Smashing Peacocks Further: Drawing Quasi-Trees from Biconnected Components

Daniel Archambault, Tamara Munzner, *Member, IEEE*, and David Auber

Abstract— Quasi-trees, namely graphs with tree-like structure, appear in many application domains, including bioinformatics and computer networks. Our new SPF approach exploits the structure of these graphs with a two-level approach to drawing, where the graph is decomposed into a tree of biconnected components. The low-level biconnected components are drawn with a force-directed approach that uses a spanning tree skeleton as a starting point for the layout. The higher-level structure of the graph is a true tree with meta-nodes of variable size that contain each biconnected component. That tree is drawn with a new area-aware variant of a tree drawing algorithm that handles high-degree nodes gracefully, at the cost of allowing edge-node overlaps. SPF performs an order of magnitude faster than the best previous approaches, while producing drawings of commensurate or improved quality.

Index Terms—Graph and Network Visualization, Quasi-Tree

1 Introduction

Several approaches to graph drawing have used a spanning tree to accelerate or improve the visual quality of general graph layout, in applications such as: bioinformatics [1]; computer networking [8]; web site design and software engineering [17]; and co-citation analysis [6]. These methods succeed when the graph is a **quasi-tree**; that is, a graph with tree-like structure. Intuitively, for these types of data, using a spanning tree skeleton for layout is useful, even when the graph is much more densely connected than a strict hierarchy. Many definitions for quasi-tree have been proposed, for example a graph with a limit on the number or graph-theoretic length of non-tree edges. Here, we define quasi-trees as graphs where the number of biconnected components is a constant multiple of the number of vertices. A **biconnected component** is a maximal subgraph where the removal of any node or edge from that subgraph does not disconnect it into two or more components. More formally, for a graph $G(V,E)$ with a set of V nodes and E edges, we consider it a quasi-tree if it has $O(|V|)$ biconnected components. The **biconnected component tree** of a graph is a tree of meta-nodes, representing biconnected components, which conveys important structural information.

In the telecommunications network domain, the biconnected components of a quasi-tree can highlight semantic information in the network such as subnetwork structure. A decomposition of the quasi-tree into biconnected components highlights strengths and weaknesses in network design [20]. In Internet tomography, seeing the subnetwork structure in the context of the global structure of the Internet is important in several user scenarios, including: visualizing infrastructure attacks on a corporation, discovering prolonged outages due to natural disasters or war, and tracing anonymous packets from an attack on a network back to their source [8].

In bioinformatics, graphs with quasi-tree structure appear in gene function, protein-protein interaction, and biochemical pathway data [1]. With the LGL system [1], the authors exploit quasi-tree structure to produce impressive drawings of protein homology maps. In these networks, nodes are proteins and there exists an edge between a pair of nodes if their aligned amino acid sequences have a high similarity score. When drawn in a certain way, homology maps cluster proteins of similar function together, allowing biologists to predict the function of unknown proteins. To make these predictions, we need to see localized protein families in the context of the homology map.

In these applications, both the **low-level** structure within each biconnected component, and the **high-level** of the how those components interconnect in a tree, are important in understanding the graph. Many of the important low-level features in protein homology maps involve the biconnected components in the graph. For example, fusion proteins that connect two functionally related families of proteins often appear as a node separating two biconnected components [1]. For Internet tomography, we believe that biconnected components are also of interest because they distinguish between peering relationships and downstream clients.

Some previous approaches to drawing quasi-trees are fast and explicitly exploit quasi-tree structure, but show only a subset of the full graph, relying on extensive interactive exploration to eventually reveal the complete structure [17, 6]. Other general approaches to graph drawing are fast and show the full graph, but produce drawings of limited visual quality [12]. The recent LGL system exploits quasi-tree structure to produce high-quality drawings of the full graph in the bioinformatics domain [1]. While faster than previous work in the networking domain [8], layout of large datasets still takes hours to compute.

The SPF system exploits the quasi-tree structure of the graph by using a two-level approach to drawing: one algorithm to draw the biconnected component of each meta-node, and a second to draw the higher-level structure of the tree of these meta-nodes. The primary contribution of this paper is to draw the full graph structure of quasi-trees much faster than previous work, and to provide equivalent or better visual quality with drawings that accurately portray the structural information of interest in application domains. We also improve the visual quality results of the LGL algorithm while keeping the algorithm running time within the same order of magnitude as the original version of LGL. Finally, we introduce an area-aware version of the RINGS tree drawing algorithm [22] to draw our biconnected component tree.

2 Definitions

A **spanning tree** is a subset of edges in a connected graph that is a tree incident to all nodes. A **minimum spanning tree** is a spanning tree of minimum cost on a graph of weighted edges. For unweighted graphs, the weights of all edges can be set to one.

A **biconnected component decomposition** divides a graph into biconnected components. After a biconnected component decomposition, by definition, the biconnected components are linked by edges and nodes whose removal would disconnect the graph into two or more components. We define **bridge nodes** and **bridge edges** as the nodes and edges which can be removed to disconnect the graph into two or

- *Daniel Archambault and Tamara Munzner are with the University of British Columbia, E-mail: {archam, tmm}@cs.ubc.ca.*
- *David Auber is with the Université de Bordeaux, E-mail: auber@labri.fr*

Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

more components. Bridge nodes are also called articulation or cut vertices elsewhere in the literature. Bridge edges correspond to edges incident to two bridge nodes. If we replace each biconnected component with a single meta-node, we create a new higher-level graph with meta-nodes linked by only the bridge nodes and edges. The topology of this graph is always a tree, and we refer to it as the **biconnected component tree**. We use this more evocative name here, rather than the synonymous term block-cut point tree found in other areas of the literature.

3 Related Work

Related work for quasi-trees is divided into three categories: multi-level graph drawing algorithms, spanning tree visualization, and domain-specific graph visualization.

3.1 Multi-Level Graph Drawing

Multi-level schemes for graph drawing have been studied extensively to improve the running time and visual quality of large graph layouts. In multi-level algorithms, a coarsening operator is recursively applied to an input graph, forming a hierarchy of coarse graphs which are similar in structure, but cheaper to lay out. The coarsest graph of the hierarchy is laid out first and the locations of its nodes are used as an estimate for the next finest level. Coarsening operators suggested in the literature include: estimates of maximal matching [23], graph filtration based on shortest path distance in GRIP [10], eigenvector computation on coarse approximations of the Laplacian matrix in ACE [15], and solar system models based on local graph connectivity in FM^3 [12]. These algorithms produce layouts of high visual quality for several types of undirected graphs, but none of them take into account the topological features that are integral to quasi-trees.

