Feature-Based Graph Visualization

by

Daniel William Archambault

B.Sc.H., Queen's University at Kingston, 2001
M.Sc., The University of British Columbia, 2003

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

(Vancouver, Canada)

Abstract

A graph consists of a set and a binary relation on that set. Each element of the set is a node of the graph, while each element of the binary relation is an edge of the graph that encodes a relationship between two nodes. Graph are pervasive in many areas of science, engineering, and the social sciences: servers on the Internet are connected, proteins interact in large biological systems, social networks encode the relationships between people, and functions call each other in a program. In these domains, the graphs can become very large, consisting of hundreds of thousands of nodes and millions of edges.

Graph drawing approaches endeavour to place these nodes in two or three-dimensional space with the intention of fostering an understanding of the binary relation by a human being examining the image. However, many of these approaches to drawing do not exploit higher-level structures in the graph beyond the nodes and edges. Frequently, these structures can be exploited for drawing. As an example, consider a large computer network where nodes are servers and edges are connections between those servers. If a user would like understand how servers at UBC connect to the rest of the network, a drawing that accentuates the set of nodes representing those servers may be more helpful than an approach where all nodes are drawn in the same way. In a feature-based approach, features are subgraphs exploited for the purposes of drawing. We endeavour to depict not only the binary relation, but the high-level relationships between features.

This thesis extensively explores a feature-based approach to graph visualization and demonstrates the viability of tools that aid in the visualization of large graphs. Our contributions lie in presenting and evaluating novel techniques and algorithms for graph visualization. We implement five systems in order to empirically evaluate these techniques and algorithms, comparing them to previous approaches.

Table of Contents

Appendices

List of Figures

Acknowledgements

Writing a doctoral dissertation is a very long process. Without the support of a huge number of people, my success in completing this work simply would not have been possible. I would like to start off by thanking everyone who helped me here at UBC and at Queen's University who helped me make this moment possible. I know that I'm going to miss someone, primarily because I'm writing this the night before the thesis is due, but your help and efforts to bring me to this point were very much appreciated.

First of all, I would like to thank Tamara Munzner for all of her guidance and support throughout the program. When I discovered that I was interested in visualization somewhere along the course of my Master's degree, I was thoroughly delighted to discover that one of those researchers was coming to UBC. Even better, she did all this cool mathy stuff! I do very much appreciate her perspective and her approach to information visualization, and the many ideas we have discussed about the field as a whole. I consider myself very lucky to have such a strong researcher in the field as my advisor.

Secondly, I would like to thank my co-adviser from the University of Bordeaux, David Auber. Thank you for exposing me to Tulip and letting me play, and sometimes hang myself, on such a large software project. I will never look at coding in the same way ever again. Your knowledge of graph drawing software practices is very deep, and I appreciate the many conversations we had. Also, I thank you for the we spent in Europe gave me the experience of doing research in a completely different part of the world. I have learned, and hope to continue to learn, a lot from you.

I would like to thank my remaining committee member, David Kirkpatrick. I would also like to thank my university examiners, Michiel van de Panne and Wayne Maddison. Finally, I would like to thank my external examiner, Peter Eades. Thank you for your insightful comments on the thesis and my research. You have made this product better and have given me plenty of ideas to think about in the future.

I would like to thank Bill Cheswick for numerous discussions about how graph drawing can be applied to computer networking. Although we did not succeed in producing a computer network visualization tool in the end,

your suggestions helped me in the development of many of the techniques and algorithms presented in this thesis.

I would like to thank the many members of the Imager Lab for graphics at UBC with whom I had the pleasure of working. There are too many names to put here so I will just mention the members of the InfoVis group over the years: Heidi Lam, Stephen Ingram, Peter McLachlan, James Slack, Aaron Barsky, Ciarán Llachlan Leavitt, Qiang Kong, Dmitry Nekrasovski, Adam Bondar, Matt Williams, and Kristian Hildebrand. You guys made "Queen's II" more fun.

In the fall, I had the wonderful opportunity to teach my first class at the University of Victoria where I met a number of wonderful people. First of all, I would like to thank everyone who took CSC 115 in the fall term of 2007. You guys where an amazing bunch of students and I'm sure that you're all going to do very well. I would like to thank Yvonne Coady and Lill Ann Jackson for their guidance in the wonderful world of teaching. I would like to thank Michael Zastre and Jon Muzio for giving me the opportunity to teach. Also, I would like to thank all of the faculty, staff, and students who made my time there more than enjoyable: Melanie Tory, Bruce and Amy Gooch, Brian Wyvill, Isabel Campos, Nancy Chan, Carol Harkness, Victoria Li, Sharon Moulson, David Sprague, Fuqu Wu, Kedar Shrikhande, Dan Dan (We're Dan^3!) Huang, and Subhanil Chakrabarty. You guys were amazing!

I would also like to thank a few people at Simon Fraser University who were good friends and excellent people to work with. Thank you Torsten Möller for introducing me to the field of visualization through what I still call "the toughest but coolest course of my life." Thanks to Richard Zhang and Matt Olson for interesting discussions about silhouettes and meshes.

I would also like to thank all the teaching role models I've had here at UBC. First off, the Jedesi, or Jennifer Jasper and Desiree Mou and all the people at TAG. I would really like to thank Paul Carter for helping me with the TA training program and allowing me to teach one lecture in his class. Thank you for the experience.

There are a vast number of people I need to thank from Queen's University, but some of you have already been mentioned above. Above all, I really need to thank Robin Dawes. You were the one who inspired me to start on this quest way back in the fall of 1997.

I must know other people outside of computer science? Oh, yes! I would like to thank all the wonderful people I met at Green College, one of the best environments in which to live. Also, I would specifically like to thank Laura Turner, Donald Derrick, Sherry Geraghty, Shaun Narine, Ron Fussell,

Ducky Sherwood, Raul Pacheco, and Tom De Rybel for being some of the coolest friends out there. I also need to thank Walter and Jill for warm cups of tea on rainy days, good books, and ρ.

