TopoLayout: Multi-Level Graph Layout by Topological Features

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

***Abstract*— We describe TopoLayout, a feature-based, multi-level algorithm that draws undirected graphs based on the topological features they contain. Topological features are detected recursively inside the graph, and their subgraphs are collapsed into single nodes, forming a graph hierarchy. Each feature is drawn with an algorithm tuned for its topology. As would be expected from a feature-based approach, the runtime and visual quality of TopoLayout depends 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 a variety of datasets with a range of connectivities and sizes. TopoLayout frequently improves the results in terms of speed and visual quality on these datasets.**

***Index Terms*— Information Visualization, Graphs and Networks, Graph Visualization**

I. Introduction

Recently, **multi-level** approaches for graph drawing have been studied to overcome the size and visual quality limitations of previous work. Multi-level algorithms typically construct a graph hierarchy with the original graph at the leaf level and coarser approximations at higher levels. Current multi-level approaches typically only exploit local connectivity in the graph and treat all nodes and edges similarly. The resulting drawings are uniform, but low-level structure within the high-level structure of the graph is difficult to see.

We introduce a **feature-based** approach to multi-level graph drawing. In this approach, features of interest are recursively detected in the graph and replaced with meta-nodes at a coarser level. Appropriate drawing algorithms for each feature are selected based on the type of feature detected. Our approach to feature-based, multi-level graph drawing recursively detects **topological features** such as trees, connected components, and biconnected components, which have been well studied in the literature. We also detect highly connected clusters: features of interest in power law or small world graphs. To show that we can expand our system beyond strict topological features, we detect when the high-dimensional embedder (HDE) [22] algorithm is a suitable choice for layout. HDE is an efficient algorithm for drawing a specific subset of general graphs, many of which are grids.

The primary contribution of this work is TopoLayout, the first feature-based, multi-level algorithm. Unlike previous multi-level algorithms, the graph hierarchy is drawn bottom-up, taking the space required to draw the features into account at higher levels of the graph hierarchy. Thus, all of our layout algorithms should be **area-aware**; that is, take varying node size into account. TopoLayout also introduces passes to eliminate all node-node overlaps and to reduce the number of node-edge and edge-edge crossings.

The performance of TopoLayout is compared to existing multi-level algorithms. Although TopoLayout does have its limitations, the approach is often faster and better able to illustrate low-level structure in the context of high-level graph structure.

II. Previous and Related Work

Given a general, undirected graph G consisting of N nodes and E edges, we concern ourselves with the problem of drawing G in two dimensions. Nodes are assigned two dimensional coordinates, and if two nodes share an edge it is drawn between them as a straight line.

The problem of drawing general, undirected graphs has been well studied. Before the late 1990s, the methods were primarily focused on force-directed approaches [7], [10]–[12], [20]. These methods perform well for many types of graphs, but do not scale to graphs of thousands of nodes. To overcome this limitation, multi-level approaches and approaches which rely more heavily on user interaction have been proposed. In addition, a few previous approaches do exploit topology. We also describe the HDE approach, so that our HDE detector can be understood.

In addition to the work presented here, we have also described some preliminary work on TopoLayout in a poster [2].

D. Archambault and T. Munzner are with University of British Columbia, {archam, tmm}@cs.ubc.ca
D. Auber is with University of Bordeaux I, auber@labri.fr

A. Multi-Level Graph Drawing Algorithms

Multi-level methods for graph drawing have been studied to improve algorithm run time with drawings of equal or increased visual quality. The spirit of these multi-level approaches is to recursively apply a coarsening operator to divide a very large input graph into a hierarchy of coarser ones. These techniques exploit the property that coarser graphs in the hierarchy are representative of the detailed ones, but can be more quickly laid out. Also, such decompositions help avoid local minima, allowing the algorithms to scale to larger datasets, improving both the running time and the visual quality of the final layout.

In Walshaw [30], an estimate of a solution of the maximal matching problem is used as a coarsening operator to construct the hierarchy. The maximal matching problem is to select the largest possible set of edges in the graph such that no two edges are incident to the same node.

Harel and Koren [19] 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.

The GRIP algorithm [13] coarsens by applying a filtration to the node set of the input graph. The filtration operator recursively constructs a maximal subset at each level i such that the graph theoretic distance between any two nodes of the subset is at least $2^{i-1}-1$.

The ACE algorithm [21] 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 matrix have been computed.

The Fast Multipole Multilevel Method, or FM^3, algorithm [16] is the first multi-level algorithm for general graphs with a provable worst case asymptotic runtime of $O(N \log N + E)$. In this approach, the graph is partitioned into subgraphs called *solar systems*. These solar systems are contracted down to single nodes and the process is repeated to create a hierarchy. The authors show that 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 [17].

All the multi-level algorithms described above use heuristics to construct their graph hierarchies which do not exploit low-level and high-level features in the data. The principal advantage of a feature-based approach, such as TopoLayout, over existing multi-level algorithms is the visualization of low-level features embedded in the high-level graph structure. In this case, our features are primarily topological features.

B. Interactive Exploration of Graph Hierarchies

A few papers have focused on the interactive exploration of graph hierarchies. These results could be applied to the hierarchies produced by multi-level algorithms. In most of these techniques, a precomputed layout of the graph is recursively coarsened into graph hierarchy using some graph theoretic distance information. Interesting views of the graph hierarchy can be displayed using a fisheye metaphor. The user specifies a focus region that is shown at its maximum level of detail. Coarser levels of the hierarchy are displayed at increasing distances from the focus region, providing context.

The topological fisheye views of Gansner *et al.* [14] constructs the hierarchies based on Delaunay triangulations and relative neighborhood graphs. The compound fisheye views of Abello *et al.* [1] constructs hierarchies based on a binary space partition of the layout or areas of relatively high connectivity detected using Markov clustering. The work of van Ham and van Wijk [29] provides a similar technique for visualizing small world graphs by merging clusters pairwise based on geometric distance between the clusters. A force-directed layout algorithm that reflects the underlying clusters in the graph is used as an initial step.