TopoLayout [3, 2] was the first multi-level algorithm to take topological features into account. Previous algorithms have exploited topological features [18, 20], but never in a multi-level context. The coarsening operator of TopoLayout recursively detects topological features such as trees, biconnected components, and highly connected clusters in the graph and draws them with appropriate algorithms. In the quasi-tree problem, we are only interested in specific types of topology, mainly trees and biconnected components. SPF is built inside the TopoLayout framework, but is directed specifically to the quasi-tree problem. SPF is similar to the work of Six and Tollis [20], but it uses a more sophisticated approach to drawing biconnected components than their method, which uses circular layout.

3.2 Spanning Tree Visualization

Spanning tree methods have appeared previously in the graph drawing and information visualization literature, but they only draw a subset of the graph edges and depend heavily on user interaction for understanding graph structure. In contrast, SPF attempts to convey the full complexity of the underlying graph at all times.

The H3 system [17] uses domain-specific knowledge to determine a spanning tree for the graph. Node positions depend only on the chosen spanning tree, and the drawing of non-tree edges is toggled based on user selection.

Boutin and Hascoët [6] characterize *tree-like* graphs as having non-tree links only below a threshold graph-theoretic distance. They support filtering a potentially dense general graph to create a tree-like graph by eliminating graph edges that do not fit their constraint. An interactively chosen focus node is used as the spanning tree root.

3.3 Domain-Specific Graph Visualization

Two key domain-specific papers are the primary inspiration for this work. Cheswick *et al.* [8] presented drawings of substantial portions of the Internet. They mapped the hardware structure of the Internet using traceroute packets from a source machine. Outgoing paths were tracked to determine the connectivity between servers. In order to visualize the collected data, the authors employ a force-directed approach. The first optimization eliminates sufficiently distant nodes from the repulsive force calculations. The second optimization is the use of a spanning tree as a skeleton for the force-directed layout. The nodes of the test servers are laid out in the centre of the drawing. Iterations of force-directed layout are applied until this graph reaches equilibrium. Nodes are added to the layout in breadth-first order using a spanning tree centred at the traceroute packet source. The edges that connect these nodes to previously placed nodes are added. The result has been described as a "smashed peacock on a windshield" by Dave Presotto [8]. Our biconnected component decomposition divides this dataset into smaller components, inspiring the title of this paper. We could not directly compare our results with this system because the code is proprietary.

In bioinformatics, Adai *et al.* [1] implemented the approach of Cheswick *et al.* [8] to draw protein homology maps. In their system, LGL, the spanning tree used as a skeleton is computed on a graph with edges weighted by BLAST e-values. The root vertex of this graph is chosen arbitrarily, is user specified, or is chosen based on graph centrality. As in Cheswick *et al.*, by embedding the graph in a grid, repulsive forces between sufficiently distant nodes are culled. Two nodes share a repulsive force if they are present in the same or adjacent cells. LGL was tested on several protein homology maps and the layouts grouped proteins into families of related function. In this work, we improve some aspects of the LGL drawing algorithm and integrate the improved algorithm into SPF. Drawings produced by SPF are an order of magnitude faster than those produced by LGL. They retain much of the high-level graph structure and better illustrate the low-level graph structure in the quasi-tree.

4 Algorithm

The inputs to SPF are a graph and an unrooted, minimum spanning tree. For weighted graphs we use Kruskal's algorithm [21] to compute the spanning tree, and for unweighted graphs we use breadth first search [5]. For unweighted graphs, all spanning trees are of equal cost. Spanning trees computed from a breadth-first search are a logical choice to use as they tend to be representative of the underlying distance structure in the graph [21]. We root the input spanning tree at its tree centre.

The drawing algorithm runs in three phases:

1. Decompose the graph into biconnected components.
2. Draw each component using an improved version of LGL.
3. Draw the biconnected component tree using an area-aware version of the RINGS [22] algorithm.

Our first phase decomposes the input graph into biconnected components. In the second phase, the algorithm draws each biconnected component with our improved version of LGL. The sizes of the nodes in the biconnected component tree are set to the size of the biconnected component drawings. The third phase draws the biconnected component tree using an area-aware version of the RINGS [22] algorithm that we have developed.

4.1 Decomposition into Biconnected Components

In SPF, the graph is decomposed into biconnected components using a standard algorithm from the literature [5]. It works by performing a depth-first search through the graph. Edges that point back to higher levels of the depth-first search are called back edges. When a subtree s of the depth-first search tree has no back edges to any ancestor of s, it is a separate biconnected component. The algorithm has a running time of $O(|V|+|E|)$.

The biconnected component tree is constructed as shown in Figures 1(a) and 1(b). Bridge edges, such as the edge between components C_3 and C_4, are edges in this tree. Bridge nodes appear as nodes in the tree. If a bridge node shares two or more edges with a component, the bridge node is duplicated and placed into those adjacent components as shown with C_1 and C_2. This duplication keeps nodes directly adjacent to the bridge node together when during layout of the biconnected component, but the duplicated bridge nodes are not actually drawn.

