Efficiently Drawing a Significant Spanning Tree of a Directed Graph

Martin Harrigan* Patrick Healy†

Department of Computer Science and Information Systems, University of Limerick, Ireland.

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 [18, 19, 25, 13]. In the directed case the spanning tree is a tree DAG (a directed graph without any undirected cycles) and not simply a directed tree with one appropriate root. It may have multiple sources (vertices with indegree equal to zero) 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 can be considered a variation of the Sugiyama framework [23] in that it combines two steps of the framework (leveling and crossing minimisation) 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 the globally oriented Fiedler vector we choose permutations of the vertices on each level that reduce the number of crossings between the remaining edges.

Keywords: graph drawing, significant spanning tree, level planarity, Fiedler vector.

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 [2], draw the spanning tree using the simpler tree drawing algorithms [21, 24, 4] 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 any one of these vertices 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 other graph-theoretic knowledge. We find a leveling of the graph that is level planar with respect to T 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 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 crossings between the remaining edges.

*e-mail: martin.harrigan@ul.ie

†e-mail: patrick.healy@ul.ie

Asia-Pacific Symposium on Visualisation 2007
5 - 7 February, Sydney, NSW, Australia
1-4244-0809-1/07/$20.00 © 2007 IEEE

In addition, we use appropriate visual cues, for example, drawing the less significant edges in a lighter colour, to make the structure more readily discernable.

This can be considered a variation of the Sugiyama framework [23] for drawing directed graphs. 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 maximise the size of a level planar subgraph. Even if there are only two levels and the vertices on one are fixed, both the crossing minimisation and maximum level planar subgraph problems are $\mathcal{NP}$-complete [8, 7] and so heuristics are generally employed. Once we have computed a Fiedler vector, our method efficiently combines the leveling and crossing minimisation steps in linear time.

This paper is organised 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. 4.4 we describe a crossing minimisation heuristic based on a Fiedler vector and show how it can be combined with the previous step. Section 5 applies the method to a directed graph modeling the dependency relationship between packages related to the Python programming language and Sect. 6 gives some concluding remarks and directions for future research.

2 PRELIMINARIES

A *leveling* of a DAG $G = (V,E)$ is a 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. 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 proper leveling ϕ consists of total orderings $<_j$ 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 proper leveling ϕ. 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).

The *Laplacian* $\mathscr{L}(G)$ is defined by

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

$\mathscr{L}(G)$ 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 on the matrix $\lambda_n I - \mathscr{L}(G)$ or more efficiently using multi-scale methods [9].

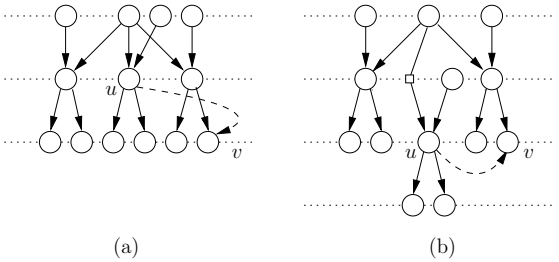

Figure 1: A non-level planar and level planar drawing of spanning tree DAGs (solid edges).

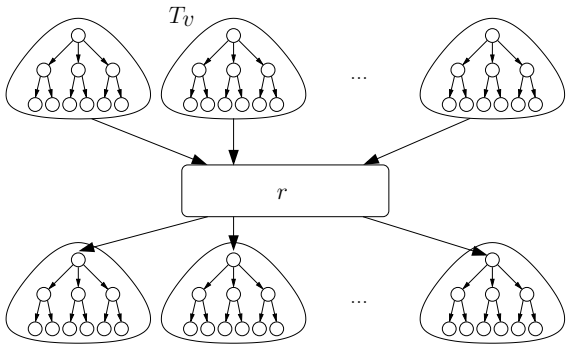

Figure 2: The Base Case.