These techniques provide insight into the multi-level structure of graphs. However, in TopoLayout, we are primarily concerned with displaying as much multi-level structure as possible in the static layout without resorting to interaction.

C. Topological Features in Graph Drawing

Graph topology has been exploited previously in graph drawing, but never in a multi-level context. Two previous algorithms search for topology in the graph at a single level, both employing different algorithms depending on the topology detected.

Niggemann and Stein [25] 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 containing statistics about the subgraph, including the

number of connected components, biconnected components, and Λ-clusters found. An optimal layout for a feature vector is found through regression learning on a large database of graphs. Each graph in the database is drawn with several layout algorithms and evaluated using a quality metric, then the best drawing is selected. Although the work produces some visually convincing results, the largest graph drawn was a thousand nodes. No explicit performance numbers were given, but the time required for precomputation is a major limitation.

Six and Tollis [27] decompose the graph into biconnected graphs and draw 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 types.

D. High-Dimensional Embedder (HDE)

In addition to strict topological features, TopoLayout detects when the High-Dimensional Embedder, or HDE, algorithm [22] of Harel and Koren is an appropriate choice. HDE is related to a rich family of mathematical approaches which have been explored as solutions to problems ranging from flattening curved surfaces [26] to texture mapping in computer graphics [31]. These algorithms select 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 the pivots 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 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 a breadth-first search for unweighted graphs or Dijkstra's algorithm for weighted graphs. For the remaining $d-1$ pivots, the node with furthest graph theoretic distance from the pivot is selected in order to maximize variance on each axis. The layout of the graph in the d-dimensional space is encoded in a n by d matrix (Harel and Koren used $d = 50$). PCA maps the drawing into two dimensions by computing the eigenvectors of the matrix and selecting the two of largest eigenvalue. These principal components correspond to the directions of maximal variance in the high-dimensional space. The eigenvectors are mapped to the x and y positions of the nodes to produce the final layout. HDE thus has a running time of $O(d(N \log N + E))$ or $O(d(N+E))$ depending on whether breadth-first search or Dijkstra's algorithm is used.

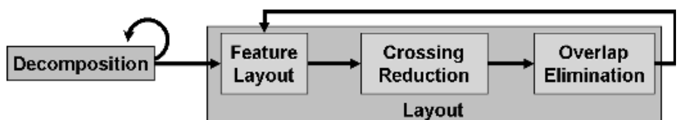

Fig. 1. TopoLayout algorithm phases.

III. ALGORITHM

The TopoLayout framework consists of four main phases as shown in Figure 1. The **decomposition** phase is the same as the coarsening operator of multi-level techniques. It recursively creates our feature hierarchy and identifies the feature type of each subgraph. The **feature layout** phase draws each subgraph in the graph hierarchy 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 in each subgraph. Finally, the **overlap elimination** phase ensures that no two nodes overlap in the final drawing.

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. In the decomposition phase, we have not found previous work describing our tree detection algorithm. In the feature layout phase, we provide a weighting scheme for HDE [22] and slightly modify GEM [11] so that they are area-aware. The crossing reduction phase is new to multi-level algorithms and is a novel algorithm of our own. The overlap elimination phase is new to multi-level algorithms, but is a direct application of previous work. All other algorithms are straightforward applications of the literature.

A. 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 feature is collapsed into a single node. The process is applied recursively to the graph, constructing our feature hierarchy.

In this section, we first define some terms which will be used to describe our decomposition phase. Then, we describe the general decomposition phase algorithm. Finally, we describe the individual topological feature detection algorithms which are applied to the input graph.

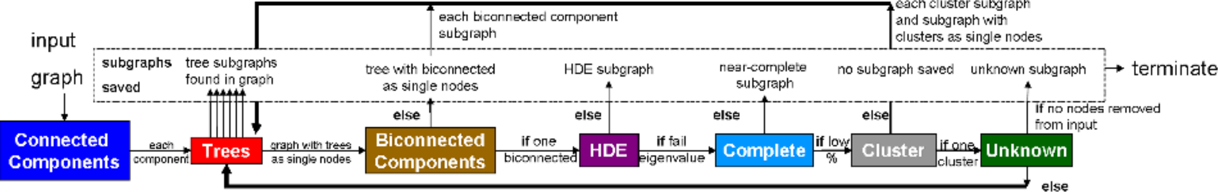

Fig. 3. Decomposition phase for TopoLayout. Detection algorithms in boxes coloured by feature type as in Figure 2. If a clause on a horizontal is true, we transition along the arrow. Otherwise, we follow the vertical arrow to save some subgraphs and recursively decompose others. Bold arrows indicate the recursive cases.

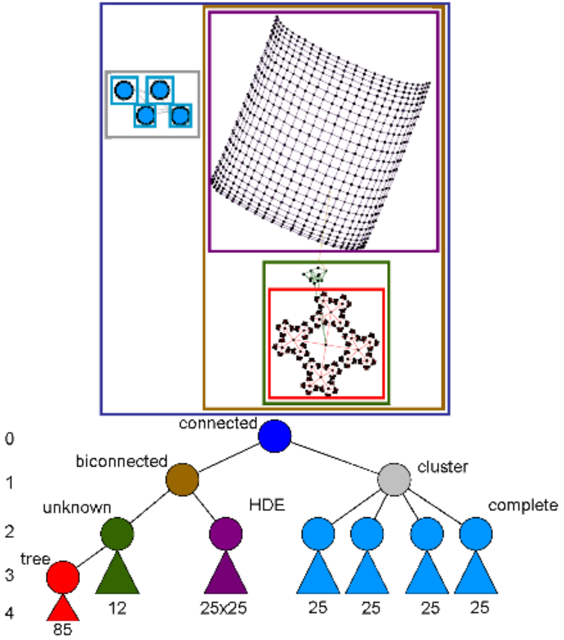

Fig. 2. Feature hierarchy after decomposition, with topology encoded by colour. **Top**: Layout annotated with bounding boxes to show hierarchy structure: meta-nodes encompass the subgraphs of their children. **Bottom**: Diagram of feature hierarchy, with levels enumerated and nodes labeled by feature type.

