^-(r)$ and r we decrement each entry in the corresponding column, shift the column up one row (adding extra rows to the top of $M(T)$ if necessary) and repeatedly remove any last entry in the column whose value is now less than $\phi(r)$. We call this operation `shiftUp`$(M(T), j)$ where j is the index of the column to shift.

We wish to apply `shiftUp`$(M(T), j)$ a small number of times so that each row contains at most two entries. Lemma 1 suggests the following heuristic. Find the last row with more than two entries. Fix two columns whose last entries occur the latest (j_{L} and j_{R}). If more than two columns meet this criterion, choose two of these whose first entries occur the latest. If more than two columns meet both criteria, then the choice between these columns is arbitrary. Apply `shiftUp`$(M(T), j)$ to all other columns that have an entry in this row. Then go to the previous row and repeat. Once all the rows have been processed then $M(T)$ is left with at most two (populated) entries in any one row. Therefore inserting the new dummy vertices into T ensures level planarity. We will address the time complexity of this process in Sect. 4.4 and provide a complete worked example in Sect. 4.5. Before proceeding to the recursive case, we briefly consider the computational complexity of minimizing the number of dummy vertices for the base case.

4.1.1 Interpreting the Base Case as a Scheduling Problem

The problem of inserting a small number of dummy vertices to ensure the level planarity of T for the base case can be interpreted as a simple task scheduling problem. There are two identical processors, p_{L} and p_{R}, where p_{L} handles the tasks or columns assigned to j_{L} and likewise with p_{R} and j_{R}. There are at most $|\mathcal{N}^-(r)|$ independent tasks. The tasks are released in the (descending) order specified by $\texttt{last}(M(T), j)$ where j is the index of the corresponding column. The processing times for each task are the number of entries in the corresponding column but each task has the added peculiarity that making it wait (*i.e.* inserting a dummy vertex) may result in the task becoming temporarily unavailable and requiring a decreased processing time. Our goal is to minimize the sum of the waiting times for all available tasks so we are employing a shortest task first algorithm. This is optimal if the value of each entry is sufficiently large so that the peculiarity does not arise.

This interpretation shows the difficulty in minimizing the number of dummy vertices even for the base case. The peculiarity we referred to involves tasks with time dependent processing times: the length of the task depends on the time at which it is started. In classical task scheduling theory, task processing times are constant; however, there is a growing interest in scheduling models with time-dependent processing times, see, *e.g.* the survey of [1]. We are particularly interested in a result of [7]. They prove that minimizing the total length of a schedule (the 'makespan') for a set of tasks on one processor with arbitrary release times and whose processing times decrease linearly at individual rates with respect to their starting times is strongly NP-Hard. The problem can be stated more formally as follows:

Problem 1 (MIN-MAKESPAN). *We are given a set of n independent tasks $\{T_j\}$. Each task T_j has a release time r_j, an initial processing time α_j and a decreasing processing rate w_j. r_j, α_j and w_j are all rational numbers. If a task T_j is scheduled to start at time $s_j \geq r_j$, then its actual processing time is $\alpha_j - w_j s_j \geq 0$. We wish to find a schedule on one processor that minimizes the makespan, $\sum_{i=1}^{n} p_i$.*

[7] proved that:

Theorem 2. *[7]*
MIN-MAKESPAN *is strongly* NP*-Hard.*

We can formally state our problem as:

Problem 3 (MIN-DUMMY-VERTICES). *Given a tree DAG T with leveling ϕ and exactly one vertex of in-degree greater than one, find a minimum number of dummy vertices to ensure level planarity.*

This brings us to our result:

Theorem 4. MIN-DUMMY-VERTICES *is* NP*-Hard.*

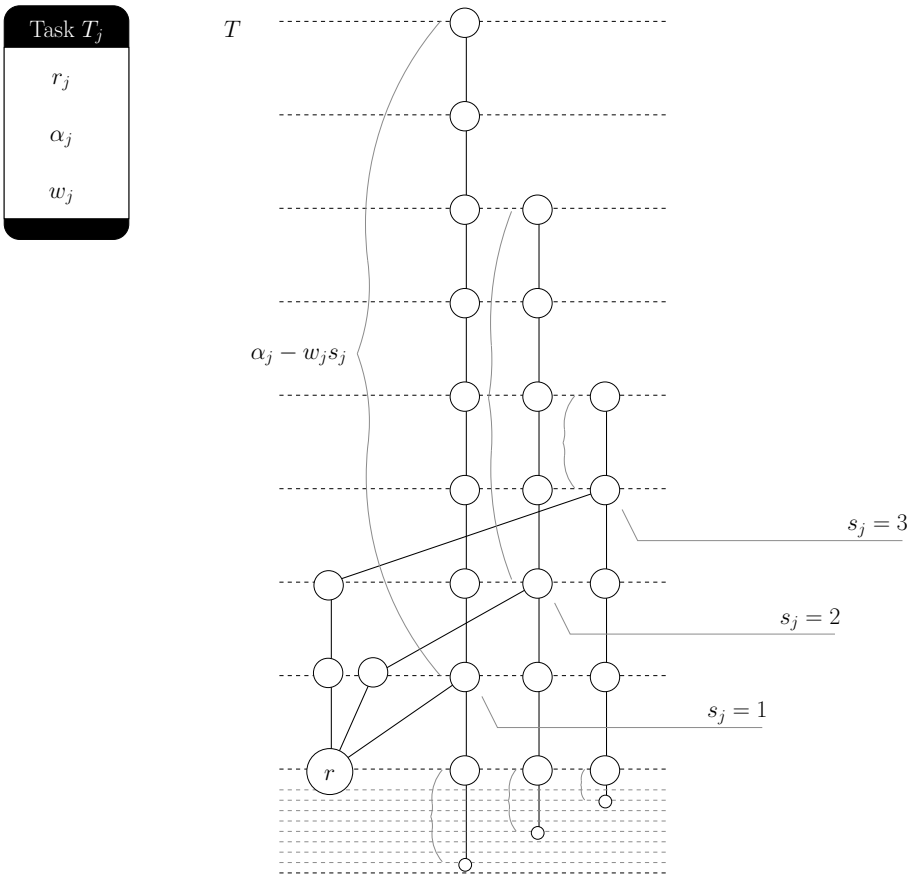

Figure 4: Every possible start time $s_j \geq r_j$ for a task T_j such that the actual processing time is greater than zero contributes a single directed tree to T.