3 Choosing a Significant Spanning Tree DAG

Let $G = (V,E)$ be a connected directed graph. We seek a significant spanning tree DAG $T = (V,E_T)$ of G, i.e. one in which the edges in E_T are more significant than those in $E \setminus E_T$. This may be done using domain-specific knowledge or using any of the numerous methods to measure significance (see e.g., spanning tree algorithms in [15] and structural indices in [17]). The method we choose is to weight each edge $(u,v) \in E$ by $(\mathbf{x_u} - \mathbf{x_v})^2$ where $\mathbf{x}$ is a Fiedler vector of $\mathscr{L}(G)$. T is then a minimum weighted spanning tree DAG of G. This will preserve intra-cluster edges at the expense of inter-cluster edges.

Our choice of T is primarily due to the fact that we will be reusing $\mathbf{x}$ later. However, we give two reasons to justify T as being a 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 $\mathbf{x'_v}$ then we want to minimize $\sum_{(u,v)\in E}(\mathbf{x'_u} - \mathbf{x'_v})^2$ subject to $\mathbf{x'x'^T} = \mathbf{1}$ (to avoid the trivial solution of setting $\mathbf{x'} = \mathbf{0}$). It turns out that a solution is precisely $\mathbf{x}$. Secondly, $\mathbf{x_u}$ can be interpreted as a measure of u's 'degree-normalized' eigenvector centrality [1] and so E_T are the edges that propagate most of this centrality through G.

4 Ensuring Level Planarity

Level planarity can be tested in $\mathscr{O}(|V|)$ time using the PQ-tree data structure [12, 14] or in $\mathscr{O}(|V|^2)$ time using the simpler Vertex-Exchange Graph [10]. 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.

The algorithm `makeTreeDAGLevelPlanar` is based on the recursive nature of a tree DAG. It processes a tree DAG T with proper leveling ϕ by recursively decomposing T into smaller tree DAGs with fewer vertices of indegree 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. We apply this algorithm twice, on T and on its reverse graph (replacing every $(u,v) \in E$ with (v,u)) after inserting the dummy vertices from the first pass.

4.0.1 The Base Case.

Let $T = (V,E_T)$ be a tree DAG with proper leveling ϕ and $\mathscr{V} = \{v \in V : \texttt{indegree}(v) > 1\}$. We assume $|\mathscr{V}| > 0$ since otherwise T with ϕ is trivially level planar. In the base case $|\mathscr{V}| = 1$ and we let this vertex be r. The vertices in the outgoing and incoming neighbours of v, $\mathscr{N}^+(r)$ and $\mathscr{N}^-(r)$, 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 maximally connected component that is joined to r through some $v \in \mathscr{N}^+(r) \cup \mathscr{N}^-(r)$. To ensure level planarity, we need only insert dummy vertices along edges joining $\mathscr{N}^-(r)$ to r.

We proceed by populating a matrix $M(T)$. For each $v \in \mathscr{N}^-(r)$ 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 descendent 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 unpopulated. We note the edge (v,r) that corresponds to each column.

$M(T)$ is a $k \times |\mathscr{N}^-(r)|$ matrix (where k is the number of levels) consisting of partially populated columns of consecutive non-decreasing entries. 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 row or 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.

Lemma 4.1 *If T is a tree DAG with leveling ϕ and has one vertex of indegree 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 $\{v_i\} \subseteq \mathscr{N}^-(r)$ 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 $v'_i, w_i \in V_{v_i}$, $\phi(v'_i) = j$ and $\phi(w_i) \geq \phi(r)$, $1 \leq i \leq 3$. For any total ordering $<_j$ of the vertices in $\phi^{-1}(j)$ there must be at least one crossing in any level drawing of T (see Fig. 3). If there are at most two entries in any one row, the directed trees can 'hang' over either side of r. ∎

To insert a dummy vertex along an edge between some $v \in \mathscr{N}^-(r)$ 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 $\texttt{shiftUp}(M(T),j)$ where j is the index of the column to shift.

We wish to apply $\texttt{shiftUp}(M(T),j)$ a small number of times so that each row contains at most two entries. We use the following heuristic. Find the last row with more than two entries. Leave two columns whose last entries occur the latest fixed (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 $\texttt{shiftUp}(M(T),j)$ to all other columns that have an entry in this row. Go to the previous row and repeat.