1) Definitions: The decomposition phase recursively constructs the feature hierarchy. An example feature hierarchy is shown in Figure 2. The levels of this hierarchy are defined with containment relationships: a node at level i is a parent of all the nodes in the feature it contains at level $i+1$.

We call the nodes of the input graph **leaves** as they terminate all paths in the feature hierarchy. Note that the computed hierarchy is rarely balanced and leaves can occur at any level. A **meta-node** is a node that contains either leaves of the hierarchy, or other meta-nodes. It represents a topological feature in TopoLayout. During the construction of a meta-node n, for the set of edges adjacent to one node inside of n and one node outside of n, we create a **meta-edge** between n and the node outside of n. A meta-edge contains a list of pointers to the edges in the input graph which they represent. We construct this list as the algorithm creates each meta-node. In Figure 2, meta-nodes are the rectangles in the diagram. 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.

A **connected component** is a subgraph where there exists a path between any pair of nodes in the subgraph. They are coloured blue. **Trees** are subgraphs without cycles and are coloured red. 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. Nodes and edges separating biconnected components are coloured tan. A **complete** graph has all possible edges present, so each node is connected to all others. Nodes and edges of complete graphs are coloured cyan. A **cluster** is a subgraph formed by some clustering algorithm. In our implementation, we use the strength metric [4] for clustering. Edges separating clusters are coloured grey. We then determine if **HDE** is a suitable algorithm to lay out the subgraph. If it is, it is coloured purple. Finally, if the decomposition phase cannot identify the topology of the subgraph, it is labeled **unknown** and coloured green.

2) Decomposition Algorithm: Figure 3 describes the decomposition algorithm in detail with the boxes of the diagram coloured using the scheme described above.

The first step of the decomposition phase replaces each connected component with a meta-node. The decomposition operator is recursively applied to each connected component detected. Connected component decomposition is never executed again, as subsequent detection algorithms do not disconnect the graph.

Next, we segment out the trees present in each connected component. Each tree is saved as a subgraph and replaced by a single meta-node. The graph with all trees removed and replaced by meta-nodes is passed to biconnected component detection.

If more than one biconnected component is detected, the decomposition phase is recursively applied to each of them. The tree with all biconnected components removed

and replaced by meta-nodes is saved as a subgraph. This subgraph must be a tree as explained in Section III-B.1. If only one biconnected component is present, it is passed to HDE detection.

If a layout of the subgraph with HDE has the properties we describe in Section III-A.8, it is saved and area-aware HDE is used for the subgraph in the final layout. If the layout does not have these properties, the graph is passed to complete detection.

If a graph is complete, its subgraph is saved. Otherwise, the graph is passed to cluster detection.

If more than one cluster is found by the clustering algorithm, the decomposition operator is recursively applied to every cluster, and also to the graph which results from replacing each cluster with a meta-node. If there is only one cluster, the subgraph is labeled unknown.

If the unknown subgraph has any collapsed features resulting from this pass of the decomposition operator, the decomposition operator is recursively applied to the subgraph. Otherwise, the unknown subgraph is saved and the decomposition phase terminates.

We experimented with several orderings of the decomposition algorithms. Our rationale for applying the detection algorithms in the order presented is as follows. 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 would disconnect the tree into two components. 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 partition of the graph into highly connected subgraphs when more meaningful topological features cannot be found.

3) Connected Components: We detect connected components using a series of depth-first searches to compute spanning trees for each component. We refer the reader to Baase and Van Gelder [5] for details of this standard algorithm which runs in $O(N+E)$ time.

4) Trees: 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. 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 $O(N+E)$ time.

5) Biconnected Components: A good description of a standard biconnected component detection algorithm is also given by Baase and Van Gelder [5]. Biconnected components are detected in the graph by performing a depth-first search. 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 $O(N+E)$ time.

6) Complete Graphs: We detect complete graphs by taking the ratio of the number of edges in the graph to the number of possible edges given the number of nodes. We could easily detect near-complete graphs by considering a threshold below 100%. An interesting area for future work would be to determine an appropriate value that has a sound theoretical justification, rather than being determined empirically. If the number of nodes and edges is known, the ratio is computed in $O(1)$ time.

7) Clusters: We compute clusters using the strength metric [4]. 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 connecting 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 of an edge between elements in any pair of these three sets or two elements in $W(u,v)$. These edges can be computed in $O(r)$ time where r is the maximum degree of a node in the graph. We can thus detect clusters in $O(rE)$ time. For near-complete graphs, the performance of the algorithm would degrade to $O(N^3)$, but typically, the nodes of the graph are of bounded degree and there are few high degree nodes. Thus, in practice, the algorithm can be run on large graphs.

8) HDE Detection: To determine if HDE is a suitable layout algorithm for the subgraph, we analyze the eigenvalues produced by an HDE layout of the graph. In PCA, the amount of variance in the data captured by an eigenvector is its eigenvalue [23]. We first determine if there is enough variance in the data, or if its largest eigenvalue is above a minimum threshold value. In our system, a value of 100 was determined empirically. This minimum variance is required as some projections place the majority of their nodes on top of each other. Next, we compare the percentage of variance accounted for by the top two eigenvectors. This percentage is computed and compared to the sum of all eigenvalues. In good two dimensional layouts, the percentage of variance of the largest two eigenvalues is nearly the same. If the variance is not symmetric along these two directions, we only use HDE in the case when the top three eigenvalues hold all of the variance, no eigenvalue holds too much of the

variance, and variance in the third dimension is small. Threshold values of 60% and 15% respectively were determined empirically. Since the edges of the graph are unweighted, our HDE detector uses a breadth-first search and runs in $O(d(N+E))$ time.

B. 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; that is, 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.

The initial layout of the features in the graph depends on the detected feature type. We employ four types of layout algorithms: tree, circular, HDE, and force-directed. We also describe passes to reduce the number of node-edge and edge-edge crossings and to eliminate all node-node overlaps.