Proof: We reduce MIN-MAKESPAN to MIN-DUMMY-VERTICES. The parameters in MIN-MAKESPAN may be rational so we first scale them up to integers. Then, for each task T_j, we add a group of directed trees to a tree DAG T. Every possible start time $s_j \geq r_j$ such that the actual processing time $\alpha_j - w_j s_j > 0$ contributes a single directed tree to this group.

More specifically, we create a tree DAG T with one vertex r of in-degree greater than one. T has a leveling ϕ with both *macro-* and *micro-levels*. The macro-levels are numbered 1 to $M = \max_j\{\alpha_j - w_j r_j\} + 2$. There are $\mu = \max_j\{\alpha_j - w_j r_j\}$ micro-levels in between each pair of consecutive macro-levels. We refer to a macro-level as being the p^{th} macro-level and to a micro-level as being the q^{th} micro-level above or below the p^{th} macro-level, $\forall 1 \leq q \leq \mu$ and $1 \leq p \leq M$. We place r on the $M - 1^{\text{th}}$ macro-level.

For each task T_j, we add a group of directed trees to T and join each one to r as follows. We add a directed tree for every possible start time $s_j \geq r_j$

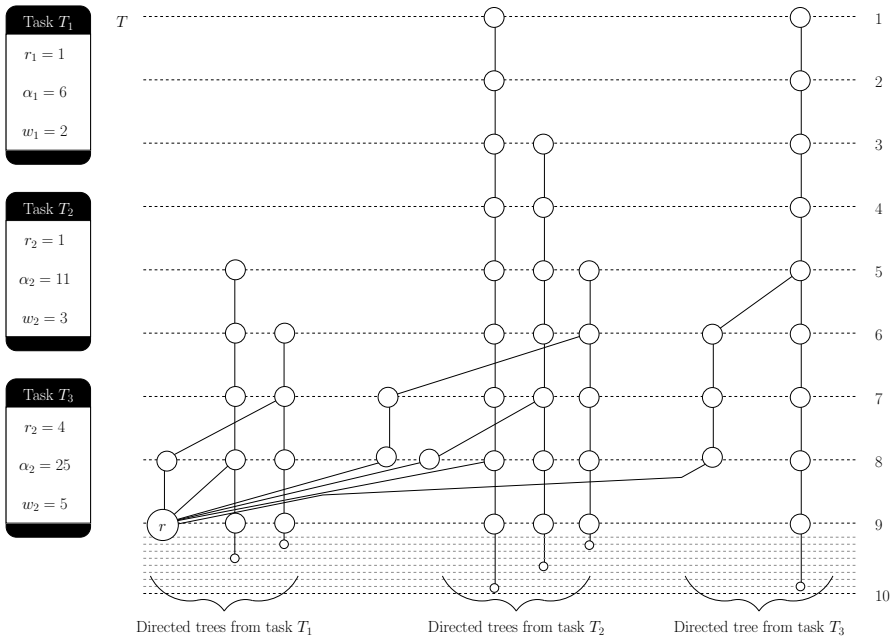

Figure 5: The three tasks T_1, T_2 and T_3 contribute three groups of directed trees to T.

such that the actual processing time $\alpha_j - w_j s_j > 0$. We place the root of this directed tree on the $M - (\alpha_j - w_j s_j) - s_j - 1^{\text{th}}$ macro-level, extend a branch down to a vertex on the $M - s_j - 1^{\text{th}}$ macro-level, then extend one branch from here to join r and another down to a vertex on the $(\alpha_j - w_j s_j)$th micro-level below the $M - 1^{\text{th}}$ macro-level. Figure 4 exemplifies this process for one task which results in the addition of three directed trees to T.

We repeat this process for each task. Figure 5 shows three groups of directed trees representing three tasks; the first having two directed trees, the second having three and the third having just one.

Finally, when the groups of directed trees for each task have been added, we add one last directed tree. We place the root of this tree on the 1^{st} macro-level, extend a branch to a vertex on the $M - 2^{\text{th}}$ macro-level, then extend one branch from here to join r and another down to a vertex on the M^{th} macro-level. In effect, this directed tree keeps one of the two 'processors' (sides of r), p_{L} or p_{R} (j_{L} or j_{R}), occupied throughout. This is required since MIN-MAKESPAN is formulated in terms of one processor.

Within each group of directed trees for some task T_j, at most one directed tree can 'hang' on the free side of r. All others will be pushed up by the insertion of dummy vertices. The macro-level of the only degree-3 vertex in this directed tree, $M - s_j - 1$, determines the scheduled starting time s_j of the task T_j. This directed tree then occupies the free side of r for a number of

macro-levels equivalent to the duration of its processing time, $\alpha_j - w_j s_j$. If all the directed trees in the group are pushed up then the task T_j is never started and its processing time is left decay to zero.

The final uppermost occupied macro-level of T determines the makespan of the schedule. If we minimize the number of dummy vertices then we are keeping this level as low as possible. The total number of vertices in T is bounded by a polynomial in the parameters of MIN-MAKESPAN and, since MIN-MAKESPAN is strongly NP-Hard, our theorem follows.

□

4.2 The Recursive Case

We now return to the recursive case of algorithm `makeTreeDAGLevelPlanar`. If $|\mathcal{V}| > 1$ then we choose some $r \in \mathcal{V}$ that maximizes $\phi(r)$. Note that the order in which we choose each r can be pre-computed. We proceed as before by populating a matrix $M(T)$. However, if any T_v is not a directed tree, we need to recursively ensure its level planarity. Let $M(T_v)$ be the final matrix associated with T_v. We populate a new column j_{T_v} of the present matrix $M(T)$ in two steps. Firstly, we visit each vertex v' along the (undirected) path from v to its earliest ancestor whose in-degree is anything other than one and set the entries as in the base case. Secondly, we populate column j_{T_v} of $M(T)$ bottom-up by setting $M(T)_{i,j_{T_v}}$ to

$$\begin{cases} \max(maxRow, maxCol) & \text{if } M(T)_{i,j_{T_v}} > 0 \wedge maxRow > 0 \\ maxCol & \text{if } M(T)_{i,j_{T_v}} > 0 \wedge maxRow = 0 \\ 0 & \text{if } M(T)_{i,j_{T_v}} = 0 \wedge maxRow = 0 \end{cases} \tag{2}$$

where $maxRow = \max_j M(T_v)_{i,j}$ and $maxCol = \max_{i'} M(T)_{i',j_{T_v}}$. This new column can be considered a substitute that is at least as restrictive as T_v on the level planarity of T.