The problem 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

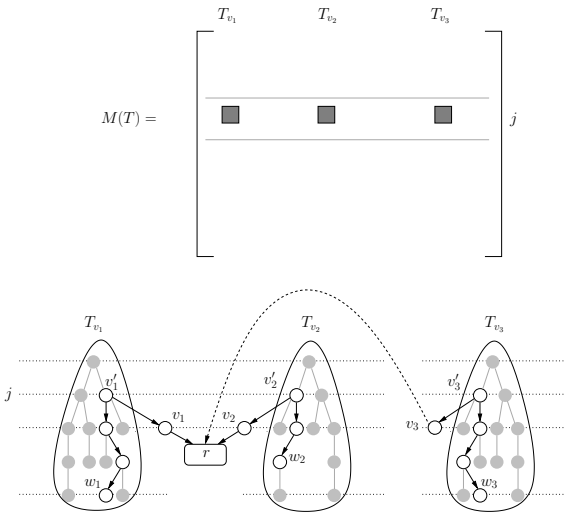

Figure 3: Row j of $M(T)$ has more than two entries and so T with leveling ϕ is not level planar.

j_R. There are at most $|\mathscr{N}^-(r)|$ independent tasks. The tasks arrive 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 j 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 minimise 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.

4.0.2 The Recursive Case.

If $|\mathscr{V}| > 1$ we choose some $r \in \mathscr{V}$ that maximises $\phi(r)$. 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 indegree is anything other than one and set the entries as in the base case. Secondly, we combine $M(T_v)$ into the same column by setting $M(T)_{i,j_{T_v}} = \max(M(T)_{i,j_{T_v}}, \max_j M(T_v)_{i,j})$. This column can be thought of as representing a directed tree 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. 4). In this case T_v, whose level planarity we have already ensured, has three vertices $r_{T_v}, w_L, w_R \in V_v$ such that r_{T_v} has indegree greater than one, $\phi(w_L), \phi(w_R) \geq \phi(r)$ 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 has 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 leave one column fixed (j_F) and apply $\texttt{shiftUp}(M(T_v), j)$ to the others. Go to the previous row and repeat.

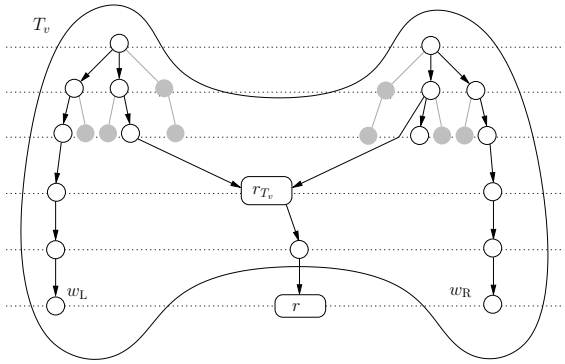

Figure 4: T_v smothers r.

Algorithm 1: `makeTreeDAGLevelPlanar`

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

```
1  M(T) ← empty;
2  col ← 1;
3  Choose r ∈ {v ∈ V : indegree(v) > 1} that maximises 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 indegree(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. 4.0.2);
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. 4.0.1);
16 return M(T);
```

4.1 Proof of Correctness

Theorem 4.2 *Given a tree DAG T with proper leveling ϕ,* $\texttt{makeTreeDAGLevelPlanar}(\texttt{T}, \phi)$ *inserts dummy vertices to ensure level planarity.*

Proof (based on a characterisation of level planar graphs [11]) T is level planar if and only if it does not contain any minimal level non-planar (MLNP) subgraph pattern. Since T is a tree DAG we need to show that it does not contain the MLNP 'tree pattern'. Briefly, let i and j be the extreme levels of the pattern and r be a vertex with three incident subtrees that have the following properties:

- Each subtree has at least one vertex on both extreme levels.
- Each subtree is either an undirected path or two branches which are undirected paths.
- All leaf vertices are on the extreme levels and each subtree has at most one leaf vertex on each extreme level.