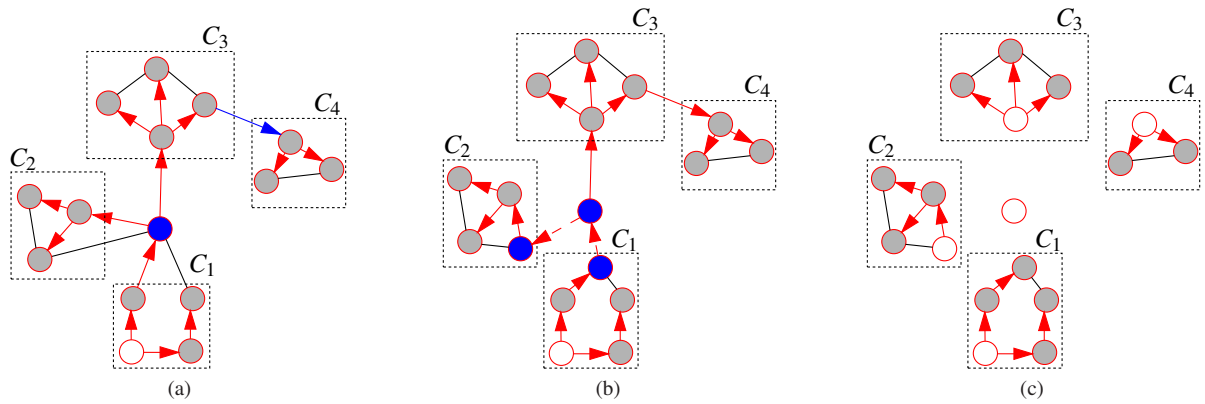


Fig. 1. Decomposition of the graph into biconnected components preserves the input spanning tree. Spanning tree edges are shown with directed red edges, and the roots are shown in white. **(a)** Identify the bridge nodes and edges, shown in blue, in the decomposition phase. **(b)** If a bridge node shares two or more edges with a component, the bridge node is duplicated and placed into those adjacent components. **(c)** Break spanning tree up into individual biconnected components, and find the root for each new component.

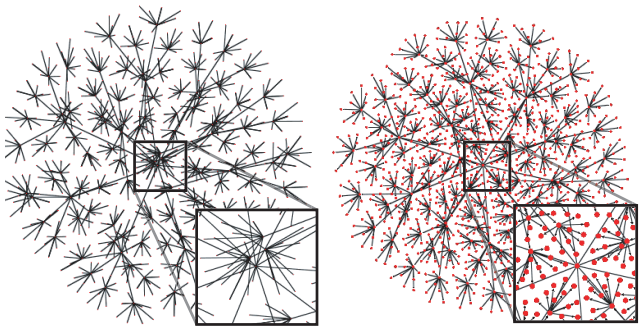


Fig. 2. Comparison of the final layout of a ten-ary tree of depth three between **(a)** LGL and **(b)** optimized LGL. Repulsive forces are diminished in optimized LGL to roughly the size of a node.

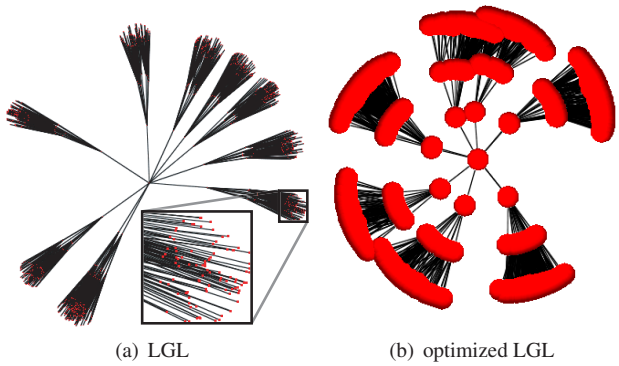


Fig. 3. Comparison of the initial placement of nodes between **(a)** LGL and **(b)** optimized LGL, on a ten-ary graph of depth tree. Fan placement also places nodes closer to each other at the beginning.

The decomposition phase does not take into account the prespecified spanning tree. Our choice of spanning tree could only be limited by the biconnected component decomposition if the chosen spanning tree did not contain all of the bridge nodes and bridge edges of the biconnected component tree; the remaining edges in the biconnected components can be present or absent from the spanning tree as needed.

All bridge edges need to be present in the spanning tree. The bridge edge is the only edge connecting two biconnected components in the graph, and, if it were not present, the tree would be disconnected. Bridge nodes must be present by the definition of a spanning tree, and the edges between them and their duplicate nodes must be present as they are bridge edges. Therefore, a biconnected component decomposition does not limit our choice of spanning tree.

4.2 Biconnected Component Drawing with Optimized LGL

Once the graph is divided into biconnected components, we use our improved version of LGL, **optimized LGL**, to draw each biconnected component. The roots are computed from the spanning tree as shown in Figures 1(b) and 1(c). If the root of the input spanning tree is present in a biconnected component, it is chosen as the root of the biconnected component. Otherwise, the node from which the spanning tree entered this biconnected component on a depth-first search from the root of the spanning tree is used. The input spanning tree continues through the biconnected component as specified.

The first optimization to LGL improves running time in some cases and leaves visual quality unchanged. Recall that LGL uses a grid to cull repulsive forces of sufficiently distant nodes. To compute the repulsive forces, LGL marches through each cell of the grid. Computing repulsive forces in this manner can be costly in the early stages of the algorithm when many cells are empty. Instead, we keep a list of nodes already placed by the spanning tree, determine the position of the node in the grid, and compute the repulsive forces directly. We then mark the cell to ensure that the repulsive forces between nodes in a cell are computed only once. The drawing remains unchanged, because exactly the same set of repulsive forces are computed as when marching through the grid.

The second optimization improves the visual quality of the final drawing of the graph, as shown in Figure 2. This optimization consists of two parts that influence initial placement of the nodes in the layout. In LGL, nodes are placed into the layout using the input spanning tree as a skeleton according to $\vec{S}$:

$$\vec{S} = c(\hat{M} + \hat{P}) + x_{\text{parent}} \tag{1}$$

where c is a constant proportional to the number of nodes in the graph, $\hat{M}$ is the centre of mass, and $\hat{P}$ is the vector between the parent and the grandparent of the placed node in the input spanning tree. Both $\hat{M}$ and $\hat{P}$ are normalized. The value of x_{parent} is the position of the parent in the spanning tree.

The first part improves placement by reducing the constant c to the sum of the size of the parent node, the size of child node, and the average size of the nodes in the graph. Reducing c yields more compact drawings. In Figure 2(a) the nodes are spread far apart and can

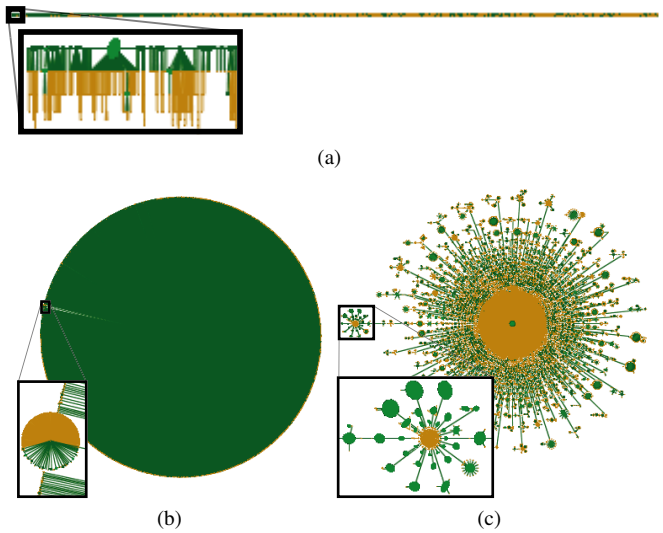


Fig. 4. Layouts of the Cheswick 2005 dataset biconnected component tree using: **(a)** area-aware Walker's algorithm [7], **(b)** bubble tree [11], and **(c)** our adaptation of area-aware RINGS. With area-aware RINGS, we gain a significantly more compact drawing at the price of introducing edge-node overlaps.