An additional complication occurs if some T_v *smothers* r (see Fig. 6 for a typical example). In this case T_v, whose level planarity we have already ensured, has three vertices $r_{T_v}, w_{\mathrm{L}}, w_{\mathrm{R}} \in V_v$ such that r_{T_v} has in-degree greater than one, $\phi(w_{\mathrm{L}}), \phi(w_{\mathrm{R}}) \geq \phi(r)$ and the least common ancestors of the pairs w_{L}, r_{T_v} and r_{T_v}, w_{R} are on the same level. In other words, in any level planar drawing of T_v the undirected paths from r_{T_v} to w_{L} and from r_{T_v} to w_{R} 'hang' on either side of r.

In terms of $M(T)$, T_v smothers r if $M(T)$ has more than one entry in any one row whose index is less than $\phi(r_{T_v})$ and whose value is greater than or equal to $\phi(r)$. The solution is much the same as in the base case where we were required to shift the columns so that each row was left with at most two entries. We start with the last row with more than one entry whose index is less than $\phi(r_{T_v})$ and whose value is greater than or equal to $\phi(r)$. Using the same criteria as before, we fix one column (j_{F}) and apply `shiftUp`$(M(T_v), j)$ to the others.

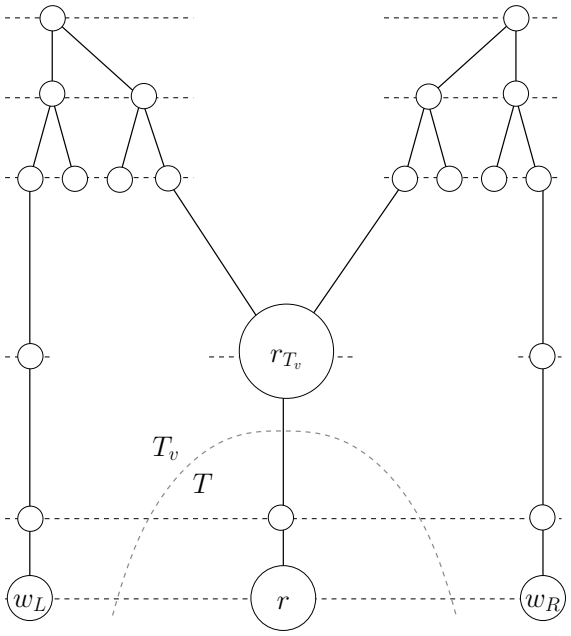

Figure 6: T_v is smothering r.

Then go to the previous row and repeat.

Algorithm 1: `makeTreeDAGLevelPlanar`

Input: $T = (V, E_T), \phi$
Output: $M(T)$

```
1  M(T) ← empty;
2  col ← 1;
3  Choose r ∈ {v ∈ V : inDeg(v) > 1} that maximizes φ(r);
4  foreach v ∈ N⁻(r) do
5      while v ≠ empty do
6          entry ← φ(v) + height(the tree in T_v rooted at v);
7          if entry ≥ φ(r) then
8              M(T)_{φ(v),col} ← entry;
9          v ← parent(v);                    /* empty if inDeg(v) ≠ 1 */
10     if T_v is not a directed tree then
11         M(T_v) ← makeTreeDAGLevelPlanar(T_v, φ);
12         Use shortest task first heuristic when applying shiftUp(M(T_v), j)
           to leave at most one (populated) entry in certain rows of M(T_v)
           (see Sect. 4.2);
13     Combine M(T_v) into column col of M(T);
14     col ← col + 1;
15 Use shortest task first heuristic when applying shiftUp(M(T), j) to leave
   at most two (populated) entries in any one row of M(T) (see Sect. 4.1);
16 return M(T);
```

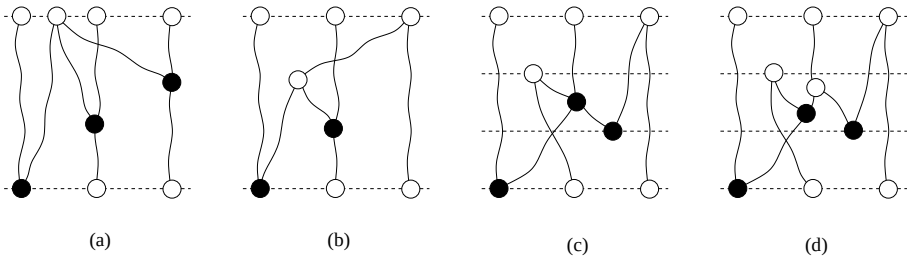

Figure 7: The four MLNP tree patterns: (a) P_1 (b) P_2 (c) P_3 (d) P_4.

4.3 Proof of Correctness

In this section we prove the correctness of our algorithm, `makeTreeDAGLevelPlanar`. Before this we introduce the concept of *level non-planar patterns*.

A *pattern* is a set of graphs that are homeomorphically equivalent. Any graph matching a *level non-planar pattern* (LNP) is itself level non-planar. Removing any edge from a graph that matches a *minimum level non-planar pattern* (MLNP) results in a graph that is level planar. [8] provided three MLNPs for *hierarchies* and proved their necessity and sufficiency. [17] refined these to cover all leveled graphs but their characterization was shown to be incomplete. [12] have completed the characterization for leveled trees. The four MLNP patterns for trees, P_1, P_2, P_3 and P_4, which are formally described in the work of [12] are shown in Fig. 7. A tree T with leveling ϕ is level non-planar if and only if T has a subtree matching one of the MLNPs P_1, P_2, P_3, or P_4. We use this result as the basis for our proof of correctness.

