TopoLayout: Graph Layout by Topological Features

Daniel Archambault*
University of British Columbia

Tamara Munzner*
University of British Columbia

David Auber†
Université de Bordeaux I

ABSTRACT

We describe TopoLayout, a novel framework to draw undirected graphs based on the topological features they contain. Topological features are detected recursively, and their subgraphs are collapsed into single nodes, forming a graph hierarchy. The final layout is drawn using an appropriate algorithm for each topological feature. A more general goal is to partition the graph into features for which there exist good layout algorithms, so in addition to strictly topological features such as trees, connected components, biconnected components, and clusters, we have a detector function to determine when High-Dimensional Embedder is an appropriate choice for subgraph layout. Our framework is the first multi-level approach to provide a phase for reducing the number of node-edge and edge-edge crossings and a phase to eliminate all node-node overlaps. The runtime and layout visual quality of TopoLayout depend on the number and types of topological features present in the graph. We show experimental results comparing speed and visual quality for TopoLayout against four other multi-level algorithms on ten datasets with a range of connectivities and sizes, including real-world graphs of web sites, social networks, and Internet routers. TopoLayout is frequently faster or has results of higher visual quality, and sometimes, it has both. For example, the router dataset of about 140,000 nodes which contains many large tree subgraphs is drawn an order of magnitude faster with improved visual quality.

nformation Visualization, Graphs and Networks, Graph Visualization

1 INTRODUCTION

TopoLayout is a framework for drawing large, undirected graphs. Our approach is **multi-level**: we decompose the graph into a hierarchy of subgraphs of decreasing size and exploit the hierarchy for layout. Unlike previous multi-level approaches, TopoLayout partitions the graph into **topological features**, which can be laid out with an algorithm tuned for feature topology. The features detected are primarily strict topological features such as trees, connected components, and biconnected components. A more general goal is to partition the graph into features for which there exist good layout algorithms. Thus, we have also developed an algorithm to detect when the very fast High-Dimensional Embedder [19] algorithm, or HDE, will perform well on a subgraph. Previous multi-level algorithms draw the hierarchy of subgraphs beginning with the coarsest graph in the hierarchy to the finest. TopoLayout is the first to draw the subgraph hierarchy using a depth-first, post-order traversal. Traversing the hierarchy in this way allows features at lower levels to determine their screen-space extents before the higher level feature is drawn. Thus, our framework works best if the algorithms used to draw our feature types are **area-aware** or take varying node size into account. TopoLayout is also the first multi-level algorithm to include passes to eliminate node-node overlaps and a pass to reduce the number of node-edge and edge-edge crossings in the layout. These passes contribute to the high visual quality of our layouts with comparable running times to previous reasonable multi-level approaches.

*{archam, tmm}@cs.ubc.ca
†auber@labri.fr

2 PREVIOUS WORK

2.1 Multi-Level Graph Drawing Algorithms

Significant work has been done in developing hierarchical methods for graph drawing to improve algorithm run time with drawings of equal or increased visual quality. The spirit of these multi-level graph layout approaches is to recursively apply a coarsening operator to divide the graph into a hierarchy of coarse graphs that are used to represent a very large input graph. These techniques exploit the property that coarser graphs in the hierarchy are representative of the more detailed ones, but are cheaper to lay out.

In Walshaw [26], an estimate of a solution of the maximal matching problem is used as a coarsening operator to construct the hierarchy. The maximal matching problem selects the largest possible set of edges in the graph such that no two edges are incident to the same node. However, the author acknowledges his technique may not be suitable for densely connected graphs or graphs containing high degree nodes.

Niggemann and Stein [22] describe a multi-level algorithm based on the recursive application of Λ-maximization clustering. For each recursively clustered subgraph, the algorithm constructs a feature vector, which stores statistics about the subgraph, including the number of connected components, biconnected components, and Λ-clusters found. This feature vector is passed to a function which determines the best layout. This function is refined by applying regression learning to a large database of many typical graphs. This best layout method for a typical graph is determined by laying out all the graphs in the database with as many graph drawing algorithms as possible and evaluating a quality metric on each. Although the work does produce some visually convincing results, the largest graph drawn was 1000 nodes. Even though no performance numbers were given, one can assume that the large amounts of precomputation required is a major limitation.

Harel and Koren [15] recursively apply an approximate solution to the k-centres problem, using graph theoretic distance as the ideal distance between two nodes. The k-centres problem groups a set of points into k clusters where the distance between any pair of points in the cluster is minimized. Their algorithm relies on the assumption that the Euclidian distance between two nodes should be proportional to their graph theoretic distance. However, for highly connected graphs, the graph theoretic distance between many pairs of vertices is similar and the chosen clustering could be poor. Similarly, for graphs of low connectivity, such as trees, the assumption that graph theoretic distance should be proportional to Euclidian distance leads to poor layouts.

Gajer *et al.* [11] coarsen by applying a filtration to the node set of the input graph. The filtration operator constructs a maximal subset at each level i using the input graph at level $i-1$ such that the graph theoretic distance between any two nodes of the subset is at least $2^{i-1}-1$. Although the technique works well on graphs of low connectivity, the authors acknowledge that it does not perform well on graphs of higher connectivity.

The ACE algorithm [18] solves for the eigenvectors of the Laplacian matrix to determine a suitable projection of the graph into two, three, or any dimension less than or equal to the number of eigenvectors of the matrix. The eigenvectors are computed by constructing a hierarchy of coarse matrices and computing the eigenvectors of the coarsest matrix. The solution is recursively used as an estimate for the eigenvectors one level down until the eigenvectors of the original graph have been computed. However, a recent empirical evaluation of fast graph drawing algorithms [14], demonstrates that it does not perform well on many types of graphs.

The Fast Multipole Multilevel Method, or FM^3, algorithm [13] is the first multi-level algorithm for general graphs with a provable worst case asymptotic runtime of $O(N \log N + E)$. The graph is partitioned into subgraphs called solar systems. Sun nodes are the central node of the solar system. Planet nodes are the nodes immediately adjacent to a sun node. Moon nodes are the set of nodes immediately adjacent to a planet node. The solar systems are contracted down to single nodes and the process is repeated to create a hierarchy. They show a fixed fraction of nodes and edges are present in each solar system, proving the hierarchy is balanced. Using this fact, they are able to prove that the final graph layout can be obtained in $O(N \log N + E)$ time. A subsequent evaluation of FM^3 convincingly demonstrates that FM^3 yields higher visual quality results than previous work [14]. Although TopoLayout cannot be proven to be asymptotically faster than FM^3, we will show that, in terms of speed and visual quality, it empirically outperforms FM^3 on many types of graphs.