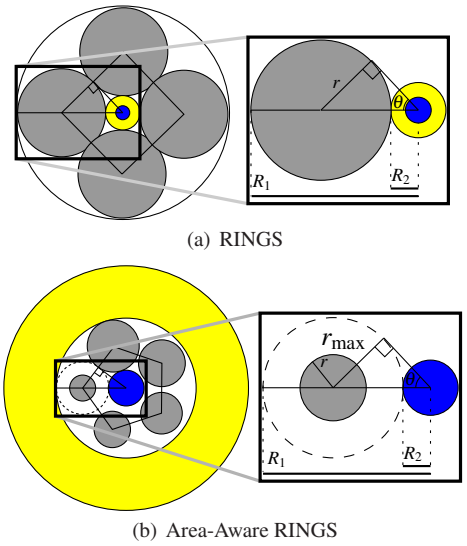


Fig. 5. Comparison of subtree placement between **(a)** the RINGS algorithm and **(b)** Area-Aware RINGS. The blue node is the root of the current subtree. The grey circles are subtrees that have filled the white ring. The yellow circle indicates where the next ring is started.

hardly be seen while in Figure 2(b) they are easily visible along with the topology. The second part places nodes on small directed fans in the direction of the vector $\vec{S}$ as shown in Figure 3(b) instead of circles as shown in Figure 3(a). Placing the nodes on fans inhibits them from being directed back inwards, preserving low-level structure. We can see the problem near the root node of the tree in Figure 2(a). In contrast, the problem is less prevalent in optimized LGL as demonstrated by Figure 2(b).

4.3 Biconnected Component Tree Drawing with Area-Aware RINGS

Once each biconnected component has been laid out, we know the screen-space extents required by each of these meta-nodes. In order to draw the biconnected component tree, we need a tree drawing algorithm that is **area-aware**; that is, that can correctly handle nodes of variable rather than uniform size.

To begin this section, we first motivate the need for area-aware RINGS. Next, we describe the original RINGS algorithm as presented by Teoh and Ma [22]. Then, we describe our area-aware adaptation of RINGS and discuss how to use it to draw our biconnected component trees.

For large quasi-trees, the degree of nodes inside the biconnected component tree can be very high. As we can see in Figure 4, the few previously existing tree layout algorithms that are area-aware perform poorly in this case. The area-aware version of the Walker [7] layout, shown in Figure 4(a), lays out the children of the high-degree node on a horizontal line, and the details of their structure are too small to be seen without zooming. Similarly, Figure 4(b) shows that Bubble Tree [11] lays out the high-degree node in the centre of a circle so large that the children are too small to be seen individually on the circumference. In contrast, the RINGS algorithm [22] provides a much more compact drawing at the price of introducing node-edge overlaps and edge crossings. By adapting it to be area-aware, we achieve the result shown in Figure 4(c).

In RINGS [22], each child of a root is placed in a circle enclosing its entire subtree as shown in Figure 5(a). The dark blue node at the centre of the drawing is the root while the dark grey circles are the children of the root and their subtrees. The root is placed at the centre of the drawing, and we sort its children by their number of children; that is, by their number of grandchildren with respect to the root. The subtrees are placed onto concentric rings inward towards the root in order from the child with the most to least children. Each of the subtrees on a ring is given an equal-sized enclosing circle of radius r. The radii R_1 and R_2 are as shown in Figure 5(a). Let $N_{\text{GrandPlaced}}$ be the number of grandchildren placed in the current ring. Let $N_{\text{GrandTotal}}$ be the total number of grandchildren to place with respect to the root. A new ring is started on the inner, yellow circle when:

$$\frac{N_{\text{GrandPlaced}}}{N_{\text{GrandTotal}}} > \frac{{R_2}^2}{{R_1}^2} = \frac{(1-\sin(\frac{\pi}{n}))^2}{(1+\sin(\frac{\pi}{n}))^2} \tag{2}$$

Given n children and their subtrees, the right hand side of Equation (2) is the ratio of the circle areas with radii R_1 and R_2 respectively. This ratio does not depend on r and only requires that $R_1 = 2r + R_2$. The process is repeated for the remaining subtrees.

From the left hand side of Equation (2), we notice that the size of the nodes in the tree is not considered, only the number of children. Therefore, RINGS assumes a uniform node size. Moreover, the node size should be much smaller than r to ensure that the entire subtree is fully contained by the enclosing circle, since only grandchildren, and not the entire subtree, are considered in this ratio. In our work, this subtree could contain a large number of nodes of substantial size. Therefore, we cannot choose such an r.

In our area-aware variant of RINGS, instead of counting the number of grandchildren, we determine the area needed to lay out each subtree by drawing the tree bottom-up. At a leaf node, we use the bounding circle of the node. At an interior node, we use the bounding circle of the subtree of which it is the root. The subtrees are placed into concentric rings outward from the root in order from the subtree which requires the least area, to the subtree which requires the most area, as shown in Figure 5(b). We keep track of the largest enclosing circle radius in $r_{\max}$. The radii R_1 and R_2 are as shown in Figure 5(b) with $R_1 = 2r_{\max} + R_2$. A new ring is started on the outer, yellow circle when:

$$\frac{A_{R_2}}{A_{R_1}} < \frac{{R_2}^2}{{R_1}^2} = \frac{(1-\sin(\frac{\pi}{n}))^2}{(1+\sin(\frac{\pi}{n}))^2} \tag{3}$$

where A_{R_1} and A_{R_2} are the areas of the circles with radii R_1 and R_2. Since we have drawn the tree from the bottom up, we know the actual areas of these circles and can compute the ratio directly. Where Equation (2) compares ratios of grandchildren, Equation (3) compares ratios of the areas needed to draw the nodes and subtrees.