I've always maintained that the arts were invented to keep humans sane. By that definition, some people think I should do more writing. I would like to thank the three editors I have had the opportunity to work with thus far: Julie E. Czerneda, Holly Phillips, and Cory Doctorow. You guys put into print dreams which may end up becoming research projects in a few years. I would also like to thank all those NGers out there, specifically: Jana, Jihane, Ruth, Lorne, Mark, Sarah, Fran, Kristen, Karina, Bobby, Jim, Darwin, Kate, and Nathan. Also, the countless others I know that I'm missing.

How could I forget the great people at WeWriters and Scribblers? You guys really helped me improve my writing over the years. I would like to also thank good friends like Keith, Mari, and Wes for being good friends and providing honest critiques.

I have had the best of luck with the places I lived in Vancouver. I need to thank Don Chandler for three years of companionship and running advice. Ruby, Nick, and Dennis for an amazing place and companionship in Victoria. I also need to thank Julie, Professional Princess Claudia, Best Bud Caroline, Bob, and M.T. for this last term in Vancouver. It was way too short. I know that you won't believe these words, but you guys are the best thing that has happened to me in Vancouver. I wish I had discovered you sooner. With any luck, I'll be DBBD soon!

Finally, I would like to thank my family for their continual support: Mom, Dad, Leah, and Chad. Even though I'm usually thousands of kilometres away, every time we talk on the phone it feels like I'm sitting on the deck overlooking the Rideau, and it's summer, of course.

Chapter 1

Introduction

Graphs, namely sets and the relationships between their elements, can be a useful model for data in many fields [47] including science, engineering, and the social sciences. For example, servers exchange packets and are linked by these connections, proteins interact in large biological systems, social networks depict the relationships between people, and functions call each other in a program. People frequently draw a picture of a graph by hand when thinking about a problem in these domains or to communicate findings [56]. For example, when discussing a large software project, people draw class diagrams to help them understand and design the system. Thus, the visual representation of a graph seems to play an important role in understanding the relationships between elements it encodes.

A graph consists of a set where each element is a **node**, and a binary relation on that set where each element of the binary relation is an **edge**. The field of graph drawing endeavours to automatically assign positions to the nodes in space to illustrate the binary relation. Nodes are typically placed in two or three-dimensional space because people cannot directly perceive drawings in four or more dimensions. Moreover, the information visualization literature documents the difficulties that people have in comprehending three-dimensional drawings of graphs [77]. Therefore, in this thesis, we only study two-dimensional placements of the nodes.

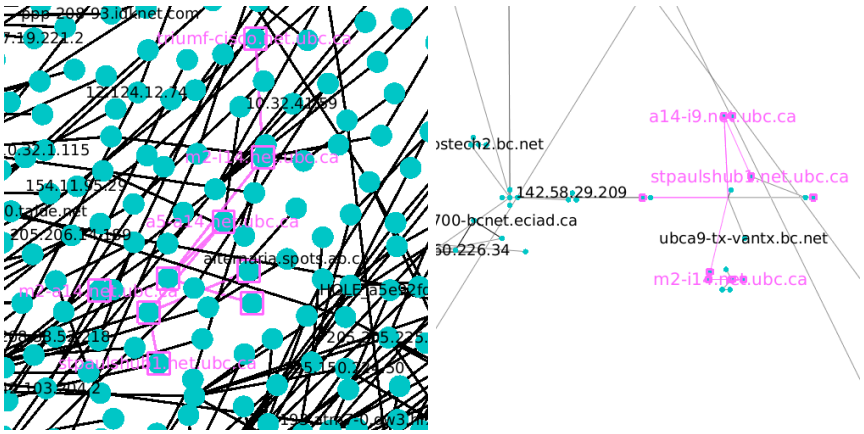
Much of graph drawing has concentrated on the drawing of graphs in two-dimensions based on graph connectivity. Graph connectivity is how the nodes are joined by the edges of the graph. Frequently, the heuristic used in graph drawing algorithms tries to map nodes that are close to each other in the binary relation close to each other in terms of Euclidean distance. Figure 1.1(a) shows an algorithm that is designed on this principle. The approach does well in keeping elements that are close in the graph close to each other spacially. However, elements of the graph not close to each other often interfere with this structure. Purchase [63] and Ware *et al.* [78] both determined and studied aesthetic criteria that have strong effects on drawing readability in these situations. Through formal user experiments, they determined and confirmed many different factors that help people read

and understand graphs, including edge bends, number of crossings, and drawing uniformity. These algorithms work well for viewing graphs when nodes of the graph are not classified explicitly into categories. However, in some tasks, users may want to impose categories on the nodes.

In Figure 1.1(b), we show the same part of a larger network, but the graph has been segmented into components that have few connections to other components in the graph. By using this clustering, we are better able to clarify the local structure of the network and how this local structure is connected to other elements of the graph. The field of graph clustering [47] existed for years before the innovation of graph drawing. Clusters are computed from the connectivity of the graph or an extrinsic classification of the nodes supplied with the graph. Many graph clustering algorithms have been applied in the fields of information visualization and graph drawing when a categorization of the nodes is available. Spacial position, one of the strongest perceptual cues, can be used to accentuate this categorization of the nodes. In these cases, uniform edge lengths in a graph drawing may not be ideal [59]. Algorithms in clustered graph drawing use spacial position explicitly, making tradeoffs to preserve the classification while trying to follow the criteria specified by Purchase and Ware.

In many domains, a clustered graph drawing approach may be more intuitive for users. In computer networks, a user may want to investigate how the primary servers of UBC interact with the entire Internet. This domain requires a classification of servers: clusters of connected servers that are part of UBC and clusters of connected servers that are not. In bioinformatics, users may want to explore chemical pathways in a graph representing a biological system. Nodes in this application may be classified by biological system, helping users understand how these systems interact. A representation of the graph using clusters may help users generate theories about how clusters are related. This thesis explores techniques for the investigation of this type of data.