The work of Six and Tollis [24] is perhaps closest in spirit to our own. Although the method is not technically multi-level because the hierarchy is not recursively constructed, it does share some properties of multi-level techniques; they decompose the graph into biconnected graphs and lay out the tree of biconnected components using a radial tree layout algorithm which is area-aware. The individual biconnected components are drawn using a circular layout. However, the only topological feature type detected is biconnected components, whereas TopoLayout handles many feature types.

We also describe preliminary work on TopoLayout in a recent poster [1].

2.2 High-Dimensional Embedder (HDE)

In addition to strict topological features, TopoLayout detects when the **High-Dimensional Embedder**, or **HDE** algorithm [19] of Harel and Koren, is an appropriate choice. This approach is related to a rich family of mathematical approaches which have been explored as solutions to problems ranging from flattening curved surfaces [23] to texture mapping in computer graphics [27]. These algorithms start by selecting a subset of d points called pivots and compute the pairwise geodesic or graph theoretic distance between the pivots and all other points on the surface. Each pivot corresponds to a dimension, and the graph theoretic distance between a pivot and all other points defines a position for each point in a d-dimensional space. The point set is centred, and principal component analysis (PCA) or multi-dimensional scaling (MDS) maps the d-dimensional embedding down to a two or three dimensions.

In HDE, the first pivot of the graph is selected randomly. The graph theoretic distance between the first pivot and all other nodes in the graph is computed using Dijkstra's algorithm[1]. For the remaining $m-1$ pivots, the node with furthest graph theoretic distance from the pivot is selected in order to maximize variance on each axis. Once the graph is embedded in the d-dimensional space, PCA is used to map the graph down into two dimensions. A typical value for d is 50 and the algorithm has an asymptotic runtime of $O(d(N \log N + E))$.

[1] The graph theoretic distance can be computed using either Dijkstra's algorithm or breadth-first search; here we use Dijkstra's algorithm because we need weighted HDE to handle graphs of positive, unequal edge weights.

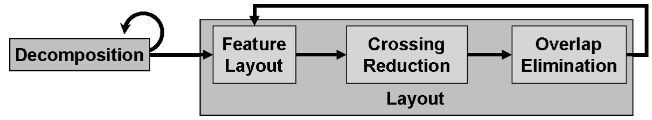


Figure 1: TopoLayout algorithm phases.

3 ALGORITHM

The TopoLayout framework consists of four main phases as shown in Figure 1. The **decomposition** phase is the same as the coarsening operator in other multi-level techniques. It is recursively called on the input graph, creating our hierarchy and identifying the feature type of the subgraph. The **feature layout** phase draws the subgraph using an appropriate algorithm for the feature type. The **crossing reduction** phase reduces, but does not completely eliminate, the number of node-edge and edge-edge crossings in the subgraph by rotating nodes. Finally, the **overlap elimination** phase ensures that no two nodes overlap in the final drawing. The last three algorithms are applied to each subgraph in the post-order traversal and then recursively are computed for higher levels of the hierarchy.

Many of the algorithms used in these phases are directly drawn from previous work, some are slight modifications of previous work, and some are novel algorithms of our own. Similar work on reducing node-edge and edge-edge crossings has been presented, but we present a new algorithm for this problem in Section 3.2.2. All algorithms used directly from previous work or those with small modifications are described in these sections and cited.

3.1 Decomposition

The decomposition phase consists of a series of topological feature detection algorithms, which are applied to the input graph. Upon detection of a topological feature, the subgraph of the feature is collapsed into a single node. This process forms a hierarchy, where a node at level i containing a subgraph is a parent and the nodes of the subgraph it contains at level $i+1$ are its children. Nodes which contain a subgraphs are known as **meta-nodes**. We call the nodes of the input graph **leaves** of the hierarchy as they are the only nodes in the subgraph hierarchy which are not meta-nodes and terminate all paths in the subgraph hierarchy. Note that the computed hierarchy is rarely balanced and leaves can occur at any level. During the construction of a meta-node n, for any edge e adjacent to a node contained in the subgraph of n and a node outside the subgraph of n, we create a **meta-edge** which connects the meta-node to the other node adjacent to e. Meta-edges contain a list of pointers to the edges in the input graph which they represent. Figure 2 shows an example hierarchy created by our decomposition phase. Meta-nodes are the rectangles in the diagram and their subgraphs are the set of nodes contained within the box. The nodes are coloured by topology type. This same colour encoding is used for all drawings produced by TopoLayout for the remainder of this paper.

Figure 3 describes the decomposition algorithm in detail and the topological features we detect. The boxes in the diagram are coloured using the same scheme introduced in Figure 2. **Trees** are subgraphs without cycles. A **connected component** is a subgraph where there exists a path between any pair of nodes in it. A **biconnected component** is a subgraph where the removal of any node or edge within the subgraph does not disconnect it into two or more connected components. A **cluster** is a subgraph formed by some clustering algorithm. In our implementation, we use the strength metric [3] for clustering. We then determine if **HDE** is a suitable algorithm to lay out the subgraph. We note HDE components are not topological features, but HDE component detection is a first step in selecting appropriate graph layout algorithms for subgraphs

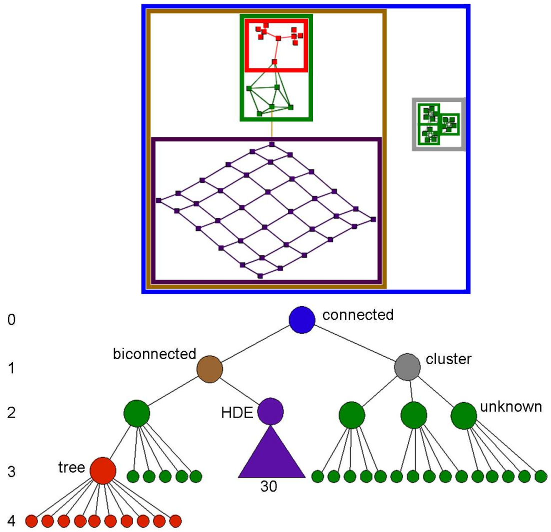


Figure 2: Subgraph hierarchy after decomposition, with topology encoded by colour. **Top**: Layout annotated with bounding boxes to show hierarchy structure: meta-nodes encompass the subgraphs of their children. **Bottom**: Diagram of subgraph hierarchy, with levels enumerated and nodes labeled by feature type. The hierarchy is not necessarily balanced: leaves can occur at any level.

in our hierarchy. Finally, if the decomposition phase cannot identify the topology of the subgraph, it is labeled **unknown**.