Theorem 5. *Given a tree DAG T with leveling ϕ, the algorithm* `makeTreeDAGLevelPlanar`(T, ϕ) *inserts dummy vertices to ensure level planarity.*

Proof: We show that every subtree of T matching one of the MLNPs P_1, P_2, P_3, or P_4 will have dummy vertices inserted so that it no longers matches any MLNP. P_1 has three vertices of in-degree greater than one. `makeTreeDAGLevelPlanar` will recurse to the tree dag whose root has the lowest $\phi(r)$. This tree dag is level planar and will be combined into a single column and added to the tree dag whose root has the next lowest $\phi(r)$. This causes smothering and requires the insertion of dummy vertices (see Fig. 8(a)) to ensure level planarity. Finally, this will be combined into a single column and added to the tree dag whose root has the next lowest $\phi(r)$. We have now traversed the entire pattern P_1 but the insertion of the dummy vertices has made it level planar. Analogous arguments handle subtrees of T that match P_2, P_3, or P_4 (see Fig. 8(b) - (d)). □

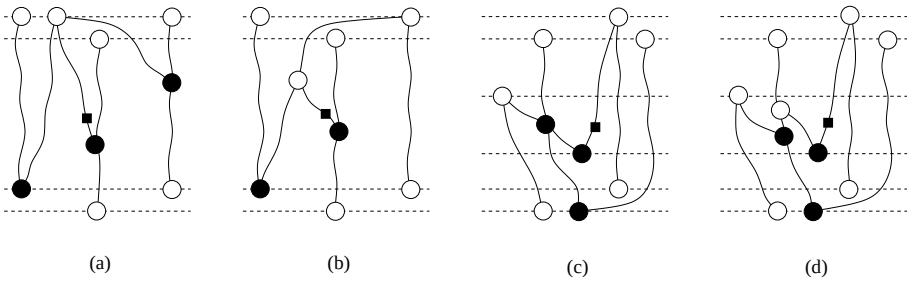

Figure 8: The four MLNP tree patterns with dummy vertices inserted: (a) P_1 (b) P_2 (c) P_3 (d) P_4.

4.4 Implementation

By storing $M(T)$ in sparse form, `makeTreeDAGLevelPlanar` can be modified to run in $\mathcal{O}(|V|)$ time. For each column, we list an *edge identifier*, an *offset*, the index of the first row where the column has a populated entry, the number of *pairs* of entries and the entries themselves (see Fig. 9). The edge identifier is the edge into which we insert dummy vertices when applying $\texttt{shiftUp}(M(T), j)$. The offset is used by $\texttt{shiftUp}(M(T), j)$ to update all entries in the column in constant time. When a column is shifted up the offset is first decremented. The entries are encoded as pairs of values where the first is the number of times the second value appears consecutively, *e.g.* the entries $8, 7, 7, 5, 5$ are encoded as $1, 8, 2, 7, 2, 5$. If a column with those values is shifted up for the first time we set its offset to -1 and, if the second value in the last pair plus the offset is less than $\phi(r)$, we remove the last pair from the entries and decrement the number of distinct entries. The sparse form allows the $\texttt{first}(M(T), j)$, $\texttt{last}(M(T), j)$ and $\texttt{shiftUp}(M(T), j)$ operations to be performed in $\mathcal{O}(1)$ time.

The shortest task first heuristics (Lines **12** and **15** of Algorithm 1) can easily be implemented using two nested `for` loops. However, we need only visit the populated entries in each matrix. Using the sparse form of $M(T)$ in Line **12** we can sort the columns in descending order by $\texttt{last}(M(T_v), j)$ using bucket sort (since the range of values is the range of levels occupied by the graph and must be less than or equal to the number of vertices) and then process each bucket from left to right as follows. Choose one of the columns with the greatest $\texttt{first}(M(T_v), j)$ to be j_{F}, apply $\texttt{shiftUp}(M(T_v), j)$ to the other columns and add them to the next bucket. A similar approach can be used in Line **15**.

4.5 A Worked Example

In this section we illustrate the workings of `makeTreeDAGLevelPlanar` with an example. Consider the tree DAG T in Fig. 10. It has two vertices of in-degree greater than one, namely r_1 and r_2. The algorithm visits r_1 but finds that one of its neighbors in $\mathcal{N}^-(r_1)$ is a tree DAG and needs to recursively ensure its level planarity first. After traversing the (undirected) paths from each $v \in \mathcal{N}^-(r_2)$

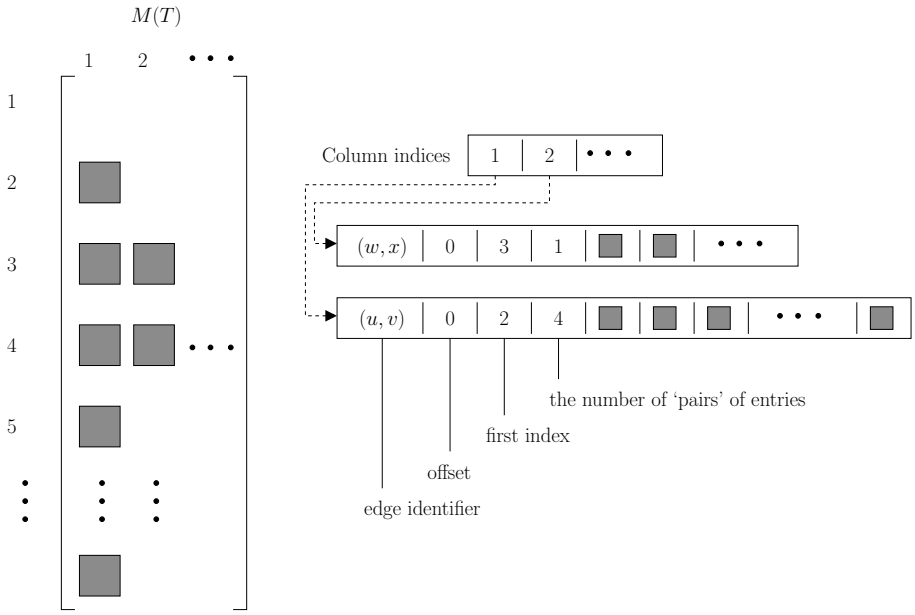

Figure 9: $M(T)$ can be stored in a sparse form.

to the roots of their respective trees, we get a matrix for the sub-tree DAG T' (the graph induced by the shaded vertices in Fig. 10):

$$M(T') = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} (v_1, r_2) & (v_2, r_2) & (v_3, r_2) & (v_4, r_2) \end{array} \\ \begin{bmatrix} & 6 & & \\ & 6 & 6 & 7 \\ 7 & 5 & & \\ 7 & & & \\ & & & \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \end{array}.$$

By Lemma 1, T' is not level planar. We fix the second and third columns and perform $\texttt{shiftUp}(M(T'), 4)$ to get the following matrix:

$$M(T') = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} (v_1, r_2) & (v_2, r_2) & (v_3, r_2) & (v_4, r_2) \end{array} \\ \begin{bmatrix} & 6 & & 6 \\ & 6 & 6 & \\ 7 & 5 & & \\ 7 & & & \\ & & & \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \end{array}.$$