This thesis explores **feature-based** approaches to graph visualization. A **feature** is a cluster we automatically detect in a larger graph. Features are used as a basis for hierarchy creation and graph layout. Unlike a cluster, a features has properties a specific graph drawing algorithm can exploit, whereas a cluster does not guarantee any such properties. If we consider a computer network, where nodes are servers and edges are connections between those servers, a feature can be a subgraph of connected servers that are part of UBC. This type of feature would be based on **attributes**, or data associated with the nodes of the graph. In this computer network, we could also group servers into subgraphs with many redundant connections.



(a) (b)

Figure 1.1: An example of how local network structure can be clarified in a feature-based approach. **(a)** The network, drawn with an approach that does not use clustering, draws UBC in a large computer network. Many nodes and edges interfere with the feature. **(b)** Feature-based drawing of the network. Fewer unrelated edges cross over the UBC portion because the feature is segmented out and drawn with suitable drawing algorithms.

This type of feature would be based on **connectivity**, or how the nodes are intrinsically linked by the edges of the graph.

We extensively explore areas of feature-based graph visualization and demonstrate the viability of tools that aid in the visualization of large graphs. Our contributions lie in presenting and evaluating novel techniques and algorithms for graph visualization. We implement systems in order to empirically evaluate these techniques and algorithms, comparing them to previous approaches.

1.1 Background

This section defines some key terms used in this thesis. Except for Section 1.5, the Introduction does not present any novel contributions of the work. This chapter simply provides necessary information for future chapters.

Section 1.1.1 presents definitions of standard graph and graph drawing terminology. Section 1.1.2 presents definitions for connectivity features while Section 1.1.3 defines attribute features. The difference between steerable and

interactive systems is discussed in Section 1.1.4. Section 1.1.5 defines graph hierarchy concepts that are central to many areas of this thesis. Finally, Section 1.1.6 describes general components of a feature-based graph drawing system.

1.1.1 Graph Definitions

A graph $G = (N, E)$ is defined by a set of N nodes and E edges. The nodes are a set of elements and the set of edges is a binary relation on N, as described by Equation (1.1):

$$E \subseteq N \times N \tag{1.1}$$

Asymptotic complexity numbers in this thesis are reported in terms of the number of nodes or edges in the graph. As such, they are reported using cardinalities of these sets where $|N|$ denotes the number of nodes in the graph and $|E|$ denotes the number of edges.

Using Equation (1.1), an edge set is represented using a collection of pairs (u, v). The techniques in this thesis are geared towards **undirected graphs** where E is a symmetric relation or $(u, v) \in E$ implies $(v, u) \in E$. Although the techniques would work on directed graphs, in situations where the pairs are ordered, specialized techniques exist for this specific type of data. These techniques are not the focus of this thesis. The graphs in this thesis are also **simple**, meaning that there are no multiedges or loops in the graph. A **multiedge** is a pair in E that has multiplicity greater than one. A **loop** is an edge (u, v) where $u = v$. Thus, a simple graph must satisfy the following two properties:

1. E is not a multiset.
2. For all $u \in N$ then $(u, u) \notin E$

Two nodes, u and v, in the graph are **adjacent** if $(u, v) \in E$. A node w is **incident** to (u, v) if $u = w$ or $v = w$. The **degree** of a node is the number of incident edges.

A **path** between nodes x and y is a sequence of nodes in G such that the sequence begins with x and ends with y where there is an edge between adjacent elements of the sequence. A **simple path** is a path where the multiplicity of any node in the sequence is at most one. A **cycle** is a simple path where $x = y$. The **graph-theoretic distance** between two nodes x and y is defined as the length of the shortest sequence of nodes along a path between x and y.

A **subgraph** G is a subset V' of V and a subset of all edges incident to two elements of V'. More formally, a subgraph $G' = (N', E')$ has node set N' defined by Equation (1.2) and edge set E' defined by Equation (1.3).

$$N' \subseteq N \tag{1.2}$$

$$E' \subseteq \{(u, v) | (u, v) \in E, u \in N', v \in N'\} \tag{1.3}$$

An **induced subgraph** is a subgraph where *subset or equal to* is replaced by *equality* in Equation (1.3). Informally, E' contains all edges incident to two elements of N'.

In order to define a **cluster** and **metanode**, we require several intermediate concepts. We define this terminology as done in *Drawing Graphs: Methods and Models* [47].

A **k-way** partition of a set C is a family of subsets $\{C_1, ..., C_k\}$ with:

1. $\bigcup_{i=1}^{k} C_i = C$
2. $C_i \bigcap C_j$ is empty for $i \neq j$

A **clustered graph** is a graph with a partition $\{C_1, ..., C_k\}$ on its node set. Each C_i is called a **cluster**.

Given a k-way partition $\{C_1, ..., C_k\}$ on the node set of a graph $G = (N, E)$, the **quotient graph** $G_q = (N_q, E_q)$ is defined by shrinking each partition into a single node where:

1. $N_q = \{C_1, ..., C_k\}$
2. $(C_i, C_j) \in E_q$ if and only if $i \neq j$ and there exists $v \in C_i$ and a $w \in C_j$ such that $(v, w) \in E$

Metanodes are the elements of N_q and **metaedges** are the elements of E_q. In our work, a metanode C_i is associated with the induced subgraph of C_i. This induced subgraph is known as the **metagraph** of C_i. Frequently, we will say that *a metanode contains a metagraph* when we mean *is associated with a metagraph*. This phrase stems from the fact that in graph hierarchies, as will be explained in Section 1.1.5, a metanode contains its children in the hierarchy.