The resulting hierarchy, and therefore the resulting layout, can change with the order in which these detection algorithms are applied. Our order is based on the following logic. Connected components of the graph should be detected first, since if there are multiple components, we can lay them out independently. Trees need to be detected before biconnected components because the removal of any edge or node from a tree disconnects the tree into two components. Therefore, a biconnected algorithm run on a tree would fragment all trees into disjoint nodes and edges. Before we further decompose the graph using strength clustering, we check to see if HDE is an appropriate algorithm for layout. Finally, cluster detection provides a reasonable decomposition to partition the graph into highly connected subgraphs as a last step when more meaningful topological features cannot be found.

3.1.1 Topological Feature Detection Algorithms

We detect connected components using a series of depth-first searches to compute spanning trees for each component. We refer the reader to the Baase and Van Gelder textbook [4] for details of this standard algorithm for connected component detection, which runs in $O(N+E)$.

We detect trees by finding the first cycle in the graph and selecting a node n on that cycle. If a cycle is not found, the entire graph is a tree. Otherwise, starting at n, we perform a depth-first search on the entire graph. When we visit a node of degree one, we remove it and continue the depth-first search. The algorithm removes all nodes of degree one it encounters until there are no more, or when a maximal tree is detected. The time required for tree detection is therefore $O(N+E)$.

A good description of a standard biconnected component detection algorithm is given by Baase and Van Gelder [4]. Biconnected components are detected in the graph 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 takes at most $O(N+E)$ time.

We compute clusters using the strength metric [3]. The strength metric partitions the graph into subgraphs by the number of 3- and 4-cycles shared by the nodes of the subgraph. For each edge subtending nodes u and v, we partition nodes adjacent to u and v into three sets: $M(u)$, those adjacent to u; $M(v)$, those adjacent to v; and $W(u,v)$, those adjacent to both u and v. The total number of 3-cycles is the number of elements in $W(u,v)$. We determine the number of 4-cycles by checking for the existence an edge between an element of $W(u,v)$ and an element of either $M(u)$ or $M(v)$. These edges can be computed in $O(r)$ time using a hash table, where r is the maximum degree of a vertex in the graph. We can thus detect clusters in $O(rE)$ time.

Finally, to determine if HDE is a suitable layout algorithm for the subgraph, we lay out the subgraph with HDE and compute the unweighted Kruskal Stress-1 function [20] on a random subset of $\sqrt{N}$ nodes taken from the graph. Because HDE is a very fast graph drawing algorithm, we can afford to perform the layout and evaluate the stress. When evaluating the stress of the resulting layout, we exclude the pivots chosen by HDE because the stress was inherently minimized for them by the algorithm. The Kruskal Stress-1 is frequently used in MDS to evaluate how faithfully the low-dimensional embedding of a set of points represents the high-dimensional distances between them. It determines the normalized disparity between the high-dimensional representation of the point set and the low-dimensional representation. Kruskal Stress-1 produces a value between $[0,1]$ with zero corresponding to no disparity between the two-dimensional drawing and the high-dimensional space, and one corresponding to maximum disparity. Although HDE uses PCA to map the d-dimensional space down to two dimensions, Kruskal Stress-1 is still applicable as a good layout will place nodes of small shortest path distance close together and nodes large shortest path distance far apart. Thus, good HDE layouts should have low stress. The time required to compute the stress is $O((N\log N+E)\sqrt{N})$ as Dijkstra's shortest path algorithm is computed for each of the $\sqrt{N}$ nodes of the subset. This stress computation is computed a small constant number of times and if one of those stresses is below a threshold of 0.2, the layout is accepted.

3.2 Layout

During the layout phase of level i of a hierarchy, the features at level $i+1$ contained by all the meta-nodes at level i must be laid out first to determine the screen-space bounds of the meta-node. The required screen space of the leaves at level i is already known: the original size of the node. The layout stage, shown in Algorithm 1, draws the topological feature at level i using an appropriate layout algorithm, rotates meta-nodes of the hierarchy to reduce crossings, and eliminates all node-node overlaps in the subgraph.

Algorithm 1 Pseudocode for the feature layout phase.

layout (subgraph s)
  for all meta-nodes $c \in s$ **do**
    c.size ←**boundingBox** (**layout** (c.subgraph));
  layOutFeature (s);
  reduceCrossings (s);
  eliminateOverlaps (s);

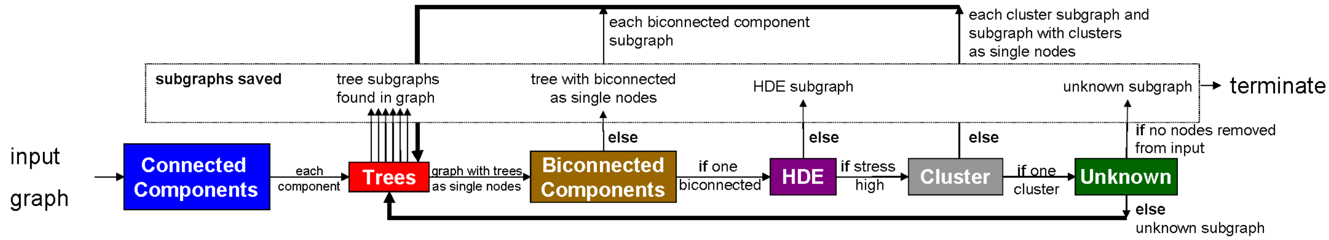


Figure 3: Decomposition phase for TopoLayout. Detection algorithms in coloured boxes with the same colouring scheme used for the features in Figure 2. If a clause on a horizontal is true, we transition along the arrow. Otherwise, we follow the vertical arrow to save some subgraphs and recursively decompose others.

3.2.1 Feature Layout

The initial layout of the topological features in the graph depends on the detected topological type. We employ four types of layout algorithms: tree, circular, HDE, and force-directed.