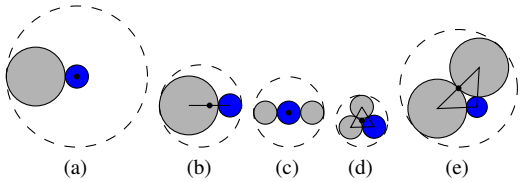


Fig. 6. Comparison of special cases for small subtree layout. **(a)** One subtree in RINGS. **(b)** One subtree in area-aware RINGS. **(c)** Two subtrees in RINGS. **(d)** Two subtrees in area-aware RINGS. When the triangle connecting the three centres of the root and subtrees is obtuse, the centre of the bounding circle is placed at the centre of the sum of the two largest diameters as shown in **(e)**.

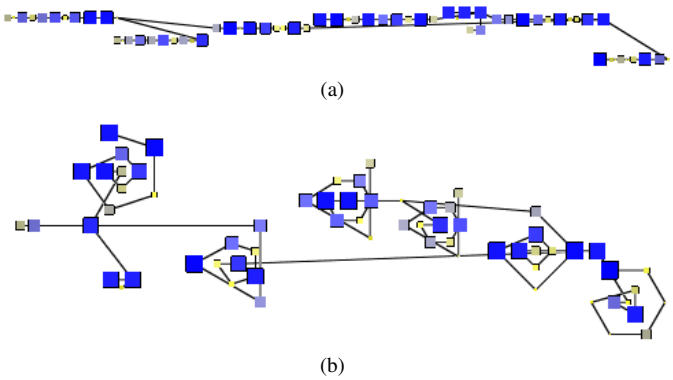


Fig. 7. Demonstration of the chain optimization, with nodes set to random sizes between one and ten unite. **(a)** Node drawn in linear chains. **(b)** More compact drawing of chain nodes in a spiral.

We further optimize area-aware RINGS when the number of subtrees to be placed on a ring is two or fewer as shown in Figure 6. The root node of a subtree is usually placed at the centre of its bounding circle, which leads to wasted space as shown in Figure 6(a) and 6(c). In the case of a root node with a single child subtree, we place the root and its child subtree tangent to each other as shown in Figure 6(b). In the two-child subtree case, we place the root and its two subtrees tangent to each other in a triangle as shown in Figure 6(d). When the triangle connecting the three centres is acute, the smallest bounding circle is the outer circle tangent to all three enclosing circles. This circle is known as the outer Soddy circle [9, 14]. When the triangle is obtuse, the bounding circle is the circle enclosing two largest circles as shown in Figure 6(e).

We also improve how area-aware RINGS handles chains. A chain is defined as a linear sequence of nodes each having exactly one child. Without this improvement, each node of the chain is the case shown in Figure 6(b), resulting in the long lines of nodes as seen Figure 7(a). In our optimization, we treat each node of the chain as though it was a direct child of the node which began the chain. The chain spirals around the node that began it, drawing it compactly, as shown in Figure 7(b). Unlike other cases in area-aware RINGS, the nodes of a chain are not sorted by size.

We use a subtle three-dimensional depth effect to create perceptual layering when drawing edges. Edges are placed below the nodes, with edges to the outer rings placed more deeply than edges to the inner ones. When browsing the layout in Tulip [4], edges can be raised if a path between two nodes needs to be visible. Edge colour for edges between biconnected components is lightened.

We also choose an ordering for the biconnected components in a ring to reduce edge occlusion, since all biconnected components in the same ring have an enclosing circle of at most $r_{\max}$. The chosen ordering of the biconnected components in a ring is based on the positions of the nodes inside the biconnected component at the root. The vector between every node inside the root that attaches the biconnected component on the ring and the centre of the drawing at the root is computed. The average vector is taken as the vector of ideal placement. Biconnected components in a ring are sorted based on the direction of their ideal placement vector. The biconnected component with the most edges to the root is placed in its ideal location on the ring. All other biconnected components are placed in their sorted order.

5 Empirical Testing and Results

We implemented SPF in the Tulip [4] framework and now compare it to three other algorithms in terms of performance, qualitative visual results, and quantitative statistics. We do so using two large datasets. `Protein`, the graph shown in Figure 8 with 30,727 nodes and 1,206,654 edges, is the unweighted version of the protein homology graph presented in the LGL paper [1]. `Net05`, the graph shown in Figure 9 with 190,384 nodes and 228,354 edges, is an Internet tomography dataset similar to those presented in Cheswick *et al.* [8], but generated in 2005 by Cheswick's Internet Mapping Project[1]. All benchmarks were run on a 3.0GHz Pentium IV with 3.0GB of memory running SuSE Linux with a 2.6.5-7.252 kernel. The accompanying video[2] shows interactive exploration of these graphs at multiple levels of zooming.

Space constraints preclude showing all competing algorithms, so we compare SPF to its most competitive. FM^3 [12] is a state of the art multi-level graph drawing algorithm. Other algorithms, including ACE [15], GRIP [10], and HDE [16], were shown to be less competitive than FM^3 in previous work [13, 3]. LGL is an algorithm developed in the bioinformatics domain for visualizing quasi-trees and optimized LGL is our modified form of LGL. TopoLayout [3] took too long to cluster `Protein`, because it used a clustering algorithm whose performance deteriorates as the number of edges becomes large. The TopoLayout drawing of `Net05` was not compact, because it drew most of the biconnected component tree using Bubble Tree. Its drawing is similar to Figure 4(b).

5.1 Performance

The FM^3 algorithm was the fastest algorithm on both datasets. On `Protein`, it was an order of magnitude faster than all the drawing algorithms. FM^3 was the same order of magnitude as SPF on `Net05`, but three times faster. LGL and optimized LGL were the slowest algorithms on both datasets. They are two times slower than SPF on `Protein`. On `Net05`, LGL and optimized LGL are an order of magnitude slower than all algorithms. SPF is twice as fast as LGL and optimized LGL on `Protein` and an order of magnitude faster than LGL and optimized LGL on `Net05`.