Algorithm 1 Pseudocode for the feature layout phase.

```
layout (subgraph s)
  for all meta-nodes c ∈ s do
    c.size ←boundingBox (layout (c.subgraph));
  layOutFeature (s);
  reduceCrossings (s);
  eliminateOverlaps (s);
```

1) Area-Aware Tree Layout: These algorithms are used for tree and biconnected component feature 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 [27] and place the node between the two components. If an internal node of the tree is a meta-node, its children are sorted radially around the internal node based on the positions of the nodes at lower levels in the hierarchy that connect the children to the meta-node. TopoLayout can use any tree layout algorithm that is area-aware for drawing. We use the bubble tree algorithm [15] for trees of low depth and high branching factor and an area-aware version of the Walker algorithm [6] 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.

2) Area-Aware Circular Layout: These algorithms highlight complete graphs by simply placing the nodes of the graph around a circle. Although circular layouts yield low visual quality drawings for general graphs because they have many crossings, they are a good choice for complete graphs because they provide visual pop-out for cliques. The algorithm runs in $O(N)$ time.

3) Area-Aware HDE: This algorithm is used to lay out subgraphs found by our HDE detector. Area-aware HDE is the standard HDE approach [22] with weighted edges. The weight of each edge is the maximum radius of the adjacent nodes, with a minimum weight of one. Since the graph edges are weighted, area-aware HDE uses Dijkstra's algorithm and runs in $O(d(N \log N+E))$ time.

4) Area-Aware GEM: This default algorithm is used for all other cases. It is similar to the algorithms developed by Harel and Koren [18] who adapted Fruchterman-Reingold [12], Kamada-Kawai [20], and combinations of these algorithms. Area-aware GEM is a modified version of the GEM algorithm [11] where nodes are considered charges and the edges are considered springs. The system is placed in an initial configuration and is released until it reaches an equilibrium. Oscillations and rotations about equally optimal positions are dampened.

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 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:

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

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

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 algorithm stops after the nodes in the graph do not move much in the plane

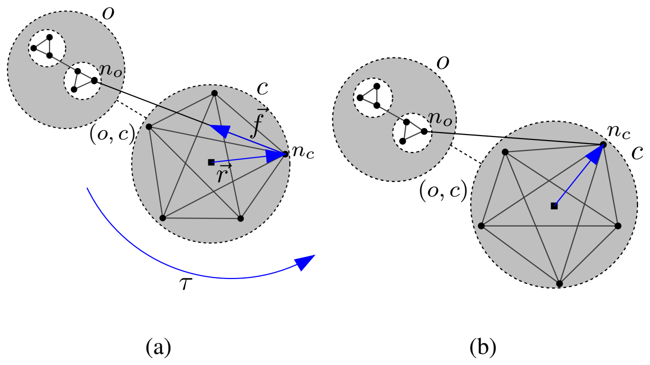

Fig. 4. Reducing crossings with torque. **(a)** Computing the torsional force τ on c exerted by the edge (n_o, n_c). **(b)** Applying τ results to rotate c. Dashed nodes and edges are meta-nodes and meta-edges. Solid nodes are leaves in the hierarchy. The square box is the centre of node c.

or N iterations have been completed. Thus, in the worst case, the complexity of the algorithm remains $O(N^3)$.

5) Crossing Reduction: We introduce a heuristic to rotate meta-nodes in our hierarchy, reducing crossings of edges in the original graph connecting nodes in different subgraphs as shown in Figure 4. The heuristic does not guarantee an elimination of node-edge or edge-edge crossings, but it reduces their number in most cases and also shortens edge length between subgraphs. Our approach is similar to that of Symeonidis and Tollis [28] 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 [20] in graph layout. In contrast, our approach is similar to GEM [11] 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 $\text{sg}(\vec{x})$ 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).

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

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 [11]. 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 the meta-node which contains it. This information can be saved 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 $O(LN_iE)$.

6) Overlap Elimination: In TopoLayout, although the area-aware tree and circular layout algorithms guarantee no node-node overlaps, neither area-aware GEM nor area-aware HDE does. To ensure that pairs of nodes do not overlap in our final layout, we perform a pass to 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 between 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.

We also implemented the Cluster Buster algorithm of Lyons *et al.* [24], 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 [8], which is discussed in detail in Dwyer *et al*'s technical report [9]. 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 V.

The overlap elimination phase is always executed on graphs drawn with HDE and area-aware GEM, since these algorithms do not guarantee the absence of overlaps. As the fast overlap removal algorithm only considers axis aligned nodes, the axis aligned bounding box of the rotated meta-node is computed.

IV. Algorithm Complexity

The worst-case complexity of our algorithm is $O(N^3)$ if no topological features are found and area-aware GEM is used. However, the algorithm in practice runs faster in most cases.

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_i. The value of d is the dimensionality of the high-dimensional space of the HDE algorithm, which is 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.

Algorithm	Complexity
Detection	
Tree	$O(N_i + E_i)$
Biconnected Component	$O(N_i + E_i)$
Connected Component	$O(N_i + E_i)$
HDE	$O(d(N_i + E_i))$
Complete	$O(1)$
Cluster	$O(r_i E_i)$
Initial Layout	
Bubble Tree	$O(N_i \log N_i)$
Walker Tree	$O(N_i)$
Area-Aware Circular	$O(N_i)$
Area-Aware GEM	$O(N_i^3)$
Area-Aware HDE	$O(d(N_i \log N_i + E_i))$
Refinement	
Crossing Reduction	$O(LN_iE)$
Overlap Elimination	$O(N_i \log N_i)$

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

V. Empirical Evaluation

We implemented the TopoLayout framework on top of the Tulip [3] 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.

Four multi-level algorithms were tested against TopoLayout. The code for GRIP[*], ACE[†], and HDE[‡] was available online and was incorporated into the Tulip framework. Stefan Hachul kindly supplied the FM^3 code, which was also incorporated into Tulip for testing. Harel and Koren's multi-level approach [19] was not tested. The source code for this implementation was unavailable. As our observed running times and visual quality results were very similar to those in Hachul and Jünger's empirical study [17], 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 III-A.1. Since the other graph drawing algorithms do not detect topological features automatically, the comparison is fair and demonstrates another advantage of our approach.