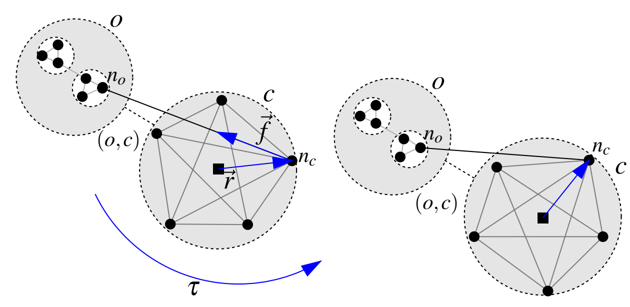
Area-aware **tree layout** algorithms are used for the tree and biconnected component topological types. Clearly, tree layout algorithms are appropriate for trees, but the reason to use them to draw biconnected components is less obvious. For a set of biconnected components residing at level i with their collapsed subgraphs at level $i+1$, the topology of the subgraph at level i is a tree; if it were not, there would be a cycle at level i and all subgraphs on that cycle would be merged into a single biconnected component at level $i+1$. If the removal of an edge created two biconnected components, the edge appears as an edge in the tree at level i. If the removal of a node created two biconnected components, we use one of the methods suggested by Six and Tollis [24] and place the node between the two components. TopoLayout can use any tree layout algorithm that is area-aware to draw these trees. We use the bubble tree algorithm [12] for trees of low depth and high branching factor and an area-aware version of the Walker algorithm [5] for all other trees. The bubble tree algorithm requires $O(N \log N)$ time while the version of the Walker algorithm runs in $O(N)$ time.

We use an area-aware **circular layout** algorithm to highlight complete graphs. Circular layout consists of simply placing the nodes of the graph around a circle, so area-aware circular layout is a straightforward adaption. Although circular layouts yield low visual quality drawings for general graphs because they have many crossings, they are a good choice for complete graphs; the symmetry of all lines crossing and the feature-based color coding leads to the visual pop-out of cliques. The algorithm runs in $O(N)$ time.

When appropriate, we use **area-aware HDE** [19] to lay out unknown components that preform well during detection. Area-aware HDE is simply normal HDE with weighted edges. The weight of each edge is set to the maximum radius of the adjacent nodes with a minimum weight of one.

We use **area-aware GEM** for all other cases of clusters and unknown components. Area-aware GEM is a minor modification of the GEM Frick algorithm [9] where nodes are considered charges and the edges are considered springs. The system is placed in an initial configuration, often random, and is released until it reaches an equilibrium. Oscillations and rotations about equally optimal positions are dampened. We use an area-aware version of GEM, which is similar to the algorithms developed by Harel and Koren [16] who adapted Fruchterman-Reingold [10], Kamada-Kawai [17], and combinations of these algorithms.

The forces for area-aware GEM can be defined for a pair of nodes n_i and n_j. Let r_i and r_j be the the radii of the bounding circles of these nodes respectively. Let p_i and p_j be their positions, and let l be some ideal spring length for the distance between the boundaries of the two nodes. The GEM forces that a node n_j exerts on a node n_i are:



(a) (b)

Figure 4: Reducing crossings with torque. **(a)** Computing the torsional force τ on c exerted by the edge (n_o, n_c). **(b)** Applying τ results to rotate c. Dashed nodes and edges are meta-nodes and edges. Solid nodes are leaves. The square box is the centre of node c.

$$f_{\text{repulsive}}(n_i, n_j) = \frac{l + \lceil \mathbf{r_i} + \mathbf{r_j} \rceil}{\|p_i - p_j\|^2}(p_i - p_j) \tag{1}$$

$$f_{\text{attractive}}(n_i, n_j) = \frac{\|p_i - p_j\|^2}{l + \lceil \mathbf{r_i} + \mathbf{r_j} \rceil}(p_j - p_i) \tag{2}$$

The bold terms in (1) and (2) are the terms we added to make GEM area-aware. The ceiling of the sum of the radii is taken so that the forces are still computed purely with integer arithmetic. Oscillation and rotation control in the algorithm is the same. The complexity of the algorithm remains $O(N^3)$.

3.2.2 Crossing Reduction

We introduce an algorithm to reduce the number of edge-edge crossings and node-edge crossings in our drawing. Our heuristic is to rotate meta-nodes in our hierarchy to reduce crossings of edges in the original graph connecting nodes in different subgraphs of the hierarchy as shown in Figure 4. The heuristic does not guarantee an elimination of node-edge or edge-edge crossings, but it reduces the number of them in most cases and shortens edge length between subgraphs as well. Our approach is similar to that of Symeonidis and Tollis [25] who provide a solution to this problem by minimizing what they call inter-group crossings. In their approach, an energy function is minimized to apply a good rotation to their circular drawings to reduce the number of crossings. This approach is analogous to Kamada-Kawai [17] in graph layout. In contrast, our approach is similar to GEM [9] and includes oscillation control.

Let o and c be meta-nodes in a subgraph at level i of our graph hierarchy. Let n_o and n_c be leaves in our graph hierarchy. We use the positions of n_o and n_c in the coordinate frame in the subgraph at level i to compute the torque τ. The nodes of n_o and n_c are not necessarily at level $i+1$ and can be nested in several levels of meta-nodes, each with their own relative coordinate frames. For the moment, we assume the location of the nodes n_o and n_c is known in the coordinate frame of the subgraph at level i and show later how these positions can be computed efficiently. The torque computed is physically inspired, but is not physically realistic. Let the force vector $\vec{f}$ be a unit force along the edge (n_o, n_c). Let $\vec{r}$ be the radius vector from the centre of node c to the node n_c. The function $\mathrm{sg}(\vec{x})$ returns the sign of the normal perpendicular to the embedding plane. The torque exerted by (n_o, n_c) on c is given by Equation (3).

$$\tau = \frac{\pi}{2}\mathrm{sg}(\vec{r}\times\vec{f})(\vec{r}\cdot\vec{f}) \tag{3}$$

Analogous to that of force-directed graph drawing techniques, our solution to the problem is incremental. The average value of τ is computed for all edges in the list of edges contained in the meta-edge (o, c). The process is repeated, computing an average τ for each meta-node in the subgraph containing o and c, using their incident meta-edges. Once the average τ is computed for all meta-nodes in the subgraph, it is applied to the cumulative rotation of each meta-node.

Meta-nodes can oscillate around equally good orientations. Our approach to dampening oscillations is similar to that of GEM [9]. We store the torque for each meta-node applied during the previous iteration and compare it with the torque computed during the current iteration. If the signs of the torque in the two iterations are opposite, we are oscillating around an optimal orientation, and a damping factor is applied. Currently, this factor is the fraction of completed iterations to the N_i iterations which will be executed, where N_i is the number of meta-nodes in the subgraph at level i.