A **layout** or **drawing** is an assignment of coordinates to the nodes of a graph in some space. In this thesis, layouts are two-dimensional where edges are drawn between the nodes using straight lines. Thus, layout and

drawing will be used instead of two-dimensional layout and two-dimensional drawing. A particular graph $G = (N, E)$ is independent of its drawing or layout.

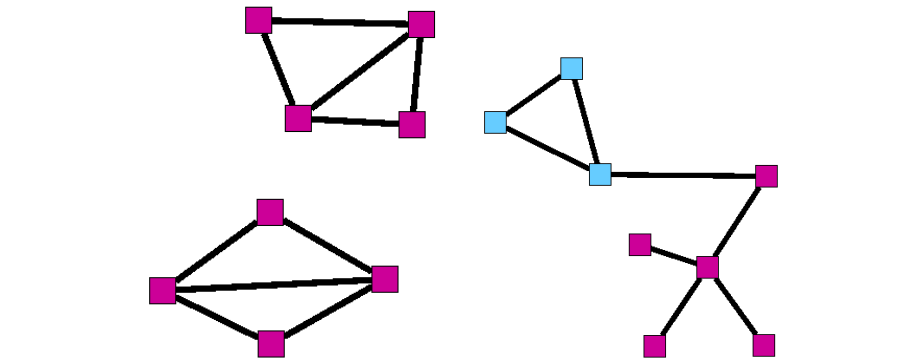
1.1.2 Connectivity Features

A **connectivity feature** is a subgraph $G' = (N', E')$ present within a graph $G = (N, E)$ where N' and E' are connected in a certain way defined by the connectivity feature type. The order in which the connectivity features are detected can influence the features that are detected. Further discussion about order of detection is presented in Chapter 3. This section introduces the types of connectivity features that algorithms exploit in our work. All the connectivity features described below, except for cluster, are described in standard algorithms textbooks [12].

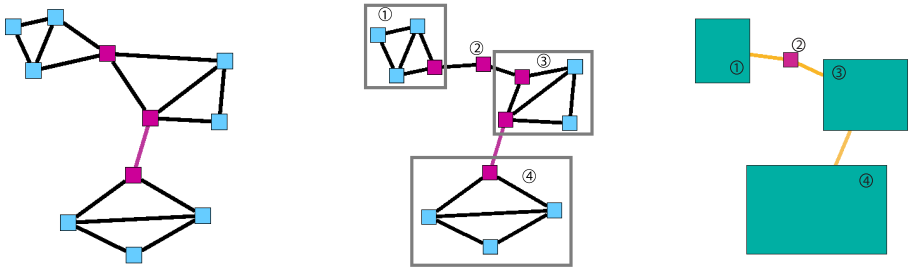
A **connected component** is a graph where there exists a path between every pair of nodes. A **disconnected graph** G has multiple connected components. A **connected graph** G has exactly one connected component. Multiple connected components are shown in Figure 1.2(a).

A **tree** is a connected graph where there does not exist a cycle in G. Trees have exactly one path between any pair of nodes in the graph. The dark purple part of Figure 1.2(b) is a tree even though the entire graph is not a tree. In this thesis, the set of trees T in a graph H is the set of connected subgraphs induced by all edges in H that do not participate in a cycle and that do not lie on a path between two biconnected components of size greater than one.

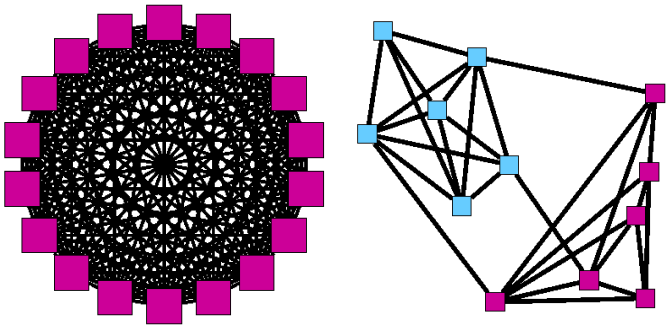
A **biconnected component** is a connected subgraph where the removal of any edge or any node and all incident edges to that node from the subgraph does not disconnect the graph. If the graph is not biconnected, it is divisible into multiple biconnected components. The research presented here uses some non-standard terms to describe nodes and edges that separate biconnected components because these terms are more evocative of how the nodes and edges appear in the visualization. A node whose removal disconnects the graph is referred to as a **bridge node**. A bridge node is an **articulation vertex** elsewhere in the literature. Bridge nodes are duplicated and placed in all the biconnected components adjacent to the bridge node. Each duplicated node is connected by an edge to the original bridge node. An edge whose removal disconnects the graph is referred to as a **bridge edge**. Bridge edges correspond to an edge incident to two articulation vertices. If a graph G^* is constructed where every biconnected component in G is replaced by a metanode, the connectivity of this graph



(a) Connected Components (b) Tree

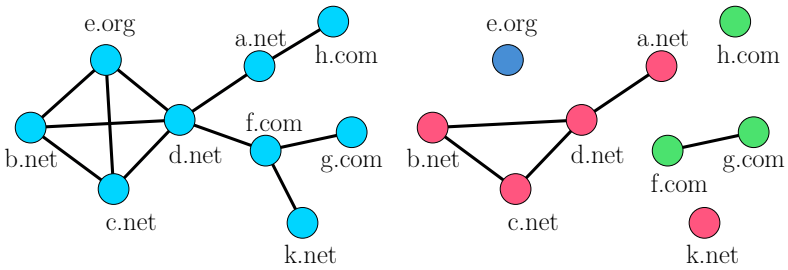


(c) Biconnected Component



(d) Complete Graph (e) Clusters

Figure 1.2: Diagram showing connectivity features. **(a)** Two connected components with no edges between them. **(b)** A tree, shown in dark purple, in context of a light blue graph. **(c)** Three biconnected components divided by bridge nodes and bridge edges in dark purple. Left is the original graph with bridge nodes. Center shows how the bridge nodes are duplicated. Right is the biconnected component tree or the G^* in the text. Numbering shows metanode to biconnected component correspondence. **(d)** A complete graph. **(e)** Two clusters, one dark purple and one blue, connected by a few edges.