Our experiment was divided into two phases. *Synthetic Data* primarily consisted of benchmark datasets taken

[*] www.cs.arizona.edu/~kobourov/GRIP

[†] research.att.com/~yehuda/programs/ace.zip

[‡] research.att.com/~yehuda/programs/embedder.zip

from the graph drawing literature. These preliminary tests provided a baseline for the comparison of multi-level algorithms. *Real World Data* mostly consisted of datasets deemed real world in previous empirical evaluations. We added two additional datasets which came from real world data sources.

A. Synthetic Data

All but one of the synthetic graphs we study came from the Hachul and Jünger empirical evaluation [17] of multi-level algorithms. `Crack` is a standard graph drawing dataset part of the Walshaw Graph Partition Archive[§]. It was categorized as a *real world* graph in their study. The `6-ary`, `Snowflake`, `Spider`, and `Flower` datasets are each of the medium sized *challenging artificial* graphs of their study. 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 edge 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}. The last graph we test, `bi_walsh`, did not appear in the study. It is thirteen datasets from the Walshaw Graph Partition Archive connected by twelve single edges into one component. The purpose of this dataset is to demonstrate that our HDE detection algorithm works well and that other multi-level algorithms, including HDE and ACE, have difficulty drawing this dataset. The results of this part of the empirical evaluation are shown in Figure 6.

B. Real World Data

In addition to synthetic data, we use data that was considered real world in other empirical evaluations. The first three datasets are *challenging real world* graphs in Hachul and Jünger's empirical evaluation [17]. The `ug_380` and `dg_1087` graphs are from the *AT&T Graph Library*[¶]. The `Add32` dataset is from the Walshaw Graph Partition Archive. The graph is representative of the underlying hardware structure of a thirty-two bit adder. In addition to these datasets, we added two more. `UBC` is the hyperlink structure of the department of computer science at the University of British Columbia's web site acquired using a depth-first search cut off at about 40,000 nodes. `IMDB 1999` is a subset of the Internet Movie Database[‖]. It shows all actors in movies and television shows released in 1999 who are three or fewer hops from Jake Gyllenhaal in the movie *October Sky*. We select this actor because he has a relatively low branching factor in that year. The results of this part of the empirical evaluation are shown in Figure 7.

[§] `staffweb.cms.gre.ac.uk/~c.walshaw/partition`
[¶] `www.graphdrawing.org`
[‖] `www.imdb.com`

VI. Discussion

Since most of the data for these tests came from the Hachul and Jünger empirical evaluation [17], we are able to compare the results of this evaluation with their findings. For the most part, we have reproduced their results. For some of the images produced by GRIP in the study of Hachul and Jünger, it seems that a three dimensional layout had been selected. In this empirical evaluation we use only two-dimensional layouts. We will highlight where this makes a major difference between the results of the two studies when we discuss the drawings of each of the datasets.

In general, three of the drawing algorithms performed well on all of the datasets: GRIP, FM^3, and TopoLayout. The ACE and HDE algorithms did not perform well on any of the datasets in this evaluation with the exception of `Crack`. ACE and HDE appear to only work well on graphs which are *mesh-like* in structure. As a result of this finding, we focus our attention on GRIP, FM^3, and TopoLayout for the remainder of this discussion and will only show the results for these three algorithms on the real world data.

A. Synthetic Data

In summary, TopoLayout was consistently faster than FM^3 on this data and had similar running times to that of GRIP. The only two exceptions are `Spider` graphs and `Flower` graphs where TopoLayout was on the order of a couple of minutes while FM^3 and GRIP were on the order of seconds. This time delay was due to GEM and the cluster detection algorithm, the slowest parts of TopoLayout. TopoLayout produced drawings of equal or improved visual quality on all datasets in these tests.

1) `Crack`*:* All three algorithms produced drawings of similar visual quality for this mesh dataset. This result differs from the Hachul and Jünger empirical evaluation in that GRIP does not have any folds in the layout.

2) `6-ary`*:* On this 6-ary tree of depth five, TopoLayout produced the drawing which clarified the most high-level and low-level structure, followed by FM^3, and GRIP. In the TopoLayout drawing, we can see both high-level and low-level structure in the tree as the tree is