5.2 Qualitative Drawing Comparison

FM^3 has difficulty depicting both the high-level and low-level structure in both datasets, as shown in Figures 8(a) and 9(a). The high-level, tree-like structure is unclear in `Protein`. On `Net05`, the high-level tree structure is somewhat visible, but details of it are difficult to see, because it draws the branches along thin lines. It is nearly impossible to see low-level structure in either dataset.

With the slower LGL and optimized LGL, the drawings are improved. The high-level, tree-like structure is apparent in `Protein` throughout the dataset as shown in Figures 8(b) and 8(c). We can clearly see high-level branches without zooming in, as well as more of the tree structure in the insets. However, the drawing of `Net05`, shown in Figures 9(b) and 9(c), only displays the high-level tree structure well at the periphery. Most of the drawing is a featureless core where the tree-like structure is hidden. In terms of low-level structure, we are able to see protein families and fusion proteins between families in `Protein`. In `Net05`, the subnetwork structure is only clear when the nodes lie on the fringes of the drawing, as we see in the insets of Figures 9(b) and 9(c).

[1] `research.lumeta.com/ches/map`
[2] `www.cs.ubc.ca/labs/imager/video/2006/spf`

Algorithm	Total	Major	Total	Major
	Protein		Net05	
	Nodes			
FM^3	95	95	381	381
LGL	920	809	6,761	5,746
LGL Opt.	54,255	13,021	60,218	1,204
SPF	71,574	12,167	4,185	42
	Biconnected Components			
FM^3	2,400	2,376	162,620	160,993
LGL	2,657	2,603	170,073	3,401
LGL Opt.	2,955	2,629	93,570	1,871
SPF	0	0	8	1

Fig. 10. Node-node overlaps. We first give the total number of node-node overlaps, then only the number of major overlaps; that is, those where the overlap covers more than half of the node area. Protein has 30,727 nodes and 2,427 biconnected components. Net05 has 190,384 nodes and 167,460 biconnected components.

Algorithm	Overall	Within	Between
	Protein		
FM^3	1.02	0.61	0.94
LGL	0.57	0.32	0.88
LGL Opt.	0.72	0.33	0.78
SPF	2.74	0.32	0.74
	Net05		
FM^3	0.62	0.17	0.48
LGL	1.21	0.19	0.93
LGL Opt.	1.26	0.18	0.98
SPF	1.96	0.24	5.03

Fig. 11. Standard deviation of normalized edge lengths, where lower numbers mean more uniformity. Overall is the standard deviation over all edges. Within is the average standard deviation of the edges within each biconnected component, and Between is the standard deviation of edges that connect the components in the biconnected component tree.

SPF improves upon the running time of LGL and optimized LGL and retains or improves much of the high-level and low-level structure. Much of the high-level tree structure is retained with SPF. The spanning tree skeleton is made visually apparent with area-aware RINGS, but at a cost of spatial locality and edge crossings in the drawings.

The principal advantage of SPF over previous work is the improved visualization of low-level structure in the graph, because it is not as occluded by the higher-level components. In Protein, we see protein families and the fusion proteins between them as shown in the insets of Figure 8(d). The core of Protein is thinned, revealing more internal structure than with previous algorithms. Protein families and fusion proteins are clearly seen in the rings of the drawing. The core of Net05 is far smaller than with LGL; however, it still contains about 37,000 nodes and suffers from a great deal of node and edge occlusion. Nevertheless, we can resolve many local network features in the context of the entire Internet. We clearly resolve subnetwork structure around servers at the University of British Columbia and the City of Baltimore as shown in the insets of Figure 9(d).

5.3 Statistical Analysis

In addition to the qualitative analysis of the drawings, we provide quantitative statistics for each of the four layouts. We compute node-node overlaps and uniformity of edge lengths, both for the low-level structure of individual nodes and edges, and for the high-level structure of the biconnected components.

5.3.1 Node-Node Overlaps

A node-node overlap is simply the intersection of two nodes in a drawing. For the biconnected components, a node-node overlap occurs when the two convex hulls of the biconnected components intersect. A smaller number of node-node overlaps in the biconnected components more clearly displays high-level structure and better represents the low-level structure as the biconnected components do not occlude each other.

We present the node-node overlap statistics in the top of Figure 10. We show the total number of overlaps, and the number of major overlaps where more than half the the area of the smallest node is covered. Major overlaps are more interesting than total overlaps as they effect the readability of the drawing more severely.

We see that FM^3 has few node-node overlaps on either dataset. However, in this approach, the nodes are spread very far from each other with respect to the standard node size. LGL and optimized LGL incur more overlaps, but have the benefit of a more compact drawing. Optimized LGL incurs many more total node-node overlaps as we have reduced the magnitude of the repulsive force constant. However, only a small percentage of them are major. With SPF, although Protein has a large total number of overlaps, the number of major overlaps is less than that for optimized LGL. We see a significant reduction in the number of node-node overlaps on Net05. The use of area-aware RINGS to draw the very large number of biconnected components reduces the possibility of low-level node-node overlaps.