(a) Input Graph (b) Attribute Features

Figure 1.3: Diagram showing attribute features. Attribute features are determined from extrinsic data associated with the nodes of the graph. **(a)** Input graph with with an attribute. **(b)** Decomposition of the input graph into five attribute features. Nodes are coloured by attribute value. Attribute features are constrained to be connected.

is always a tree. The graph G^* is called a **biconnected component tree**. Bridge nodes and edges are shown in dark purple in Figure 1.2(c). The biconnected component tree is shown in the right half of the figure. In all the chapters of this dissertation, the duplicated bridge nodes are used for drawing each biconnected component, but they do not appear in the final image presented by the system.

A **complete graph**, shown in Figure 1.2(d), is a graph where there exists an edge between every pair of nodes in the node set.

When we speak of a **cluster** as a connectivity feature, we mean a connected subgraph with small cycles connecting its nodes. There are multiple definitions for a cluster in the graph drawing literature depending on the type of algorithm used for the graph clustering. In this thesis, clusters are computed using the strength metric approach of Auber *et al.* [10]. Two clusters, one in light blue and the other in dark purple, are shown in Figure 1.2(e).

1.1.3 Attribute Features

Unlike connectivity features, attribute features are based on attributes or data associated with the nodes of the graph. Figure 1.3 shows a decomposition via attribute. In Figure 1.3(a), all nodes are labeled with a string that looks like an Internet address. A decomposition by attribute, shown in Figure 1.3(b), divides the `*.net`, `*.org`, and `*.com` nodes into connected components where all the nodes in each connected component have the same

three-letter endings to their address.

In this thesis, an attribute feature must be a connected subgraph. We could therefore view attribute features as a form of connectivity feature. However, as the decomposition via attribute is driven by an extrinsic labeling, we would like to make this distinction explicit.

1.1.4 Steerable and Interactive Systems

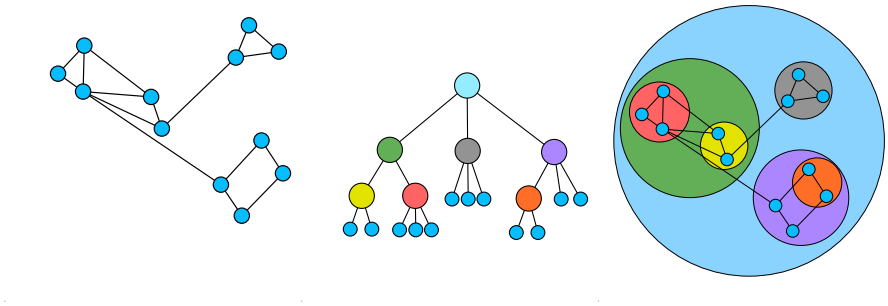
A **steerable** system allows for the user to direct the progression of a computationally expensive process. Steerable systems have existed for years in supercomputing [38]. Steerable techniques have been used to modify the progression of physical simulations [60] or visual representations of large information spaces where full representations are computationally expensive [2, 80]. A steerable technique allows a user to direct power to areas of the simulation being explored rather than applying it uniformly to a prohibitively large dataset.

An **interactive** system is one that allows user input over the course of its computation. In this thesis, steerable systems are not synonymous with interactive systems. It is true that all steerable systems allow for user interaction, but interactive systems are not necessarily steerable, as the input does not have to be used to direct computation to areas of the data being explored.

1.1.5 Graph Hierarchies

A **graph hierarchy** or **multilevel hierarchy** is created by recursively placing subgraphs into metanodes. Figure 1.4 illustrates a graph hierarchy. The original graph or input graph is presented in Figure 1.4(a). The nodes of this graph are referred to as the **leaves** of the hierarchy. A metanode contains a subgraph of leaves and other metanodes. In a graph hierarchy, a metanode is a parent of all the nodes of its metagraph or an **interior node** of the hierarchy. The root of the hierarchy behaves like an interior node for the purposes of this thesis. The hierarchy is illustrated in Figure 1.4(b) and this hierarchy is superimposed on top of the input graph in Figure 1.4(c). In the techniques presented in this thesis, graph hierarchies can be constructed by recursively detecting connectivity features in the graph or features based on attributes associated with the nodes of the graph.

If the input graph is large, a graph hierarchy is especially useful for abstracting away unnecessary details. These abstractions are defined using antichains of the hierarchy. An **antichain** A is a set of nodes in a graph



(a) Original Graph (b) Graph Hierarchy (c) Graph Layout

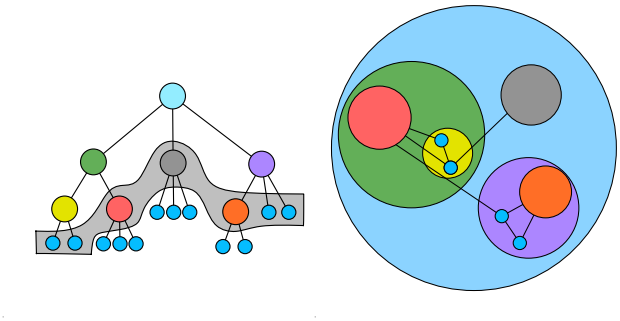
Figure 1.4: Illustration of a graph hierarchy. **(a)** Original input graph without any hierarchy. **(b)** Graph hierarchy and **(c)** graph hierarchy superimposed on top of a graph. Colours in both views show correspondences. The leaves are the small saturated blue dots. Metanodes are interior nodes of the hierarchy and appear as coloured circles that contain leaves and other metanodes. A subgraph is the subset of leaves and metanodes contained inside a metanode. The grouping of the nodes in this hierarchy is arbitrary for illustration purposes.

hierarchy such that no two nodes in A form an ancestor-descendent pair. A maximal antichain is an antichain where every path through the graph hierarchy is intersected by A exactly once. Many graph visualization systems based on graph hierarchies have used antichains for the purposes of visualization. In order to discuss these systems in an information visualization setting, we define more evocative terminology.