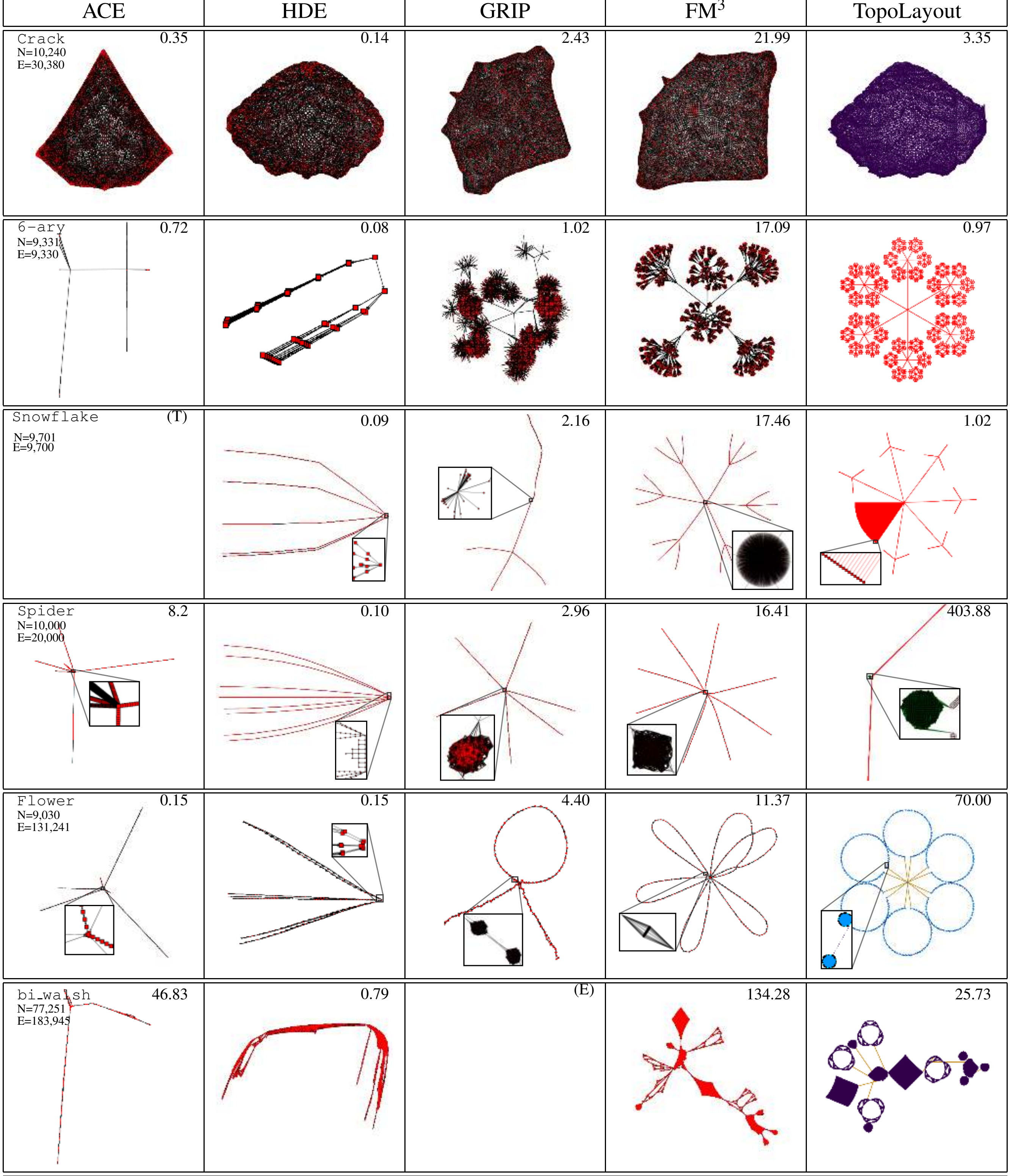

Fig. 6. Layouts of several datasets using ACE, HDE, GRIP, FM^3, and TopoLayout for the synthetic data described in Section V-A. For all rows, blank squares indicate no drawing produced. Dataset name, number of nodes, and number of edges appear in the top left hand corner of the leftmost column. Times in seconds, or reasons for no drawing, appear in the upper right corner of each entry. (T) indicates no drawing produced after a timeout of four hours of program execution. (E) indicates no drawing produced due to an execution-time error. Insets show roughly the same set of nodes.

detected and drawn using bubble tree. For FM^3, the high-level structure of the tree is apparent, but the low-level structure is obscured by many node-node overlaps and edge crossings. With GRIP, part of the high-level