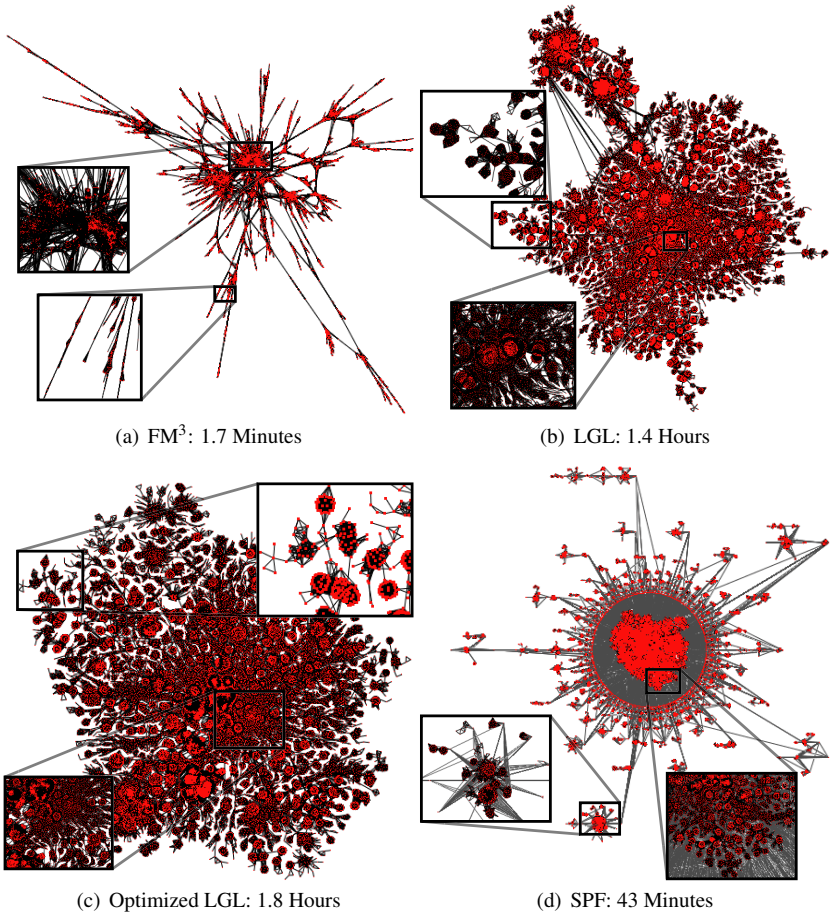
In the bottom of Figure 10, we present the overlap figures for the higher-level biconnected components. FM^3, LGL, and optimized LGL have thousands of major overlaps. These major overlaps make it difficult to see the low-level structure of the biconnected components in their higher level context. In these drawings, many but not all, of the large overlaps of biconnected components are with the large, biconnected core which is spread through the drawing. It is important to note that optimized LGL is better able to separate the biconnected components than the original LGL. This result supports our optimization of placing nodes on directed fans and reducing the repulsive force constant, which keeps nodes in biconnected components closer together. In contrast to these three methods, SPF succeeds in making these biconnected components more evident. It incurs no overlaps at all for the smaller Protein dataset, and only one major overlap for the larger Net05.

5.3.2 Uniformity of Edge Lengths

Uniform edge lengths keep all elements of the graph drawing at a similar scale. For each drawing, we compute the standard deviation of the edge lengths for each drawing. The raw edge length values are normalized by the average edge length on each dataset. This normalization sets the mean edge length in each drawing to one, so that the standard deviations can be directly compared. Standard deviations are all positive and numbers closer to zero correspond to more uniform edge lengths. When we consider the graph as a whole, SPF has highly nonuniform edge lengths. However, considering the uniformity with a particular level of structure shows its benefits.

The results are presented in Figure 11. The overall standard deviation is presented the left hand column. FM^3, LGL, and optimized LGL perform well on this metric where SPF does not. From visual inspection of the SPF drawings, we can see that this additional variance is probably due to the long edges introduced by the area-aware RINGS algorithm. However, uniform edge lengths across the entire drawing may not be appropriate for displaying the biconnected structure of quasi-trees. In the drawings produced by LGL and optimized LGL, we see this property as the drawings have a uniform but featureless core. Noack [19] stated that long edges may be required to display cluster structures. We propose that a more suitable metric for quasi-trees is to consider uniformity within a meaningful group; that is, the edges within a particular biconnected component, and the edges of the quasi-tree that connect between biconnected components.

Figure 11 shows these separate standard deviations in the centre and right columns. We see that optimized LGL is commensurate with LGL on nearly all numbers. SPF is commensurate with all algorithms in terms of the average standard deviation of edge lengths within biconnected components on both datasets. SPF has a slight improvement for the between edges in Protein, but a very high standard deviation for Net05. This situation follows directly from the size of their



(a) FM^3: 1.7 Minutes (b) LGL: 1.4 Hours

(c) Optimized LGL: 1.8 Hours (d) SPF: 43 Minutes

Fig. 8. Drawing of `Protein`, a protein homology map obtained from the LGL project. Graph contains 30,727 nodes and 1,206,654 edges. Drawings produced by **(a)** FM^3, **(b)** LGL, **(c)** optimized LGL, and **(d)** SPF, with drawing times indicated underneath.

biconnected component trees: small for the former, and large for the latter. The many concentric rings used by area-aware RINGS for large biconnected component trees contribute to this increased variance.

6 Conclusion and Future Work

We have presented SPF, a new drawing algorithm for quasi-trees. The principal advantage of SPF over previous work is the improved visualization of low-level structure within the biconnected components of the graph. These components are not as occluded by the higher-level structure of the tree formed by the interconnections between them. FM^3 is fast, but its visual quality is poor. The slower LGL approach dramatically improves the visual quality of the protein homology network dataset that it was designed for, but fails to visually distinguish the high-level structure of the Internet Mapping dataset. SPF does succeed at showing more structure for both of these datasets, as we argue both qualitatively in our discussion of the drawings and quantitatively with the very low number of biconnected component overlaps. Moreover, SPF is much closer to the speed of FM^3, ranging from twice as fast to an order of magnitude faster than LGL.

Our optimizations to LGL were designed to improve SPF itself, since it is used as one of our layout algorithms. However, considering optimized LGL as a standalone layout approach is also interesting. Although it is similar to LGL from visual inspection of the drawings, and has variable performance depending on the dataset tested, the biconnected component overlap statistic suggests that it does help distinguish more structure.

We adapted the RINGS algorithm to handle nodes of variable size, and optimized its behavior when handling small subtrees. Such an area-aware tree layout is useful not only within SPF, but also within other multi-level frameworks.

Reducing occlusion continues to be the driving problem for large-scale graph drawing, and many improvements remain for future work. The common approach of reducing edge-edge crossings would better show low-level structure, but we would also like to better show high-level structure by reducing the crossings between edges and the meta-nodes that constitute biconnected components, for example by improving the biconnected component tree drawing algorithm.

In this work we focus on the problem of exploiting quasi-tree structure to make better drawings of graphs that we assert are quasi-trees using our intuitive definition. A future challenge would be to build a quasi-tree detector that automatically determines whether a graph is a quasi-tree, allowing us to include quasi-trees as a topological feature type in the TopoLayout framework.