A **hierarchy cut** is a maximal antichain through a graph hierarchy. Occasionally, we will use **cut** as a short form of hierarchy cut. **Open metanodes** are all nodes and metanodes in the graph hierarchy above the selected antichain. These metanodes display their metagraphs in full detail. **Cut metanodes** are the elements of A in the antichain. They appear as nodes in the overview and an edge exists between them if, and only if, an edge existed in the original graph between their children. **Hidden nodes** are leaves and metanodes that are descendents of A. They are abstracted away inside the cut metanodes. A hierarchy cut is shown in Figure 1.5.

1.1.6 Feature-Based Graph Drawing

A **feature** is an subgraph we automatically detect in a larger graph that is used as a basis for hierarchy creation and graph layout. Features cannot



(a) Graph Hierarchy (b) Graph Layout

Figure 1.5: A hierarchy cut illustrated in Figure 1.4. **(a)** Graph hierarchy with hierarchy cut shown as grey curve and **(b)** graph hierarchy cut superimposed on top of a graph. Metanodes that appear on the cut are drawn opaque, hiding the subgraphs below them.

overlap. They must contain a disjoint set of nodes and edges from all other features. The features detected in a graph depend on the order of their detection. In this thesis, we only consider features based on graph connectivity or attributes. Feature-based algorithms detect features in the input graph and place them into metanodes, forming a graph hierarchy.

Low-level structures are features in $G = (N, E)$ consisting entirely of leaf nodes in a graph hierarchy context. For example, in a dataset of the primary routers on the Internet, low-level structure would be represented by the servers associated with UBC. One or several levels higher in the hierarchy is **mid-level structure**. Mid-level structure describes how low-level structure is connected. It is also the many levels of structure that may exist between this first level and high-level structure. In our computer network example, mid-level structure would describe how all the university computer networks connect in British Columbia. One more level of mid-level structure would describe how the universities in each province connect to each other. Finally, **high-level** structure is the greatest level of abstraction in the graph hierarchy. It is the subgraph contained at the root of the hierarchy. In our example, high-level structure would be the most abstract representation of the Internet available in the graph hierarchy, indicating the connectivity of computer networks all over the world.

A **feature-based** approach to graph drawing detects levels of structure in a input graph. As such, every feature-based graph drawing algorithm has two phases. The **decomposition phase** detects these levels of structure in

the input graph forming the graph hierarchy. The results of the decomposition phase can be supplied as input to the system or it can be computed by the system. Once the graph hierarchy has been created, the **drawing phase** draws the levels of structure. Either of these steps may be steerable through an interface where the user has control over how decomposition or drawing of the graph takes place.

1.2 Methodology and Empirical Evaluation

In this thesis, we are interested in demonstrating the viability of tools for the visualization of graphs acquired from real data sources. A central part of the validation of the thesis, therefore, is through the implementation of systems for graph visualization, and the comparison of those systems to the best-of-breed systems in existence on real data.

In order to evaluate the algorithms in this thesis, we employ **empirical evaluation** techniques on their implementations. Empirical evaluation techniques are frequently used to evaluate the implementation of graph drawing systems [41]. In such an evaluation, benchmark datasets [16, 28, 41, 52] used to evaluate previous work are rerun, limiting the experimenter's ability to choose the data that produces the best results. An empirical evaluation consists of the asymptotic complexity of the algorithm, quantitative timing measurements of the implementation, and qualitative discussion of images created by the system. Quantitative metrics of visual quality are also appropriate [57]. This approach is a standard one in information visualization that is used to evaluate the implementation of algorithms on real data sources. However, an empirical evaluation does not prove for a general class of graphs that the technique or algorithm is necessarily effective. Rather, an empirical evaluation provides evidence that the technique or algorithm is effective on a sample of possible input data. For all of the algorithms presented in this thesis, we present the three major components of an empirical evaluation to compare our results with existing, best-of-breed algorithms. In TopoLayout and SPF, these results are supplemented by applying some quantitative metrics.

We will frequently refer to the qualitative discussion of result images as a discussion of **visual quality**. In a discussion of visual quality, we compare and discuss the differences between generated images and the levels of structure that can be seen or cannot be seen in the results. We will also discuss **information density** in the same way. Information density refers to the area used to depict the structures in the graph compared to the amount

of free space in the drawing. A drawing with poor information density has too much or too little space between graph elements, while a drawing with good information density is compact with respect to the size of the graph. When we refer to *improved visual quality*, we mean an improvement in this qualitative sense.

In addition to asymptotic complexity numbers, quantitative timing measurements are used to demonstrate how the implementation runs on data intended for the system. These times are usually given in number of seconds, unless otherwise specified. The runs are performed on the same machine on the same data. When we refer to *faster* in the thesis, we mean faster in terms of quantitative timing numbers recorded on a test machine.

We have extended some of the quantitative metrics from a traditional single-level graph drawing to evaluate SPF as a multilevel algorithm. Specifically, we have extended edge length variance and node and metanode overlaps. These quantitative metrics are used to support the more qualitative visual structure and density arguments presented in this thesis. These two metrics are a first step towards a complete set of quantitative metrics for the evaluation of multilevel graph drawing algorithms, which remains future work. Once such metrics have been proposed, perceptual studies would need to be undertaken in order to determine the factors that most influence the readability of low-, mid-, and high-level graph structure.