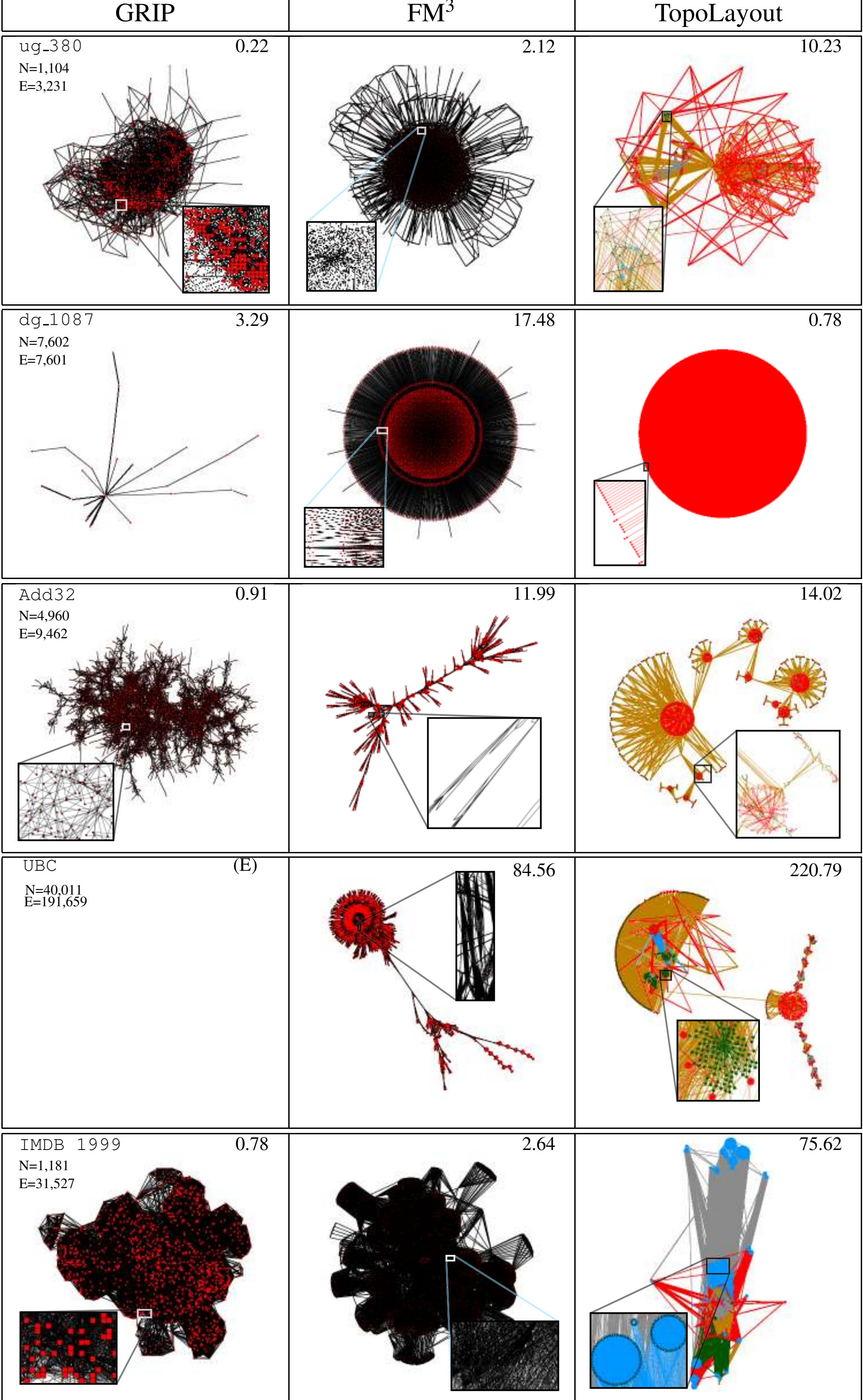

Fig. 7. Layouts of several datasets using GRIP, FM^3, and TopoLayout for the real world data described in Section V-B. For all rows, blank squares indicate no drawing produced. Dataset name, number of nodes, and number of edges appear in the top left hand corner of the leftmost column. Times in seconds, or reasons for no drawing, appear in the upper right corner of each entry. (E) indicates no drawing produced due to an execution-time error. Insets show roughly the same set of nodes.

structure is obscured because a few of the main branches of the tree overlap. Low-level structure is not apparent due to many node-node overlaps.

3) `Snowflake`: The Snowflake graph is a tree. It has a high-level deep tree structure with a low level core of many single nodes connected to the tree root. TopoLayout detects this tree and uses bubble tree to draw it without node-node or node-edge overlaps. It also shows the core with a clearly visible fan-out of the single nodes connected to the root on the lower left of the drawing. FM^3 was also able to draw the high-level structure and low-level structure in the tree to some degree. However, it is very difficult to understand how the single nodes clumped around the root are connected and it is very hard to see this feature at a high level. GRIP was able to draw only part of the high-level structure and part of the low-level structure around the root of the tree. There are many overlaps, making the drawing difficult to understand.

4) `Spider`: All three algorithms drew this dataset with a similar level of visual quality. The high-level structures of the well-connected head and the eight long paths or legs are visible in all three drawings. In TopoLayout, at a high level, it is hard to see the legs as they have been detected as two deep trees. The low-level structures in this graph are drawn with a similar level of visual quality. The drawing produced by GRIP of this dataset differs from the Hachul and Jünger empirical evaluation in that the legs do not cross.

5) `Flower`: TopoLayout and FM^3 are able to draw most of the high-level and low-level structures in this dataset while GRIP is only able to draw parts of both. TopoLayout draws each of the K_{30} cliques with circular layout, for a visual indication that the graphs are complete. Using HDE, TopoLayout draws each of the six symmetric loops. The K_{30} at the centre of the drawing, when removed, separates the graph into six connected components. Thus, the highest level structure is a set of six biconnected components drawn with the bubble tree drawing algorithm. This dataset gives promise for a feature-based approach, where several different algorithms can be integrated smoothly into one drawing. FM^3 does well on the high-level structure of `Flower`, but two of the loops cross. In terms of low-level structure, it is difficult to tell if the K_{30} subgraphs are actually complete as there are many node overlaps in the drawing. GRIP is unable to draw five of the loops of the high-level structure. As with the FM^3 drawing, it is also very difficult to tell if the K_{30} subgraphs are complete.

6) `bi_walsh`: TopoLayout and FM^3 draw the high-level biconnected structure of this dataset well. TopoLayout detects the high-level biconnected structure present in this graph and draws it using bubble tree. It uses HDE to draw each of the thirteen mesh-like datasets present in the graph. FM^3 is able to draw the high-level biconnected structure well. However, it is difficult to see the mesh-like graphs in each of the individual components. GRIP was unable to produce a drawing for this dataset due to an internal error.

B. Real World Data

On the real world datasets, the TopoLayout running times were usually of a similar order of magnitude as FM^3. The only exception was `IMDB 1999` where TopoLayout took just over a minute and FM^3 took two seconds. GRIP was faster than all algorithms, but it frequently yielded results of lower visual quality. Overall, TopoLayout either improved or had similar visual quality results on all of the graphs in the evaluation.

1) `ug_380`: This dataset contains a single node of very high degree with some interesting topological structure at some graph theoretic distance from this central node. The high-level structure could be improved in all three drawings. However, TopoLayout is able to provide an interesting insight into the high-level structure of this dataset at the central core. It is able to segment out the high degree node at the centre of the drawing and suggests that the topology of the central core is actually two components rather than one. TopoLayout is also able to draw some of the low-level structure present in the dataset. The FM^3 drawing is unable to segment out the core into these two components. It is very difficult to determine which node is the high degree node in the drawing. The topology of the core is unclear as the majority of the nodes and edges are placed at the centre of the drawing. It is also very hard to make out the low-level structure which exists in the graph, but the drawing is more uniform and compact. The drawing produced by GRIP is of similar visual quality to that of FM^3.

2) `dg_1087`: This dataset is topologically a tree with a very high degree node at the root. TopoLayout detects this tree and draws it with bubble tree, producing a drawing free of node-edge and node-node overlaps. It is very difficult to tell in the FM^3 drawing if this graph is indeed a tree. Many of the nodes are clumped into the central area of the drawing and it is difficult to see how they interconnect. Thus, the high-level structure of the tree is drawn well whereas the algorithm has difficulty clarifying the low-level structure in the dataset. FM^3 does, however, have the advantage of a more compact drawing. GRIP is able to draw some of the high-level structure, but much of it is hidden by many overlaps. It is very hard to see the low-level structure of this dataset using GRIP.

3) `Add32`: In this TopoLayout drawing we see promise in our feature-based approach on real world

data. The high-level biconnected structure of the adder is clearly visible in tan and is drawn with bubble tree. The low-level structure of the adder is integrated smoothly into the drawing using tree drawing and force-directed algorithms locally. Thus, we are able to see most of the low-level structure and high-level structure in the dataset. The FM^3 drawing does reveal some of the high-level structure in the dataset, but some of it is obscured by edge and node occlusion. It is quite difficult to see low-level structure in the dataset. With the GRIP drawing, some of the high-level tree structure is visible, but it is less apparent.