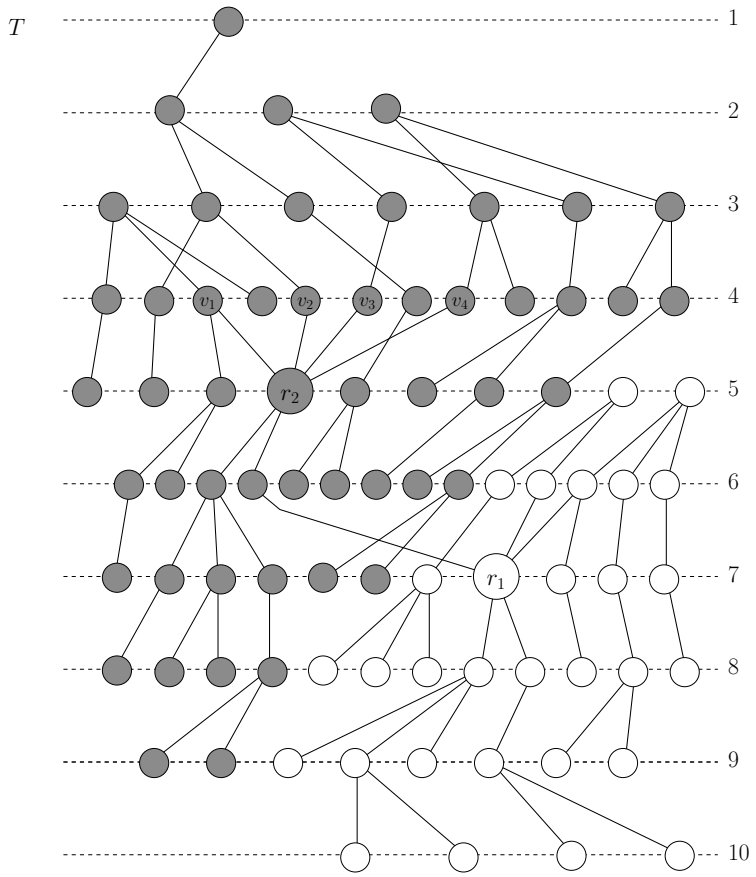

Figure 10: T has two vertices with in-degree greater than one, namely r_1 and r_2. We focus on the shaded sub-tree DAG rooted at r_1 first.

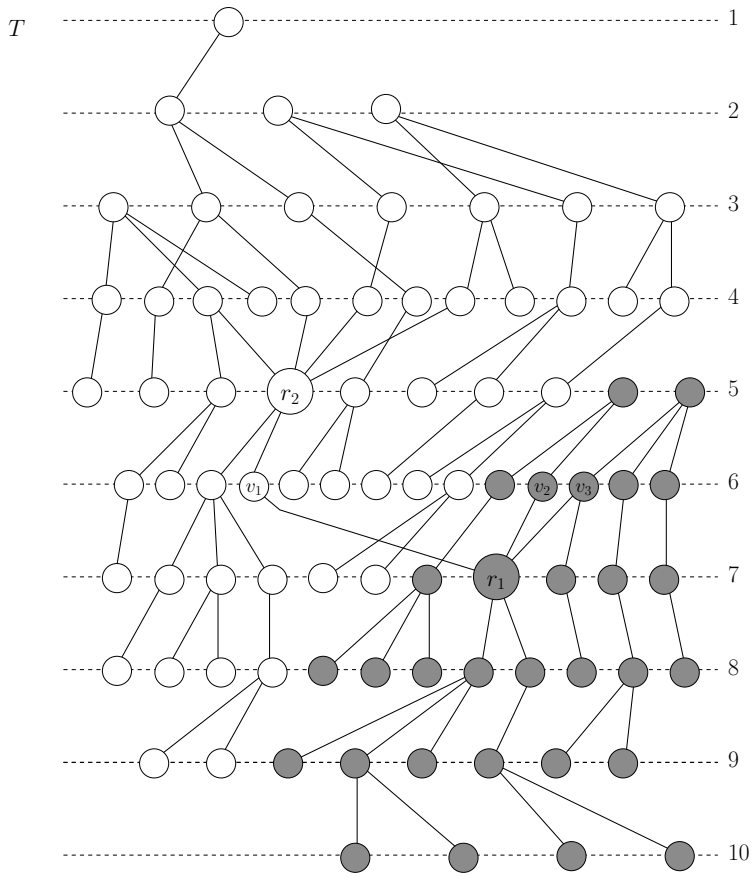

Figure 11: We focus on the entire tree DAG rooted at r_2.

This equates to inserting a dummy vertex along the edge (v_4, r_2). T' is now level planar and we can turn to the level planarity of the entire tree T (see Fig. 11). After traversing the (undirected) paths from each $v \in \mathcal{N}^-(r_1)$ to the roots of their respective trees and combining $M(T')$ into the first column using Eqn. 2, we get a matrix for T:

$$M(T) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccc} (v_1, r_1) & (v_2, r_1) & (v_3, r_1) \end{array} \\ \left[\begin{array}{ccc} 9 & & \\ 9 & & \\ 9 & & \\ 9 & & \\ 9 & 8 & 9 \\ & & 8 \\ \vdots & \vdots & \vdots \end{array}\right] \end{array}.$$

By Lemma 1, T is not level planar. We fix the second and third columns and perform $\texttt{shiftUp}(M(T), 1)$ to get the following matrix:

$$M(T) = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccc} (v_1, r_1) & (v_2, r_1) & (v_3, r_1) \end{array} \\ \left[\begin{array}{ccc} 8 & & \\ 8 & & \\ 8 & & \\ 8 & & \\ 8 & & \\ & 8 & 9 \\ & & 8 \\ \vdots & \vdots & \vdots \end{array}\right] \end{array}.$$

This equates to inserting a dummy vertex along the edge (v_1, r_1). T is now level planar and we are done.

4.6 Violating the Leveling

While ensuring the level planarity of T, we may be violating the leveling with respect to G (see the edge (u, v) in Fig. 1(b)). These upwardly directed edges can be deceptive when trying to understand the graph visually. However, we can insert compensating dummy vertices whenever the leveling with respect to G is violated.