Computing the positions of the n_o and n_c nodes in the coordinate frame of the subgraph at level i is relatively straightforward if every node in the graph hierarchy has a pointer to meta-node which contains it. This information can be constructed in the decomposition phase with no asymptotic runtime penalty when we construct meta nodes. Each meta-edge has a list of edges it represents, so each n_o and n_c involved in a torque computation can be determined in constant time. We traverse the hierarchy up to the subgraph at level i composing translations and rotations to determine the positions of n_o and n_c in the subgraph at level i. If n_o or n_c is at a depth of $i+L$, this traversal takes $O(L)$ time. Since each edge is involved in at most one torque computation and N_i iterations of torque are executed, the overall asymptotic complexity of the crossing reduction phase is $O(LN_iE)$.

3.2.3 Overlap Elimination

In TopoLayout, neither area-aware GEM nor HDE provides a guarantee of no node-node overlaps. The crossing reduction phase may also introduce node overlaps between meta-nodes and other nodes in each subgraph of the hierarchy. To ensure that pairs of nodes do not overlap in our final layout, we perform a pass to test for and reduce or eliminate these overlaps.

We experimented with several algorithms to reduce or eliminate node overlaps in the drawing. In all cases, we tried overlap reduction two ways: separately for each subgraph of the hierarchy, or a single pass on the entire final drawing after TopoLayout had executed all other phases. We found that the former approach was best, because a single pass on the final drawing causes overlap of topological features.

First, we tested the naive approach of considering every pair of nodes to determine the set of overlaps. If two nodes overlapped, they were shrunk down in size until no overlap was present. Although this $O(N^2)$ method was slow, it does guarantee a drawing free of node-node overlaps and produced drawings of high visual quality for many types of graphs.

Algorithm	Complexity
Detection	
Tree	$O(N_i+E_i)$
Biconnected Component	$O(N_i+E_i)$
Connected Component	$O(N_i+E_i)$
HDE	$O((N_i \log N_i + E_i)\sqrt{N_i})$
Cluster	$O(rE_i)$
Initial Layout	
Bubble Tree	$O(N_i \log N_i)$
Walker Tree	$O(N_i)$
Circular	$O(N_i)$
Area-Aware GEM	$O(N_i^3)$
HDE	$O(d(N_i \log N_i + E_i))$
Refinement	
Crossing Reduction	$O(LN_iE)$
Overlap Elimination	$O(N_i \log N_i)$

Figure 5: Time complexity of TopoLayout framework components, for each hierarchical level.

We also implemented the Cluster Buster algorithm of Lyons *et al.* [21], which computes the Voronoi diagram of the node set and iteratively pulls the nodes towards the centroid of each Voronoi cell. For a constant number of iterations, the algorithm runs in $O(N \log N)$ time. Unfortunately, this method does not guarantee no node overlaps in the final drawing, and the results were usually of low visual quality.

We obtained the best results from implementing the fast node overlap removal algorithm without Lagrange multipliers [7], which is discussed in detail in Dwyer *et al*'s technical report [8]. In this work, two, separate passes along the x-axis and the y-axis eliminate all node overlaps in the graph. The algorithm constructs a weighted, directed constraint graph along each dimension and uses quadratic programming to minimize node displacement. Assuming that each node in the graph overlaps with a constant number of nodes, the algorithm is $O(N \log N)$. This method guarantees no overlaps in the final drawing and was applied to every subgraph of the hierarchy to produce the results in Section 5.

The overlap elimination phase is always executed on graphs drawn with HDE and area-aware GEM, since we cannot guarantee the absence of overlaps in drawings generated from these algorithms. Overlap elimination is only executed on other topological features if they contain meta-nodes, because the crossing reduction phase can introduce overlaps. As the fast overlap removal algorithm only considers axis aligned nodes, the axis aligned bounding box of the rotated meta-node is computed.

4 Algorithm Complexity

Figure 5 shows the time complexity of the algorithms we use in TopoLayout. We report the number of operations performed on each subgraph of the hierarchy: N_i is the number of nodes in a subgraph, and E_i is the number of edges in a subgraph at level i. The maximum degree of a node in the subgraph at level i is r. The value of d is the dimensionality of the high-dimensional space of the HDE algorithm, which is typically fifty. The value of L is the number of levels we must traverse up the hierarchy to compute the level i positions of n_o and n_c when computing torques.

5 Experiment

We implemented the TopoLayout framework on top of the Tulip [2] graph visualization system and have tested it against other multi-level algorithms on datasets with a range of connectivities and sizes. All benchmarks were run on a 3.0GHz Pentium IV with 3.0GB of memory running SuSE Linux with a 2.6.5-7.151 kernel. The majority of the data used for testing was taken from Hachul and Jünger's empirical study [14] of graph drawing algorithms and Figures 6 and 7 demonstrate that we have reproduced the results of their work. In fact, even the running times are nearly the same as our hardware setup was similar. In these figures, every row is the same dataset and every column is a layout algorithm. The name and size of the dataset for each row is in the upper left hand corner of the leftmost column. The time taken to lay out each dataset with a particular algorithm appears in the upper right hand corner of each table entry. For space reasons, a representative subset of the graphs and algorithms used in this evaluation were chosen. The code for GRIP[2], ACE[3], and HDE[4], was available online and was incorporated into the Tulip framework. Stephan Hachul kindly supplied the FM^3 code, which was also incorporated into Tulip for testing. Harel and Koren's multi-level approach [15] was not tested. The source code for this implementation was unavailable and restricted graph sizes of less than ten thousand nodes due to quadratic size memory requirements. As our observed running times and visual quality results were very similar to those in the empirical study [14], one can refer to their results for a comparison. We allowed TopoLayout to colour topological features in the graph, using the scheme defined in Section 3.1. Since the other graph drawing algorithms do not detect topological features automatically, the comparison is fair and demonstrates another advantage of our approach.

From the datasets in the study [14], we chose one of their *real world* graphs, the B-size graphs for each of their *challenging artificial* graphs classes, and one *challenging real world* graph. When comparing our results to their study, note that smaller `snowflake_A` and `spider_A` drawings were shown, whereas we are providing drawings for the larger `snowflake_B` and `spider_B`. All other drawings we provide match those in the study. In addition to the datasets present in the study, we added four of our own. We show the drawings obtained from the experiment in Figures 6 and 7.