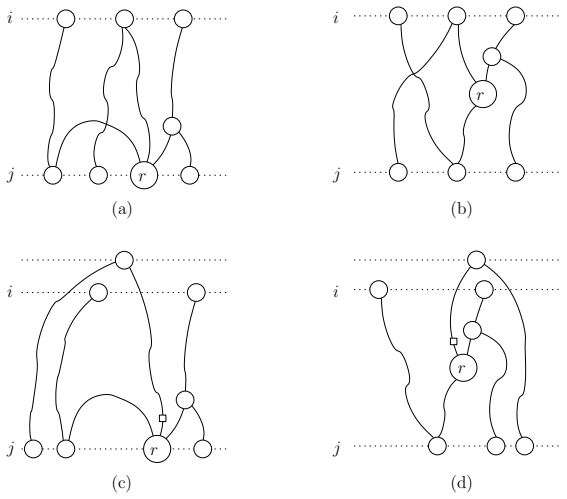

Figure 5: Two typical examples of the MNLP tree pattern ((a) and (b)) and the dummy vertices added to each to ensure level planarity ((c) and (d) respectively).

- Those subtrees which are undirected paths have one or more non-leaf vertices on the extreme level opposite to the level of their leaf vertices.

There are two cases:

1. $\phi(r) = i$ or $\phi(r) = j$ (see Fig. 5(a)): At least one of the subtrees is an undirected path starting at r, going to the opposite extreme level of r and finishing on r's level.

2. $i < \phi(r) < j$ (see Fig. 5(b)): At least one of the subtrees is an undirected path starting at r, going to the extreme level i and finishing on j and at least one of the subtrees is an undirected path starting at r, going to the extreme level j and finishing on i.

Both cases are handled by `makeTreeDAGLevelPlanar`. In Case 1, all three edges incident with r are directed similarly and so dummy vertices are inserted along one of these edges to make it level planar (see Fig. 5(c)). In Case 2, either all three edges incident with r are directed similarly, in which case it is handled as in Case 1, or two edges incident with r are directed similarly and the third is directed oppositively. This is handled by a recursive call within `makeTreeDAGLevelPlanar` (see Fig. 5(d)). The symmetrical cases to those shown in Fig. 5 (swapping i with j) are handled by the second pass of `makeTreeDAGLevelPlanar`. ∎

4.2 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 identifer*, 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. 6). 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

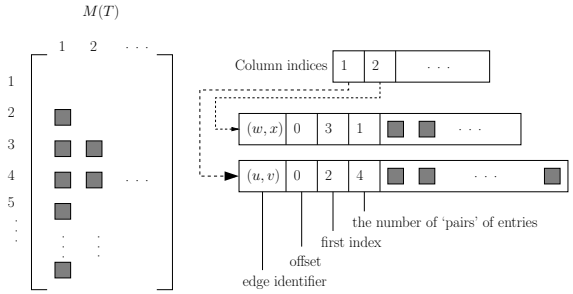

Figure 6: Storing $M(T)$ in sparse form.

number of distinct entries. The sparse form allows the initialisation of $M(T)$, $\texttt{first}(M(T), j)$, $\texttt{last}(M(T), j)$ 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 ordering according to $\texttt{last}(M(T_v), j)$ 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**.

4.3 Violating the leveling with respect to G

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)). 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 mapping $\phi : V \to \{1,2,\ldots,k\}$ where $\phi(v)$ specifies the level of each vertex v and, implicity, 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, implicity, the level of each vertex.

Every edge in $E \setminus E_T$ completes an (undirected) *fundamental cycle* C in T (see Fig. 7). 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

$$\chi(C)_e = \begin{cases} 1 & \text{if e is directed with the traversal of C,} \\ -1 & \text{if e is directed against the traversal of C,} \\ 0 & \text{if e is not in C.} \end{cases} \tag{2}$$

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

Now suppose we partition the edge set $E = \bigcup S = s_1 \cup \ldots \cup s_k$ such that each subset s_j, $1 \le j \le k$, is the maximal set of edges shared between the same subset of cycles in $\mathscr{C}$ (see Fig. 8). In other words, two edges belong to the same subset if they are both bridges or belong to the same S-vertex of an SPQR-tree [6] 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 $e \in s_j$, $1 \le j \le k$, 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 $\psi_+ : S \to \mathbb{Z}^+$

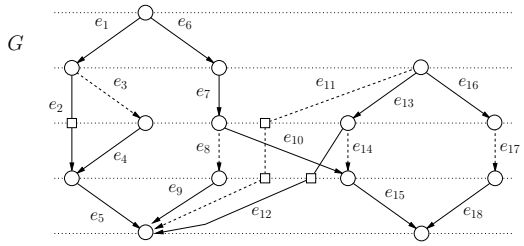

Figure 7: Each edge in $E \setminus E_T$ (the dashed edges) completes a fundamental cycle.