We previously defined a leveling of G to be a surjective mapping $\phi : V \to \{1, 2, \ldots, k\}$ where $\phi(v)$ specifies the level of each vertex v and, implicitly, the number of dummy vertices along each edge to make it proper. Alternatively, we can define a leveling of G to be a mapping $\psi : E \to \mathbb{Z}^+$ where $\psi(e)$ specifies the number of dummy vertices along each edge e to make it proper and, implicitly, the level of each vertex.

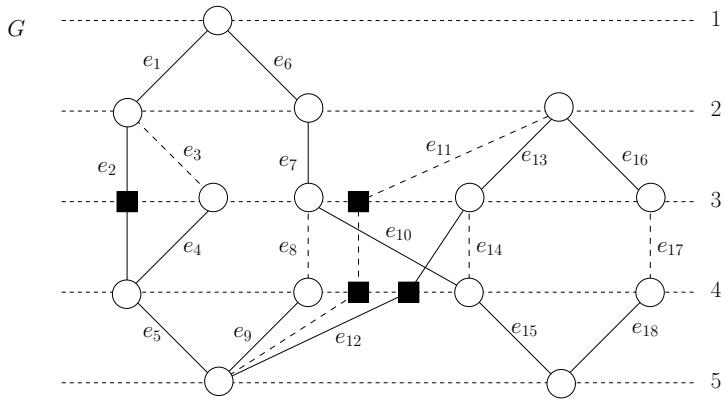

Figure 12: Every edge in $E \setminus E_T$ (the dashed edges) completes a fundamental cycle.

At the same time, every edge in $E \setminus E_T$ completes an (undirected) *fundamental cycle* C in T (see Fig. 12). By traversing each C in some arbitrary direction, we get the cycle vector $\chi(C)$ (with coordinates $\chi(C)_e, \forall e \in E$) defined by

$$\chi(C)_e = \begin{cases} 1 & \text{if e is directed with the traversal of C,} \\ -1 & \text{if e is directed against the traversal of C,} \\ 0 & \text{if e is not in C.} \end{cases} \tag{3}$$

We say that a cycle is *balanced* if $\sum \chi(C) = 0$. The set of vectors corresponding to a set of fundamental cycles $\mathcal{C}$ constitute a basis for the cycle vector space of G.

Now, suppose we partition the edge set $E = \bigcup S = s_1 \cup \ldots \cup s_t$ such that each subset s_j, $1 \leq j \leq t$, is the maximal set of edges shared between the same subset of cycles in $\mathcal{C}$ (see Fig. 13). In other words, two edges belong to the same subset if they are both bridges or both belong to the same S-vertex of an SPQR-tree [9] of the same biconnected component of the underlying undirected graph of G. Then, if ψ is some leveling of G, any dummy vertex along an edge $e \in s_j$, $1 \leq j \leq t$, can be moved to any other similarly directed edge (with respect to any cycle) in s_j. In fact, we can specify an equivalence class of levelings of G using two mappings $\psi_+ : S \to \mathbb{Z}^+$ and $\psi_- : S \to \mathbb{Z}^+$ where ψ_+ and ψ_- specify the number of dummy vertices along the edges in either direction in each subset s_j. To make sure that the directions are consistent, we use a *fundamental cycle-edge subset incidence matrix* $\mathcal{F}$ defined by

$$\mathcal{F}_{C,s} = \begin{cases} 1 & \text{if the directions of the edges in s} \\ & \text{are consistent with the traversal of C,} \\ -1 & \text{if the directions of the edges in s} \\ & \text{are inconsistent with the traversal of C,} \\ 0 & \text{if s is not in C.} \end{cases} \tag{4}$$

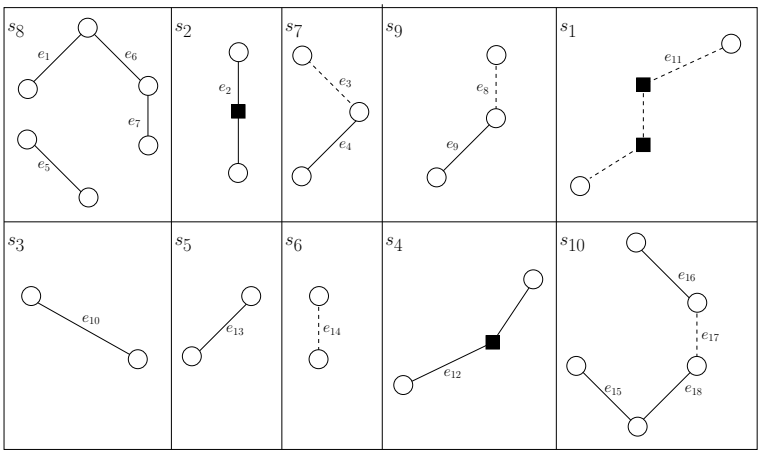

Figure 13: The partitioning of the edge set in Fig. 12.

For example, a fundamental cycle-edge subset incidence matrix for the DAG in Fig. 12 is

$$\mathcal{F} = \begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{array} \begin{array}{c} \begin{array}{cccccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \end{array} \\ \left[\begin{array}{cccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}\right] \end{array}.$$

We define the vectors $\vec{w}$ (with coordinates $\vec{w}_s, \forall s \in S$, in the order of the columns of $\mathcal{F}$) by $\vec{w}_s = \psi_+(s) - \psi_-(s)$ and $\vec{b}$ (with coordinates $\vec{b}_C, \forall C \in \mathcal{C}$, in the order of the rows of $\mathcal{F}$) by $\vec{b}_C = \sum \chi(C)$. Now, if ψ_+ and ψ_- specify a leveling of G then

$$\mathcal{F}\vec{w} = \vec{b}, \tag{5}$$

i.e. the dummy vertices balance the fundamental cycles. This formulation has been used by [14] to find a leveling of a DAG G with the minimum number of dummy vertices.

So if we start with an initial leveling of G and add dummy vertices to ensure level planarity with respect to T, we can prevent any violation of the leveling by examining the null-space of $\mathcal{F}$. For example, if the edge e_{13} in s_5 needs an additional dummy vertex, this can be compensated by adding a dummy vertex to e_{11} in s_1 and a dummy vertex to one of e_{16}, e_{17} or e_{18} in s_{10} since the columns corresponding to s_1, s_5 and s_{10} in $\mathcal{F}$ are linearly dependent.