`Crack` is a standard graph drawing dataset part of the Walshaw Graph Partition Archive[5]. It was the *real world* graph used in the study. The `6-ary`, `snowflake_B`, `spider_B`, and `flower_B` datasets are one size of each of the *challenging artificial* graphs supplied by Stephen Hachul. The `6-ary` tree dataset is simply a 6-ary tree of depth five. `Snowflake` is a tree of very high variance in degree. `Spider` has a subset of nodes S which consists of 25% of the nodes in the graph. The elements of S are each connected to twelve unique members of S. The remaining nodes are rooted at a single node along eight paths of equal length. `Flower` has a relatively high density. It consists of joining six circular chains of the graph K_{30}, a complete graph of thirty nodes, at a single instance of K_{30}. `LaBRI` is the hyperlink structure of `www.labri.fr`. `IMDB Subset` is a subset of the Internet Movie Database[6]. Nodes in this graph are actors and edges are present if two actors appeared in the same movie. `Add32` is a graph which models the hardware structure of a thirty two bit adder from the Walshaw Graph Partition Archive. It was a *challenging real world* dataset used from the study. `Bico_Walshaw` is fourteen datasets from the Walshaw Graph Partition Archive connected by thirteen single edges into one component. `Routers` is a near-spanning tree with a few cycles of the major routers on the Internet backbone from the Internet Mapping Project [6] taken in February 2002.

[2]`www.cs.arizona.edu/~kobourov/GRIP`
[3]`research.att.com/~yehuda/programs/ace.zip`
[4]`research.att.com/~yehuda/programs/embedder.zip`
[5]`staffweb.cms.gre.ac.uk/~c.walshaw/partition`
[6]`www.imdb.com`

6 Discussion

One strength of TopoLayout is that the running time performance is proportional to the amount of a particular topological feature present in the graph. For the most part, the topological feature detection algorithms are quick enough to determine if something is definitely not present in the graph. As a result, we have an algorithm whose performance is sensitive to topology type.

Figures 6 and 7 show another advantage of our approach: by choosing a layout algorithm for a feature that creates a characteristic pattern, we get visual pop-out effects that allows these features to be easily noticed. For example, we can easily pick out the node and edge biconnectivity in `LaBRI`, the cycles and tree structures in `routers`, and the cliques in `IMDB Subset`.

For nearly all datasets, except `crack`, neither ACE nor HDE is able to produce layouts of high visual quality, because they place many nodes at the same location. It has been noted in Hachul's study [14] that these algebraic methods do not work well on graphs with many biconnected components. For this reason, the remainder of the discussion will focus on the performance of GRIP, FM^3, and TopoLayout.

On `crack`, TopoLayout is able to outperform FM^3, but is slower than GRIP. As expected from the literature, all the algorithms produced pleasing drawings.

TopoLayout is able to outperform GRIP and FM^3 in terms of running time on `6-ary` and `snowflake` and it outperforms them in terms of visual quality as well. As TopoLayout detects trees, it is able to visualize the global structure of `6-ary` and `snowflake` more effectively. For `snowflake`, the FM^3 algorithm actually hides part of the graph's global structure, because clumps the single nodes attached to the root near the centre of the drawing as shown in the inset. GRIP can show part of the structure around its root node as shown in its inset. Topolayout pulls them out into the fan structure visible in the lower left of the drawing.

TopoLayout has a slower run time on FM^3 and GRIP on `spider`. The reason is that it runs $O(N^3)$ area-aware GEM on the highly connected head component. The visual quality of the layout is similar for FM^3, GRIP, and TopoLayout in the head region of the spider shown in the inset, but the eight fold symmetry of the legs is difficult to see.

TopoLayout is slower than FM^3 and GRIP on `flower`, because the strength metric [3] has slow performance when the connectivity of the graph approaches near-complete. The drawing of `flower` produced by TopoLayout has better information density; we can see the individual cliques in each loop at a single scale, whereas it is less true for the GRIP and FM^3 drawings. However, the topology of the loops are better drawn by FM^3. The structure of the cliques is more apparent using TopoLayout because they are detected and drawn using circular layout as shown in the inset. It is difficult to see the structure of the cliques in the FM^3 and GRIP drawings because many nodes overlap.

On `LaBRI`, the running time of TopoLayout is of the same order as that of GRIP and FM^3. The drawings are of similar visual quality, but the trees and biconnected components in the dataset are displayed more clearly in the TopoLayout drawing.

There are many cliques in `IMDB Subset` and all three algorithms are able to separate them out in their drawings. As TopoLayout is able to detect complete graphs and draw them using a circular layout, the topology of these cliques is made immediately apparent whereas it is hidden by GRIP and FM^3. The areas of biconnectivity are visible in all three drawings and the running times are of the same order.

For `add32`, the drawing speed of GRIP is considerably faster than that of FM^3 and TopoLayout whose running times are on the same order. As `add32` describes a 32-bit adder, it is no surprise that it contains many biconnected components and has a tree-like shape.