4) `UBC`*:* TopoLayout is able to draw the high-level tree structure of this dataset. This tree structure is expected as the dataset was acquired using breadth-first search. In addition to the high-level structure, the algorithm is able to visualize some of the low-level structure in the more strongly connected left part of the drawing. FM^3 is also able to draw the high-level tree structure in the dataset, but it has difficulty drawing the low-level structure present in the upper left corner. GRIP was unable to produce a drawing of this dataset due to an internal error.

5) `IMDB 1999`*:* This IMDB subset is a very hard dataset to draw because of its high connectivity. All three algorithms have difficulty in revealing the high-level structure in the dataset. TopoLayout is able to reveal some of the high-level structure, but much of it is obscured by large swathes of edges. It is, however, able to segment out and draw the complete cliques of actors in this dataset. These cliques correspond to movies: any actor in a movie acts with all other actors in that movie. The strength metric was able to clearly segment these movies out and TopoLayout was able to draw them with circular layout. In the FM^3 and GRIP drawings of this dataset, it is hard to see either the high-level or low-level structure in the drawing as many of the nodes and edges are placed in the same area.

VII. Future Work

One obvious way to improve our results is to have faster detection and drawing algorithms which produce results of higher or equal visual quality. Figure 5 shows that strength decomposition is the slowest detection algorithm and GEM is the slowest drawing algorithm. These complexity results are reflected in the running times of TopoLayout on `Spider` and `Flower` in Section V-A. To improve `Spider`, we would implement an area-aware version of the FM^3 algorithm. The adaption of Fruchterman and Reingold used by the algorithm should be straightforward, but making the multi-level solar system hierarchy area-aware would be non-trivial. To improve `Flower`, we could use faster clustering algorithms or improve the running time of the strength metric. Considering these two algorithms execute when no features are found, we should also investigate new types of features that can be found in graphs.

We will continue to improve upon our detection algorithm for HDE components and possibly introduce a detection algorithm for ACE. Our HDE detector is very good at finding mesh-like graphs with a two dimensional structure, but not ones with a three dimensional structure or a small number of nodes. We note that TopoLayout does not perform well on three of the *challenging real world* datasets in the Hachul and Jünger empirical evaluation [17]: `bcsstk33`, `bcsstk31_con`, and `bcsstk32`. The structure of these datasets is very mesh-like and either ACE or HDE seems to perform well on at least two them. Thus, discovering more efficient and accurate ACE and HDE detectors would be beneficial.

It would also be fruitful to adapt one of the recent approaches to interactive exploration [1], [14], [29] to work with the TopoLayout framework. We believe that this work would help clarify the multi-level structure of feature based hierarchies, especially when the amount of edge occlusion is high.

VIII. Conclusion

We have presented TopoLayout, a multi-level algorithm for drawing large graphs. The approach of TopoLayout is feature-based. It detects topological features in the graph and can determine when the HDE algorithm is an appropriate algorithm for layout. The experimental results comparing TopoLayout to four other multi-level approaches show that a feature-based approach has promise in drawing low-level and high-level structure in large graphs.

Acknowledgments

We thank Stefan 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 Ciarán Llachlan Leavitt and Juliet O'Keefe for help in editing this paper and the anonymous reviewers of TVCG for their suggestions.

References

[1] J. Abello, S. G. Kobourov, and R. Yusufov. Visualizing large graphs with compound-fisheye views and treemaps. In *Proc. 12th Int. Symp. on Graph Drawing*, volume 3383 of *LNCS*, pages 431–441. Springer-Verlag, 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.

[3] 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.
[4] 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.
[5] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 3rd edition, 2000.
[6] 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.
[7] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM TOG (Trans. Graphics)*, 15(4):301–331, 1996.
[8] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. In *Proc. 13th Int. Symp. on Graph Drawing*. Springer-Verlag, 2005.
[9] 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.
[10] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
[11] 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.
[12] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, November 1991.
[13] P. Gajer and S. G. Kobourov. GRIP: Graph drawing with intelligent placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.
[14] E. Gansner, Y. Koren, and S. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):457–468, 2005.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. *Journal of Graph Algorithms and Applications*, 6:179–202, 2002.
[20] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.
[21] Y. Koren, L. Carmel, and D. Harel. Drawing huge graphs by algebraic multigrid optimization. *Multiscale Modeling and Simulation*, 1(4):645–673, 2003.
[22] 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.
[23] D. C. Lay. *Linear Algebra and Its Applications*. Addison-Wesley, 3rd edition, 1997.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] F. van Ham and J. van Wijk. Interactive visualization of small world graphs. In *Proc. IEEE Symposium on Information Visualization (InfoVis'04)*, pages 199–206, 2004.
[30] C. Walshaw. A multilevel algorithm for force-directed graph drawing. *Journal of Graph Algorithms*, 7(3):253–285, 2003.
[31] 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.

Daniel Archambault received the BSc Hons. degree from Queen's University at Kingston in 2001 and the MSc degree from the University of British Columbia in 2003. He is currently a PhD student at the University of British Columbia in the Department of Computer Science. His interests include graph drawing, information visualization, visualization, and computer graphics.

Tamara Munzner received the PhD degree in 2000 from Stanford and has been an assistant professor in the Computer Science Department of the University of British Columbia since 2002. She was a technical staff member at the University of Minnesota Geometry Center from 1991 to 1995, and a research scientist at the Compaq Systems Research Center from 2000 to 2002. Her research interests are information visualization, graph drawing, and dimensionality reduction.

David Auber received the PhD degree in 2003 from the University of Bordeaux I. He has been an assistant professor in the University of Bordeaux Department of Computer Science since 2004. His current research interests are information visualization, graph drawing, bioinformatics, databases, and software engineering.