5 Crossing Minimization

An embedding algorithm is implied by `makeTreeDAGLevelPlanar`. Fixing columns (j_{L}, j_{R} and j_{F}) determines the relative order of their corresponding vertices on their respective levels. However, we have some freedom when positioning the remaining vertices. We use this freedom to minimize the number of edge crossings involving edges that are not part of the significant spanning tree E_T.

We use an order induced by a Fiedler vector $\vec{x}$ of $\mathcal{L}(G)$: if the relative order of two vertices u, v on the same level has not been decided and $\vec{x}_u < \vec{x}_v$ then we set u to the left of v. This heuristic has previously been used to determine exact x-coordinates when drawing a directed graph [6] and for the 2-level crossing minimization problem [28]. It is based on a result of [30] that shows a strong relation between the *minimum linear arrangement (MLA) problem* of a bipartite graph, the bipartite crossing number and an algorithm for finding an approximate solution to the MLA problem [23]: Given an undirected graph $G = (V, E)$, the MLA problem is to determine a bijection $f : V \to \{1 \dots |V|\}$ such that $\sum_{(u,v)\in E} |f(u) - f(v)|$ is minimized. [23] use a Fiedler vector $\vec{x}$ of $\mathcal{L}(G)$ to induce an order (if $\vec{x}_u < \vec{x}_v$ then $f(u) < f(v)$) which is unique up to the relative order of repeated eigenvector elements. [30] show that in most cases, if we have an approximate solution to the MLA problem for a bipartite graph $G = (V_1 \cup V_2, E)$, we can place the vertices of V_1 and V_2 on two levels in the order obtained from f to obtain a good approximation for the 2-level crossing minimization problem. Empirical evidence [28] suggests that this heuristic efficiently provides good solutions for the 2-level case and [6] have used it with much success over an arbitrary number of levels. It is worth noting that this order is determined globally, *i.e.* for all vertices on all levels at the same time.

6 Bringing It All Together

We now outline how the approach from the preceding sections can be used to draw an entire directed graph. The dependency relationship between packages in large software systems tend to have a specific structure. Packages depend on base packages with common functionality that is shared with many other packages and on certain application-specific packages. Supposing this graph has a significant spanning tree, *e.g.* the tree that propagates the most centrality, we can draw it so that this spanning tree is emphasized.

Figure 14 depicts the largest connected component of the dependency graph for packages of the Python programming language[1] after removing any redundant dependencies by computing its transitive reduction. The significant spanning tree (solid edges) was computed as in Sect. 3 and the orders of the vertices on each level were determined by the embedding algorithm implied by `makeTreeDAGLevelPlanar` along with the crossing minimization heuristic. The computation of the final x-coordinates was based on the work of [4] and the

[1] http://www.python.org/

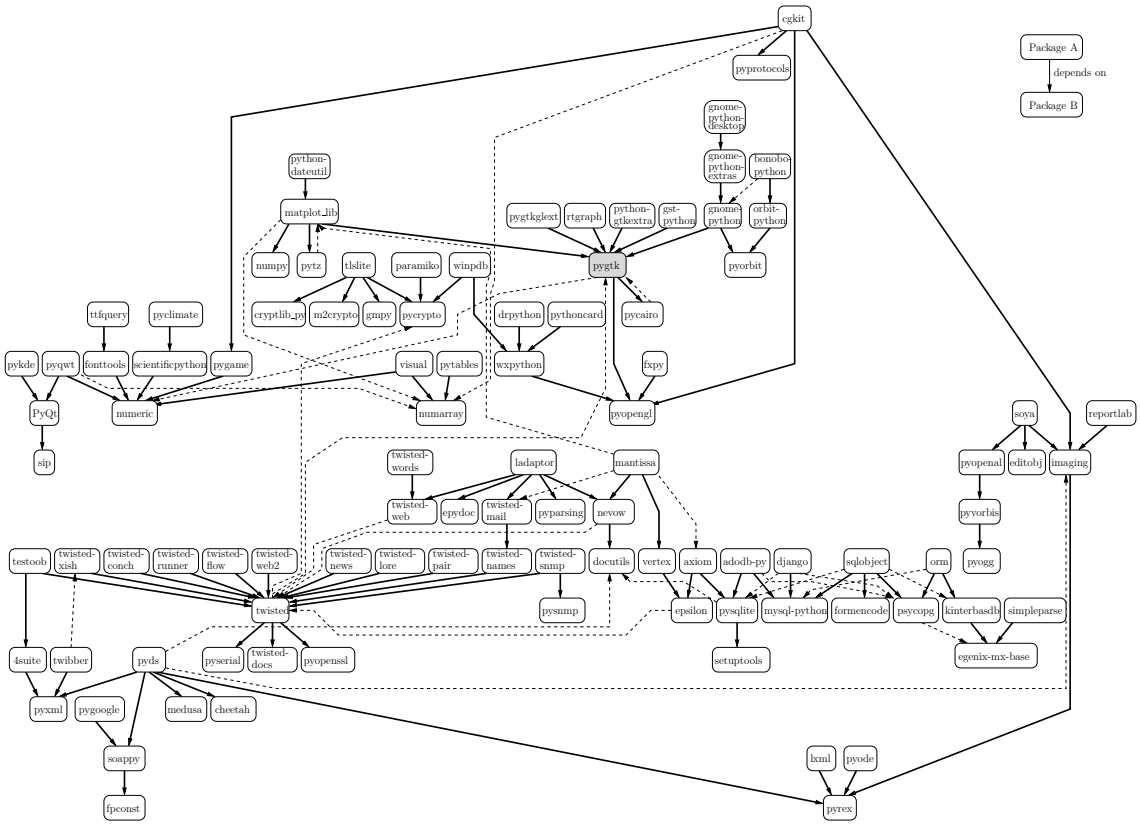

Figure 14: The dependency graph for Python packages.

less significant edges (dashed edges) were added in by hand. The technique of Sect. 4.6 has not been applied.

7 Conclusion

We have presented a method of drawing a directed graph that emphasizes a significant spanning tree. The spanning tree is a tree DAG with multiple sources and so it is more appropriate to give each of these vertices root status than to nominate any single vertex and hang the subtrees from it. We ensure the level planarity of the spanning tree by inserting dummy vertices and restrict the possible level embeddings so that no two significant edges cross. These steps can be performed in linear time using a sparse form of matrix storage. We also show that finding a minimum number of dummy vertices to ensure level planarity even when the tree DAG has exactly one vertex of in-degree greater than one is NP-Hard.