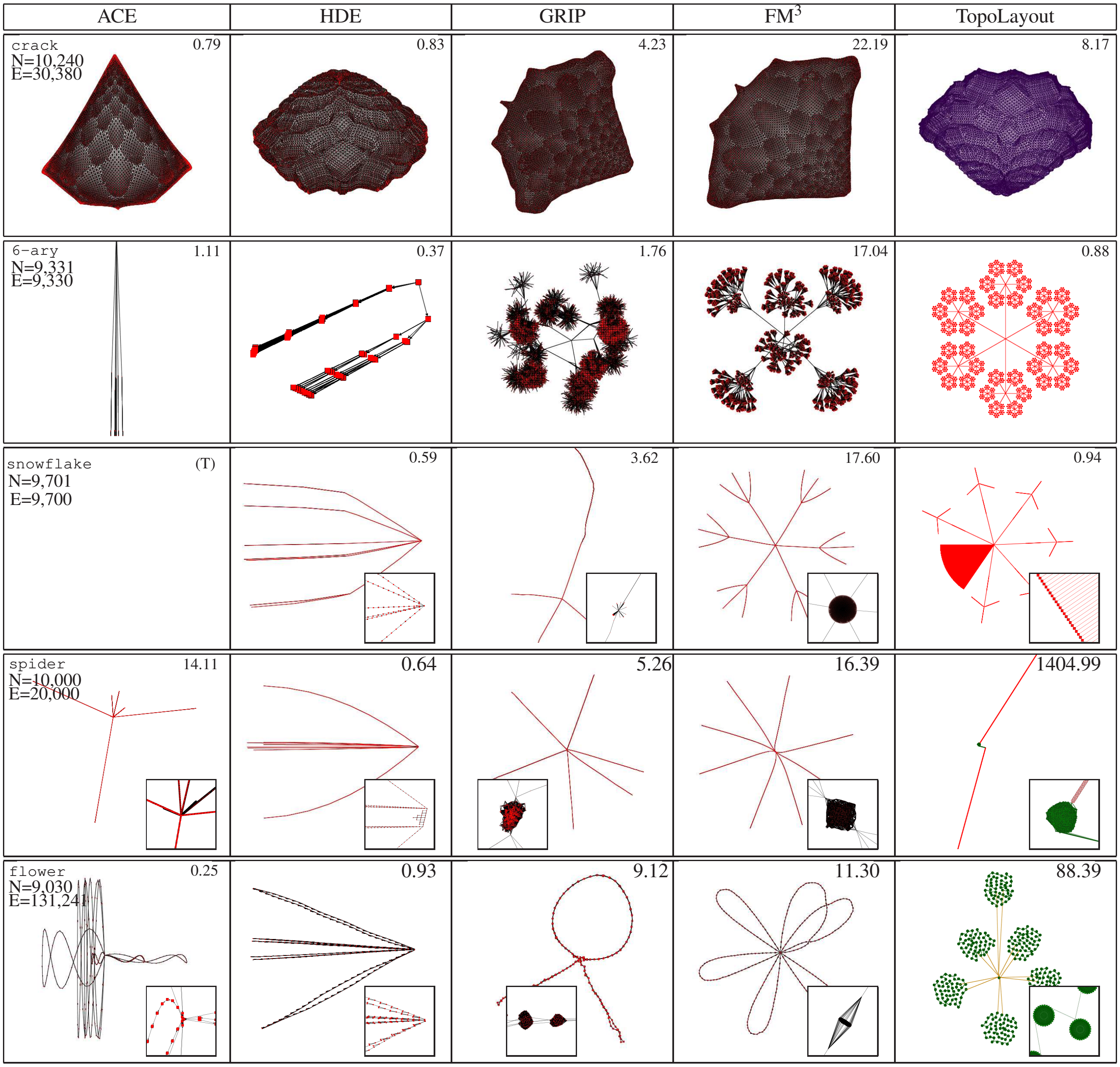


Figure 6: Layouts of several datasets using ACE, HDE, GRIP, FM^3, and TopoLayout for several datasets described in Section 5. For all rows, blank squares indicate no drawing produced. Dataset name, number of nodes, and number of edges appear in the top left hand corner of the leftmost column. Times in seconds, or reasons for no drawing, appear in the upper right corner of each entry. (T) indicates no drawing produced in four hours of program execution.

The GRIP and FM^3 algorithms are able to visualize this large-scale shape, but with TopoLayout we are also able to see some internal structure.

TopoLayout has a running time of the same order as FM^3 on `bico_walshaw` and GRIP is unable to produce a drawing. The drawings produced by FM^3 and TopoLayout are of similar visual quality.

TopoLayout draws `routers` an order of magnitude faster than FM^3, while GRIP is unable to produce a drawing. In fact, the running time of TopoLayout is similar to that of HDE. All algorithms are able to draw the cycles present in the dataset, but only TopoLayout is able to visualize the tree structures properly.

7 Future Work

Two obvious ways to improve our framework would be to have better detection and layout algorithms for the existing set of topological features, and to add support for new feature types. Figure 5 shows that the most expensive algorithm in the decomposition phase is cluster detection, and it is reflected in the slower running times for the flower graphs. For the feature layout phase, the most expensive algorithm is the $O(N^3)$ area-aware GEM algorithm, and we see it reflected in the running time of `spider` as the large head component is laid out using this algorithm. Adding more specialized detection algorithms would not only improve the visual quality of the layout, but could also improve performance if they led to a