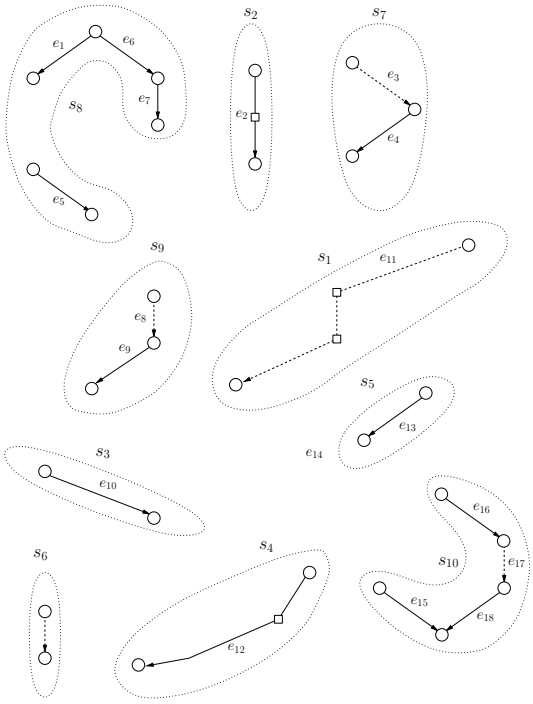

Figure 8: The partitioning of the edge set in Fig. 7.

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* $\mathscr{F}$ defined by

$$\mathscr{F}_{C,s} = \begin{cases} 1 & \text{if the directions of the edges in s} \\ & \text{are consistent with the traversal of C,} \\ -1 & \text{if the directions of the edges in s} \\ & \text{are inconsistent with the traversal of C,} \\ 0 & \text{if s is not in C.} \end{cases} \quad (3)$$

For example, the fundamental cycle-edge subset incidence matrix for the DAG in Fig. 7 is

$$\mathscr{F} = \begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{array} \begin{array}{c} \begin{array}{cccccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array}. \quad (4)$$

We define the vectors $\mathbf{w}$ (with coordinates $\mathbf{w_s}, \forall \mathbf{s} \in \mathbf{S}$, in the order of the columns of $\mathscr{F}$) by $\mathbf{w_s} = \psi_+(\mathbf{s}) - \psi_-(\mathbf{s})$ and $\mathbf{b}$ (with coordinates $\mathbf{b_C}, \forall \mathbf{C} \in \mathscr{C}$, in the order of the rows of $\mathscr{F}$) by $\mathbf{b_C} = \sum \chi(\mathbf{C})$. Now if ψ_+ and ψ_- specify a proper leveling of G then $\mathscr{F}\mathbf{w} = \mathbf{b}$, i.e. the dummy vertices balance the cycles.

So if we start with an initial leveling of G and add dummy vertices to edges to ensure level planarity with respect to T, we can compensate by examining the nullspace of $\mathscr{F}$. 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 $\mathscr{F}$ are linearly dependent.

4.4 Crossing Minimisation

An embedding algorithm is implied by `makeTreeDAGLevelPlanar`. Fixing columns (j_L, j_R, and j_F) decides the relative ordering of their corresponding vertices on their respective levels. However, we have some freedom when positioning the remaining vertices. For these we use an ordering induced by a Fiedler vector of $\mathscr{L}(G)$. This heuristic has previously been used to determine exact x-coordinates [5] when drawing a directed graph and for the 2-level crossing minimisation problem [20]. It is based on a result [22] 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 [16].

Given an undirected graph $G = (V,E)$, the MLA problem is to determine a bijection $f : V \to \{1 \dots n\}$ such that $\sum_{(u,v)\in E} |f(u) - f(v)|$ is minimised. Juvan et al. [16] use a Fiedler vector $\mathbf{x}$ of $\mathscr{L}(G)$ to induce an ordering (if $\mathbf{x_u} < \mathbf{x_v}$ then $f(u) < f(v)$) which is unique up to the relative ordering of repeated eigenvector elements. Shahrokhi et al. [22] 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 minimisation problem. Empirical evidence [20] suggests that this heuristic efficiently provides good solutions. We use this same ordering to decide on the relative position of the vertices on their respective levels as a means of reducing the number of crossings between the remaining edges.