The remaining freedom in the order of the vertices on their respective levels

is used to reduce the number of edge crossings between the remaining edges. We avoid the level-by-level sweep found in most implementations of the Sugiyama framework by using a globally oriented Fiedler vector as a multi-level crossing minimization heuristic. Upwardly directed edges in the final drawing can be fixed using compensating dummy vertices, provided the edges are not part of a directed cycle.

8 Acknowledgments

The authors are grateful to the referees for their helpful comments.

References

[1] B. Alidaee and N. Womer. Scheduling with Time Dependent Processing Times: Review and Extensions. *Journal of the Operational Research Society*, 50(7):711–720, 1999.

[2] P. Bonacich. Factoring and Weighting Approaches to Status Scores and Clique Identification. *Journal of Mathematical Sociology*, 2:113–120, 1972.

[3] R. Botafogo, E. Rivlin, and B. Schneiderman. Structural Analysis of Hypertexts: Identifying Hierarchies and Useful Metrics. *ACM Transactions on Information Systems*, 10(2):142–180, 1992.

[4] U. Brandes and B. Köpf. Fast and Simple Horizontal Coordinate Assignment. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proceedings of the 9^{th} International Symposium on Graph Drawing (GD'01)*, pages 31–44. Springer, 2002.

[5] C. Buchheim, M. Jünger, and S. Leipert. Drawing Rooted Trees in Linear Time. *Software – Practice and Experience*, 36(6):651–665, 2006.

[6] L. Carmel, D. Harel, and Y. Koren. Combining Hierarchy and Energy for Drawing Directed Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):46–57, 2004.

[7] T. Cheng and Q. Ding. The Complexity of Scheduling Starting Time Dependent Tasks with Release Times. *Information Processings Letters*, 65(2):75–79, 1998.

[8] G. Di Battista and E. Nardelli. Hierarchies and Planarity Theory. *IEEE Transactions on Systems, Man and Cybernetics*, 18(6):1035–1046, 1988.

[9] G. Di Battista and R. Tamassia. On-Line Maintenance of Triconnected Components with SPQR-Trees. *Algorithmica*, 15(4):302–318, 1996.

[10] P. Eades and S. Whitesides. Drawing Graphs in Two Layers. *Theoretical Computer Science*, 131(2):361–374, 1994.

[11] P. Eades and N. Wormald. Edge Crossings in Drawings of Bipartite Graphs. *Algorithmica*, 11(4):379–403, 1994.

[12] J. Fowler and S. Kobourov. Minimum Level Nonplanar Patterns for Trees. In S. Hong, T. Nishizeki, and W. Quan, editors, *Proceedings of the 15^{th} International Symposium on Graph Drawing (GD'07) (to appear)*. Springer, 2008.

[13] D. Harel and Y. Koren. A Fast Multi-Scale Method for Drawing Large Graphs. *Journal of Graph Algorithms and Applications*, 6(3):179–202, 2002.

[14] M. Harrigan and P. Healy. On Layering Directed Acyclic Graphs. In M. Jünger, S. Kobourov, and P. Mutzel, editors, *Graph Drawing*, volume 05191 of *Dagstuhl Seminar Proceedings*, Germany, 2005.

[15] M. Harrigan and P. Healy. Practical Level Planarity Testing and Layout with Embedding Constraints. In S. Hong, T. Nishizeki, and W. Quan, editors, *Proceedings of the 15th International Symposium on Graph Drawing (GD'07) (to appear)*. Springer, 2008.

[16] P. Healy and A. Kuusik. Algorithms for Multi-Level Graph Planarity Testing and Layout. *Theoretical Computer Science*, 320(2–3):331–344, 2004.

[17] P. Healy, A. Kuusik, and S. Leipert. A Characterization of Level Planar Graphs. *Discrete Mathematics*, 280(1-3):51–63, 2004.

[18] L. Heath and S. Pemmaraju. Stack and Queue Layouts of Directed Acyclic Graphs: Part II. *SIAM Journal on Computing*, 28(5):1588–1626, 1999.

[19] I. Herman, G. Melançon, and M. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[20] H. Hotelling. Analysis of a Complex of Statistical Variables into Principal Components. *Journal of Educational Psychology*, 24:417–441, 1933.

[21] M. Jünger, S. Leipert, and P. Mutzel. Level Planarity Testing in Linear Time. In S. Whitesides, editor, *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, pages 224–237. Springer, 1999.

[22] D. Jungnickel. *Graphs, Networks and Algorithms*. Algorithms and Computation in Mathematics. Springer, 2$^{\text{nd}}$ edition, 1999.

[23] M. Juvan and B. Mohar. Optimal Linear Labelings and Eigenvalues of Graphs. *Discrete Applied Mathematics*, 36(2):153–168, 1992.

[24] D. Koschützki, K. Lehmann, L. Peeters, S. Richter, D. Tenfelde-Podehl, and O. Zlotowski. Centrality Indices. In U. Brandes and T. Erlebach, editors, *Network Analysis*, pages 16–61. Springer, 2005.

[25] K. Lehmann and S. Kottler. Visualizing Large and Clustered Networks. In M. Kaufmann and D. Wagner, editors, *Proceedings of the 14th International Symposium on Graph Drawing (GD'06)*, pages 240–251. Springer, 2007.

[26] T. Munzner. H3: Laying Out Large Directed Graphs in 3D. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis'97)*, pages 2–10. IEEE Computer Society, 1997.

[27] T. Munzner. Drawing Large Graphs with H3Viewer and Site Manager. In S. Whitesides, editor, *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, pages 384–393. Springer, 1998.

[28] M. Newton, O. Sýkora, and I. Vrto. Two New Heuristics for Two-Sided Bipartite Graph Drawing. In S. Kobourov and M. Goodrich, editors, *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, pages 312–319. Springer, 2003.

[29] E. Reingold and J. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[30] F. Shahrokhi, O. Sýkora, L. Székely, and I. Vrťo. On Bipartite Drawings and the Linear Arrangement Problem. *SIAM Journal on Computing*, 30(6):1773–1789, 2001.

[31] K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[32] J. Walker. A Node-Positioning Algorithm for General Trees. *Software – Practice and Experience*, 20(7):685–705, 1990.

[33] G. Wills. NicheWorks – Interactive Visualization of Very Large Graphs. *Journal of Computational and Graphical Statistics*, 8(2):190–212, 1999.