In this thesis, we are interested in developing general graph drawing tools that are not tied to any specific application. In several places in this thesis, we cite possible application areas for our graph drawing approaches. We do not claim to solve these domain problems or that domain experts are interested in the problems cited. Most of the time, these scenarios are written for expository purposes to help show possible applications of the work. However, when we have confirmed a possible application area from either previous work or through conversation with a domain expert, we note this explicitly in the text.

The final goal of a visualization technique is to provide insight into the data that the user is trying to understand. We do not evaluate whether or not the technique is able to generate insight, but opt instead for the empirical evaluation of techniques and algorithms for the purpose of graph visualization. We consider this problem to be beyond the scope of this thesis as the work would require extensive ethnographic evaluation and formal user experimentation in order to validate these hypotheses. However, we do hypothesize as to the value added to graph visualization when these techniques are considered.

In the TopoLayout and SPF chapters, we hypothesize that making the

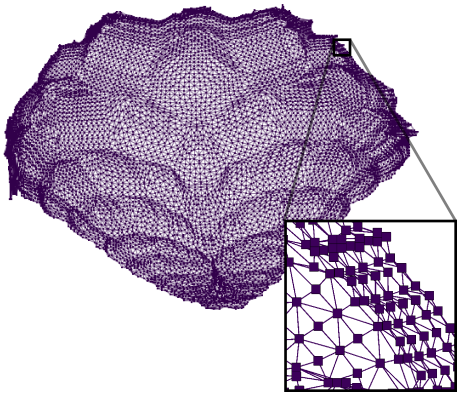


Figure 1.6: An example of how low-level structure in a graph is conveyed using insets in this thesis. The inset, present in the lower left corner, shows a close-up of the graph structure being discussed. Two lines are drawn back to a smaller box, indicating the size of the zoom area in the original drawing. These insets are never part of the drawing presented to the user of the system and are for illustration purposes in this thesis document only.

levels of connectivity structure visually apparent in the drawing will provide insight to the user about the structure of the graph. In the Grouse, GrouseFlocks, and TugGraph chapters, we hypothesize that providing steerable overviews through a graph hierarchy on top of a graph will provide insight into its structure. With GrouseFlocks, we hypothesize that viewing the raw graph from multiple perspectives provides insight into its structure. With TugGraph, we hypothesize that using hierarchy structure to model is effective in providing insight into the structure proximal to a feature.

We do not validate any of these hypotheses. As future work, we hope to apply recently developed experimental techniques [64] in order to formally evaluate whether or not these techniques provide insight for a user engaged in a particular task.

1.3 Zoomed Insets in the Thesis

In this thesis, not all elements of the graph low-, high-, and mid-level structure that we would like to discuss can be made visible in a single image. In all the systems presented in this thesis, the user has the ability to magnify

portions of the graph to see details. As it is difficult to convey zooming in a text document, insets are used to show how how certain areas of the graph appear under these levels of magnification.

Figure 1.6 gives an example of such an inset. The inset, present in the lower right corner of the figure, shows a close-up of the graph structure being discussed. Two lines are drawn back to a smaller box, indicating the size of the zoom area in the original drawing. It is important to note that these insets are never part of the image presented by the system. The insets are added to the images in this document solely for the purposes of illustration.

When movies are available, links are provided so that an interested reader can see the system browsing a few sample datasets. It should also be noted that as the chapters of this thesis progress, these insets become less prevalent in the presented figures. This indicates an improvement in our work over the years to depict many levels of structure in a single image.

1.4 Techniques and Algorithms

In order to clarify the contributions of this thesis, we need to define what is meant by technique and algorithm.

A **technique** is a class of methods for detection of, or interaction with, data. In our case, this data is a graph. Different techniques yield different visual representation or interaction methods. For example, matrix and node-link representations of graphs are different techniques, because their visual representations are different. Likewise, techniques based on graph hierarchies are different than graph drawing techniques, because they provide a different form of interaction with the graph through the opening and closing of metanodes, rather than presenting a final drawing.

An **algorithm** is a way of implementing a technique. A given technique can be implemented by many possible algorithms. For example, in graph drawing systems, there are many multilevel algorithms [34, 40, 44, 49, 76] described in Chapter 2. These algorithms are different implementations of a multilevel technique where a graph is drawn through a hierarchy of coarse graphs.

1.5 Thesis Overview and Contributions

The following section provides an outline of the remainder of this thesis.

Chapter 2 presents a summary of related work in graph drawing and information visualization pertaining to a feature-base approach to graph

drawing. Several sections of discussion present areas where feature-based approaches can improve on previous results.

Chapter 3 presents TopoLayout. TopoLayout is a multilevel, feature-based graph drawing algorithm that recursively detects the connectivity features described in the introduction, forming a graph hierarchy. Neither its decomposition nor its layout phase are steerable.

The contribution of TopoLayout is a multilevel graph drawing algorithm based on connectivity features. It also introduces a pass for reducing edge crossings in the final drawing.

Chapter 4 presents a second feature-based graph drawing algorithm, Smashing Peacocks Further or SPF. SPF is a feature-based graph drawing algorithm that is based on TopoLayout. The algorithm is tuned for graphs with tree-like structure. Neither its decomposition nor its layout phase are steerable.

The contribution of SPF is an algorithm that tunes the TopoLayout pipeline for drawing quasi-trees. Additionally, the approach extends the previous RINGS algorithm in order to make it area-aware. In the empirical evaluation of SPF, we extended some traditional graph drawing metrics to a multilevel graph drawing context in order to evaluate the high-level structure in the drawings.

Chapter 5 presents Grouse, a steerable graph hierarchy exploration system. Grouse realizes an algorithm to explore a previously constructed hierarchy of connectivity features. Grouse takes this previously constructed graph hierarchy as input, but its drawing phase is steerable.

The contribution of Grouse is an algorithm for the steerable exploration of a graph and an associated graph hierarchy of connectivity features. Additionally, a re-layout algorithm is presented that keeps nodes in the final drawing closer to their input size.