5 An Application

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, more significantly, on certain application-specific packages. It is reasonable to assume that this graph has a significant spanning tree, e.g. the tree that propagates the most centrality, and we want to draw it so that this spanning tree is emphasized.

Figure 9 depicts the largest connected component of the dependency graph for packages related to the Python programming language[1]. The significant spanning tree (solid edges) was computed as in Sect. 3 and the orderings of the vertices on each level were determined by the embedding algorithm implied by `makeTreeDAGLevelPlanar` along with the crossing minimisation heuristic. The computation of the final x-coordinates was based on [3] and the less significant edges (dashed edges) were added in by hand.

The impetus for drawing the graph in this way is to emphasize the significant dependencies. For example, consider the dependencies on the **pygtk** package (shaded). The six packages that significantly depend on **pygtk** are easily discernable. However, **twisted**'s

[1]This graph was obtained automatically by querying the Gentoo package management system (*http://www.gentoo.org*) and removing any redundant dependencies by computing its transitive reduction.

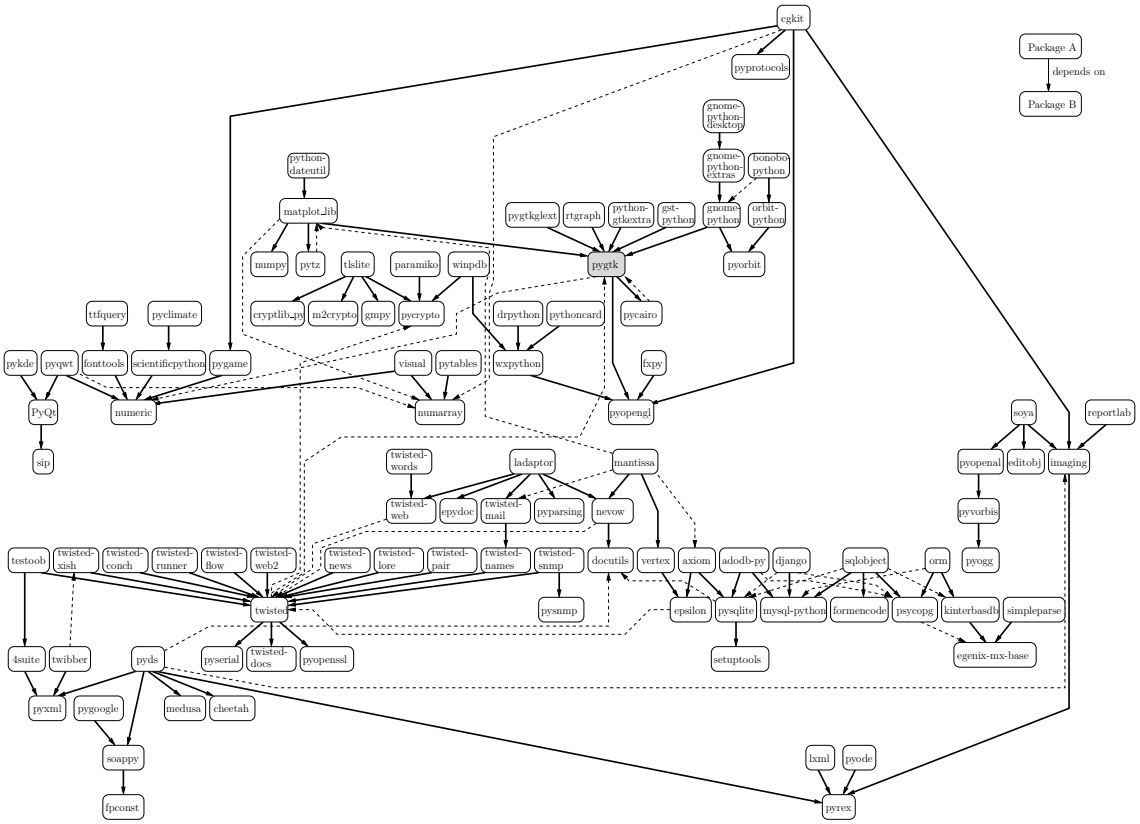

Figure 9: The dependency graph for Python packages.

dependency on **pygtk** is considered less significant and hence less obvious. With some knowledge of the domain, we know that **pygtk** consists of bindings for a GUI toolkit whereas **twisted** is concerned with networking and so this choice of significance has some merit, but of course our approach works equally well for any choice of significant spanning tree.