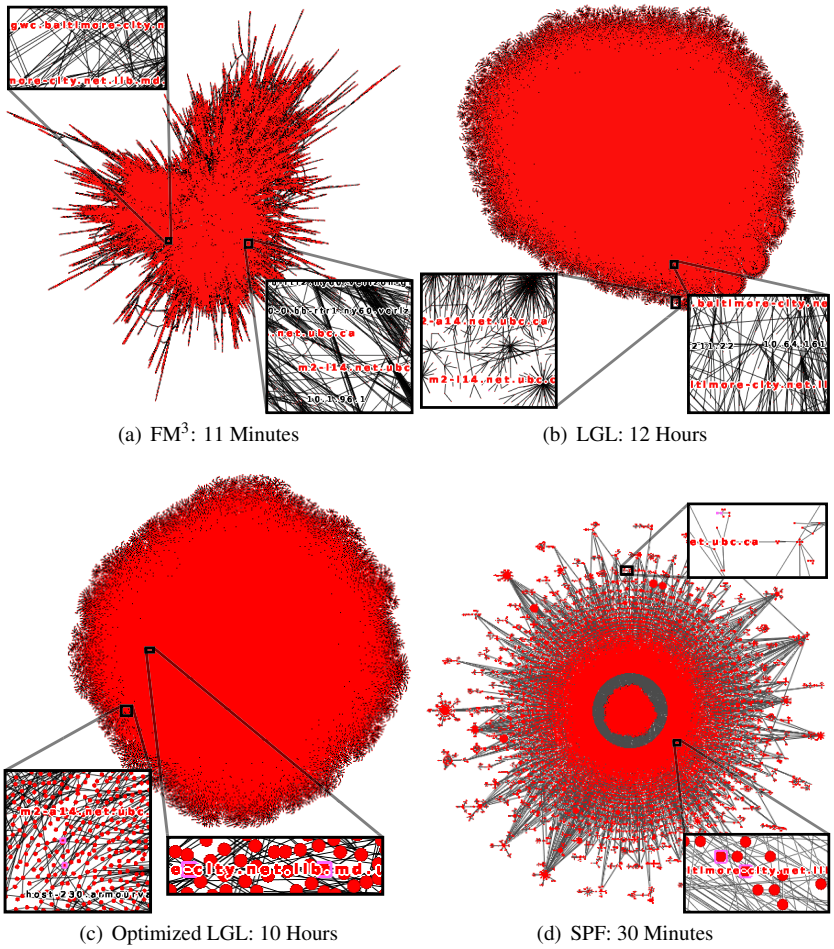
Acknowledgements

We would like to thank Ciarán Llachlan Leavitt for help in editing this paper. We would also like to thank Heidi Lam, Peter McLachlan, James Slack, and Melanie Tory for providing comments on previous drafts.

References

[1] A. T. Adai, S. V. Date, S. Wieland, and E. M. Marcotte. LGL: Creating a map of protein function with an algorithm for visualizing very large biological networks. *Journal of Molecular Biology*, 340(1):179–190, June 2004.

[2] D. Archambault, T. Munzner, and D. Auber. TopoLayout: Graph layout by topological features. In *IEEE Information Visualization Posters Compendium (InfoVis'05)*, pages 3–4, 2005.



(a) FM^3: 11 Minutes (b) LGL: 12 Hours

(c) Optimized LGL: 10 Hours (d) SPF: 30 Minutes

Fig. 9. Drawing of `Net05`, obtained from the Internet Mapping project. Graph contains 190,384 nodes and 228,354 edges. Drawings produced by **(a)** FM^3, **(b)** LGL, **(c)** optimized LGL, and **(d)** SPF, with drawing times indicated underneath.

[3] D. Archambault, T. Munzner, and D. Auber. TopoLayout: Graph layout by topological features. *Trans. on Visualization and Computer Graphics*, 2006. to appear.

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

[5] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 3rd edition, 2000.

[6] F. Boutin, J. Thièvre, and M. Hascoët. Focus-based filtering + clustering technique for power-law networks with small world phenomenon. In *Proc. of the Conference on Visualization and Data Analysis (VDA '06)*, 2006.

[7] C. Buchheim, M. Jünger, and S. Leipert. Improving Walker's algorithm to run in linear time. In *Proc. Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 344–353. Springer, Berlin, 2002.

[8] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the Internet. In *Proc. USENIX*, 2000.

[9] H. S. M. Coxeter. *Introduction to Geometry*. Wiley, 1969.

[10] P. Gajer and S. G. Kobourov. GRIP: Graph drawing with intelligent placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.

[11] S. Grivet, D. Auber, J. Domenger, and G. Melancon. Bubble tree drawing algorithm. In *International Conference on Computer Vision and Graphics*, pages 633–641, 2004.

[12] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Proc. 12th Int. Symp. on Graph Drawing*, volume 3383 of *LNCS*, pages 285–295. Springer-Verlag, 2004.

[13] S. Hachul and M. Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *Proc. 13th Int. Symp. on Graph Drawing*. Springer-Verlag, 2005.

[14] C. Kimberling. Central points and central lines in the plane of a triangle. *Mathematics Magazine*, 67(3):163–187, 1994.

[15] Y. Koren, L. Carmel, and D. Harel. Drawing huge graphs by algebraic multigrid optimization. *Multiscale Modeling and Simulation*, 1(4):645–673, 2003.

[16] Y. Koren and D. Harel. Graph drawing by high-dimensional embedding. In *Proc. Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 207–219, 2002.

[17] T. Munzner. H3: Laying out large directed graphs in 3D hyperbolic space. In *Proc. IEEE Symposium on Information Visualization (InfoVis'97)*, pages 2–10, 1997.

[18] O. Niggemann and B. Stein. A meta heuristic for graph drawing. learning the optimal graph-drawing method for clustered graphs. In *AVI 2000: Proc. of the Working Conference on Advanced Visual Interfaces*, pages 286–289, 2000.

[19] A. Noack. An energy model for visual graph clustering. In *Proc. 11th Int. Symp. on Graph Drawing*, volume 2912 of *LNCS*, pages 425–436. Springer-Verlag, 2003.

[20] J. M. Six and I. G. Tollis. A framework for circular drawings of networks. In *Proc. Graph Drawing (GD'99)*, volume 1731 of *LNCS*, pages 107–116. Springer, Berlin, 1999.

[21] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[22] S. T. Teoh and K. Ma. RINGS: A technique for visualizing large hierarchies. In *Proc. Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 268–275, 2002.

[23] C. Walshaw. A multilevel algorithm for force-directed graph drawing. *Journal of Graph Algorithms*, 7(3):253–285, 2003.