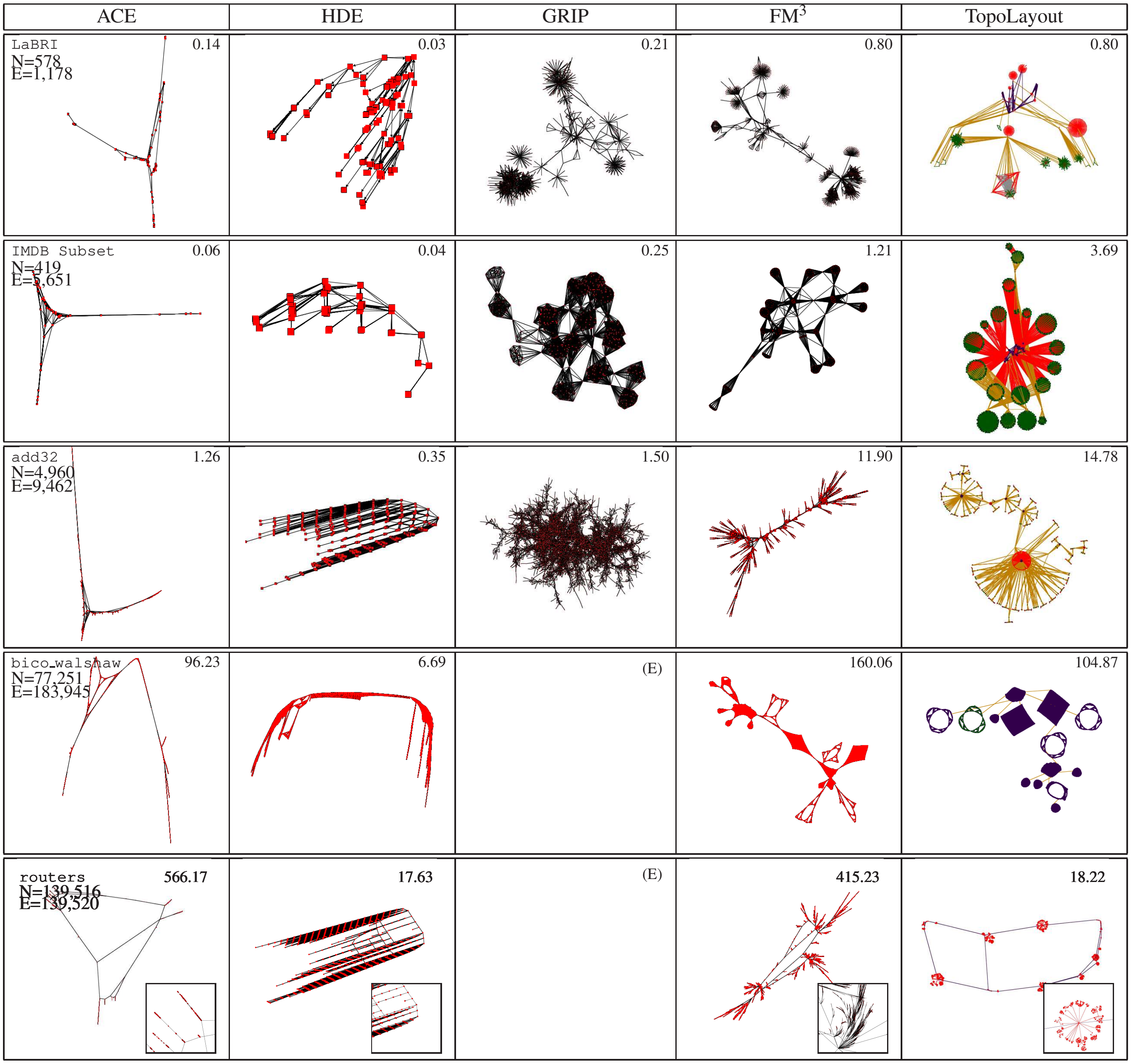


Figure 7: Layouts of several datasets using ACE, HDE, GRIP, FM^3, and TopoLayout for several datasets described in Section 5. For all rows, blank squares indicate no drawing produced. Dataset name, number of nodes, and number of edges appear in the top left hand corner of the leftmost column. Times in seconds, or reasons for no drawing, appear in the upper right corner of each entry. (E) indicates no drawing produced because of an error in the executable.

smaller set of nodes being passed to the final cluster detector. In particular, adding different cluster detection methods is an obvious next step.

For laying out unknown components, it would be interesting to use an area-aware version of FM^3 instead of area-aware GEM for its fast running time and high visual quality on many types of graphs. Since FM^3 is based on a spring embedder, adapting it to be ara-aware should be feasible. With area-aware FM^3, we would be able to significantly improve the slow run time of TopoLayout on `spider`, where the spring embedder takes over 99% of the time. However, the visual quality of the drawing would most likely remain the same. The visual quality of `flower` would be significantly improved, and we would most likely be able to see the six fold symmetry of the loops. However, we would probably not see a great improvement in the run time, since cluster detection forms a significant fraction of it.

Improving area-aware HDE and creating area-aware variants of other typical graph drawing algorithms is a topic for future research. Currently, we make HDE area-aware by using the obvious approach of weighting the edges proportionally to adjacent nodes size, which works well on graphs with uniform node sizes. However, information density suffers with nonuniform node sizes, as occurs in `routers`. We conjecture that a more sophisticated approach would result in better information density. We would also like to improve the precision of our HDE detection algorithm.

Although TopoLayout handles many types of graphs better than previous work, it still does not produce high-quality layouts for all graphs in the information visualization domain. If none of the topological feature types we detect occur in the graph, results are poor. In this case, we simply have a hierarchy of clusters and unknown components which are drawn using area-aware force-directed placement. However, our framework allows for the addition of detection functions, and incorporating better layout algorithms for specific features will improve its performance. Detection and layout of approximate large-scale topological structures such as quasi-cliques and quasi-trees, rather than exact topological features, would be an interesting next step.

8 Conclusion

We have presented TopoLayout, a novel framework for drawing large, undirected graphs. Unlike previous multi-level approaches, TopoLayout partitions the graph into topological features, which can be laid out using an algorithm tuned for their topology. In addition to topological features, we have developed an algorithm to detect when HDE will perform well on a subgraph. In contrast to previous multi-level algorithms, TopoLayout is the first to draw the subgraph hierarchy using a depth-first, post-order traversal. Traversing the hierarchy in this way allows features at lower levels to determine their screen-space extents before the higher level feature is drawn. TopoLayout is also the first multi-level algorithm to provide passes to eliminate node-node overlaps and a pass to reduce the number of node-edge and edge-edge crossings in the layout. It guarantees a final layout of no node-node overlaps with comparable running times to reasonable multi-level approaches, where runtime and layout visual quality depends on the number and types of topological features present in the graph. The experimental results comparing TopoLayout to four other multi-level approaches on a range of datasets show that TopoLayout is frequently faster or has results of higher visual quality, and in some cases, it has both.

Acknowledgment

We thank Stephan Hachul for supplying the FM^3 code and the datasets used in his empirical study. Partial funding was provided by the ACI Jeunes Chercheurs *Cube de Données: Construction et Navigation Interactive* and INRIA IPARLA. We thank Ciaran Llachlan Leavitt for help in editing this paper.

References

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

[2] 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.

[3] D. Auber, Y. Chiricota, F. Jourdan, and G. Melancon. Multiscale visualization of small world networks. In *Proc. IEEE Symposium on Information Visualization (InfoVis'03)*, pages 75–81, 2003.

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

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

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

[7] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. In *Proc. 13th Int. Symp. on Graph Drawing*. Springer-Verlag, 2005.

[8] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. Technical report, School of Comp. Science & Soft. Eng., Monash University, Australia, August 2005.

[9] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proc. Graph Drawing (GD'94)*, volume 894 of *LNCS*, pages 388–403, 1995.

[10] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, November 1991.

[11] P. Gajer, M. Goodrich, and S. Kobourov. A multi-dimensional approach to force-directed layouts of large graphs. In *Proc. Graph Drawing (GD'00)*, volume 1984 of *LNCS*, pages 211–221. Springer, Berlin, 2001.

[12] 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.

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

[14] 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.

[15] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. In *Proc. Graph Drawing (GD'00)*, volume 1984 of *LNCS*, pages 183–196. Springer, Berlin, 2001.

[16] D. Harel and Y. Koren. Drawing graphs with non-uniform vertices. In *Proc. Working Conference on Advanced Visual Interfaces (AVI'02)*, pages 157–166, 2002.

[17] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

[18] Y. Koren, L. Carmel, and D. Harel. ACE: A fast multiscale eigenvectors computation for drawing huge graphs. In *Proc. IEEE Symposium on Information Visualization (InfoVis'02)*, pages 137–144, 2002.

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

[20] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[21] K. A. Lyons, H. Meijer, and D. Rappaport. Algorithms for cluster busting in anchored graph drawing. *Journal of Graph Algorithms and Applications*, 2(1), 1998.

[22] 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.

[23] E. L. Schwartz, A. Shaw, and E. Wolfson. A numerical solution to the generalized mapmaker's problem: Flattening nonconvex polyhedral surfaces. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(9):1005–1008, September 1989.

[24] 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.

[25] A. Symeonidis and I. G. Tollis. Visualization of biological information with circular drawings. In *Intl Symposium on Medical Data Analysis (ISBMDA)*, pages 468–478, 2004.

[26] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proc. Graph Drawing (GD'00)*, volume 1984 of *LNCS*, pages 171–182. Springer, Berlin, 2001.

[27] G. Zigelman, R. Kimmel, and N. Kiryati. Texture mapping using surface flattening via multidimensional scaling. *IEEE Trans. on Visualization and Computer Graphics*, 8(2):198–207, April–June 2002.