6 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 can cross. These steps can be performed in linear time using a sparse form of matrix storage.

The remaining freedom in the ordering of the vertices on their respective levels is used to reduce the number of 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 minimisation 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.

As a practical application, we used the approach to draw the significant spanning tree of the dependency relationship between the packages related to the Python programming language. By using appropriate visual cues, for example, drawing the less significant edges in a lighter colour, the significant spanning tree and hence, the significant dependencies, are easily discernable.

Future work will involve quantifying the total number of dummy vertices – both the dummy vertices used to ensure the level planarity of the spanning tree and those used to compensate for any upwardly directed edges. Evaluations, both experimental and user-based, may be beneficial in providing more conclusive evidence regarding the advantages of this approach when drawing directed graphs.

Acknowledgements

Martin Harrigan would like to acknowledge the support of the Irish Research Council for Science, Engineering and Technology.

References

[1] P. Bonacich. Factoring and weighting approaches to status scores and clique identification. *Journal of Mathematical Sociology*, 2:113–120, 1972.

[2] 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.

[3] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing '01*, pages 31–44. Springer, 2002.

[4] C. Buchheim, M. Jünger, and S. Leipert. Improving Walker's algorithm to run in linear time. In *Graph Drawing '02*, pages 344–353. Springer, 2002.

[5] L. Carmel, D. Harel, and Y. Koren. Combining hierarchy and energy for drawing directed graphs. *IEEE Transactions on Visualization and Computer Graphics*, 10(1), 2004.

[6] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-Trees. *Algorithmica*, 15(4):302–318, 1996.

[7] P. Eades and S. Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131(2):361–374, 1994.

[8] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[9] 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.

[10] P. Healy and A. Kuusik. Algorithms for multi-level graph planarity testing and layout. *Theoretical Computer Science*, 320(2-3):331–344, 2004.

[11] P. Healy, A. Kuusik, and S. Leipert. A characterization of level planar graphs. *Discrete Mathematics*, 280(1-3):51–63, 2004.

[12] L. S. Heath and S. V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part II. *SIAM Journal on Computing*, 28(5):1588–1626, 1999.

[13] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[14] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. In *Graph Drawing '98*, pages 224–237. Springer, 1998.

[15] D. Jungnickel. *Graphs, Networks and Algorithms*. Algorithms and Computation in Mathematics. Springer–Verlag, second edition, 1999.

[16] M. Juvan and B. Mohar. Optimal linear labelings and eigenvalues of graphs. *Discrete Applied Mathematics*, 36(2):153–168, 1992.

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

[18] T. Munzner. H3: Laying out large directed graphs in 3D hyperbolic space. In *INFOVIS'97*, pages 2–10. IEEE Computer Society, 1997.

[19] T. Munzner. Drawing large graphs with H3Viewer and site manager. In *Graph Drawing '98*, pages 384–393. Springer, 1998.

[20] M. Newton, O. Sýkora, and I. Vrt'o. Two new heuristics for two-sided bipartite graph drawing. In *Graph Drawing '02*, pages 312–319. Springer, 2002.

[21] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[22] F. Shahrokhi, O. Sýkora, L. A. Szkely, and I. Vrt'o. On bipartite drawings and the linear arrangement problem. *SIAM Journal on Computing*, 30(6):1773–1789, 2001.

[23] 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.

[24] J. Q. Walker. A node-positioning algorithm for general trees. *Software – Practice and Experience*, 20(7):685–705, 1990.

[25] G. J. Wills. NicheWorks – interactive visualization of very large graphs. *Journal of Computational and Graphical Statistics*, 8(2):19–212, 1999.