Chapter 6 presents GrouseFlocks, a system for the exploration of the space of graph hierarchies that can be constructed on top of a graph using attributes associated with the nodes of the graph. In this system, both the layout and decomposition phases are steerable.

The contribution of GrouseFlocks is a path-preserving technique for the steerable creation and exploration of graph hierarchy space based on attribute data associated with the nodes of the graph. The approach also introduces a coarsening algorithm that preserves large metanodes in the created hierarchy.

Chapter 7 presents TugGraph, a system that assists users in exploring areas of a graph adjacent to a feature. Using an initial decomposition, the systems allows users to interactively modify the graph hierarchy based on

information directly adjacent to nodes and metanodes. In this system, both layout and decomposition phases are steerable.

TugGraph contributes a technique, and algorithms implementing the technique, for exploring a region of the graph located near a feature.

Chapter 8 discusses open problems in the area of feature-based graph drawing that stem from this thesis. Finally, Chapter 9 concludes by summarizing the work and its contributions.

In this thesis, we extensively explore the area of feature-based graph visualization by developing and evaluating novel techniques and algorithms. We provide support for our claims of improved visual quality and running times of these systems through empirical evaluations against previous approaches.

Chapter 2

Previous and Related Work

In graph drawing and areas of information visualization that develop or use graph drawing research, the literature is divided into techniques that focus on finding good embeddings of the graph in two or three-dimensions and those that rely heavily on user interaction to promote understanding of the data. The former are termed **graph drawing** techniques while the latter are termed **interactive graph exploration** techniques.

Section 2.1 presents graph drawing algorithms, and Section 2.2 presents interactive graph exploration systems. Graph drawing can be viewed as a prerequisite for interactive exploration that allows graph drawing algorithms to scale to larger datasets. As such, the structure of previous work mirrors the structure of this thesis whereby graph drawing techniques are developed first followed by interactive systems that use them as a basis.

2.1 Graph Drawing

The problem of drawing general, undirected graphs is one of the oldest problems in information visualization. Force-directed approaches work on spring-electrical models [3, 16, 25, 30, 33], energy-based models [18, 46], or a combination of both [43] dominated most of graph drawing for the purposes of information visualization in the early nineties. These methods model physical systems where equilibria of those systems satisfy the aesthetic criteria of Purchase [63] and Ware *et al.* [78]. Although these approaches produced excellent results for many types of graphs, they suffer from quadratic asymptotic complexity with respect to the number of the nodes in the graph, resulting in slow performance on large graphs. Additionally, these graph drawing methods are prone to local minima, or equilibria far from satisfying the desired aesthetic criteria, when modeled as a physical system.

Several algebraic approaches, or spectral approaches, have been studied [49, 50, 73]. These approaches represent the graph as a matrix and use the eigenvectors of the matrix to produce drawings. These eigenvectors typically satisfy a placement of nodes that minimizes a particular energy

function or maximizes variance in path length. These drawing algorithms typically run many orders of magnitude faster than previous graph drawing approaches. However, a recent empirical study [41] suggests such that techniques require a mesh-like connectivity in order to produce good results. Although these algorithms have the same underpinnings as energy-based techniques, they are segmented out for expository purposes.

In order to scale force-directed approaches to larger graph sizes, multilevel techniques have been proposed. Multilevel graph drawing approaches [34, 40, 44, 49, 76] recursively apply a coarsening operator to the input graph, generating a graph hierarchy. The large input graph is at the leaf level while the coarsest approximation of the input graph is at the root. Force-directed algorithms are applied to the levels of the graph from coarsest to finest, exploiting the fact that the coarse graphs are representative of the finer ones, but cheaper to lay out. The approaches produce nice drawings of graphs larger than those of force-directed approaches, but, in general, require coarsening operators that exploit features to create the feature-based algorithms discussed in this thesis.

In addition to these approaches, some algorithms have also exploited connectivity features in the graph [58, 67] to produce drawings.

This section presents previous work in each of these areas, leading into a feature-based approach to graph visualization. Force-directed approaches are described in Section 2.1.1, matrix approaches in Section 2.1.3, multilevel approaches in Section 2.1.2, and algorithms that have exploited connectivity features in Section 2.1.4.

2.1.1 Force-Directed Approaches

Force-directed approaches rely on physical analogies in order to produce a drawing of a graph in two-dimensions. **Spring-electrical** techniques treat nodes as charges and edges as springs. The system is released and terminates when it reaches an equilibrium. **Energy-based** techniques drive the initial layout using an objective function toward drawing satisfying certain aesthetic criteria. Spring-electrical techniques are methods that rely mostly on local graph connectivity where only a node and the nodes immediately adjacent to it are considered when computing the layout. In contrast, energy-based methods endeavour to find a more global minimum, optimizing over aesthetic criteria or more complex graph connectivity such as paths. Hybrid techniques have the added advantage that they can take varying node size into account when drawing the graph.

Spring-Electrical

In spring-electrical approaches, nodes are are like-charged particles and the edges act as springs, holding the drawing together. The **repulsive force** pushes every pair of nodes in the graph apart, and the **attractive force** exerted by the edges pulls adjacent nodes together. These forces are expressed as functions $f(u, v)$ where the force f is applied by node u to node v. The system is released and the forces are applied iteratively until enough iterations have passed or the system reaches an equilibrium. Equilibrium is usually said to occur when nodes move little between one iteration and the next.

Eades [25] was the first to design a graph layout algorithm of this type. His method is a direct implementation of the concept described above, with his forces designed as follows:

$$f_{rep}(u, v) = (\frac{c_\varrho}{||p_u - p_v||^2})\overrightarrow{p_u p_v} \tag{2.1}$$

$$f_{attr}(u, v) = c_\sigma \log(\frac{||p_v - p_u||}{l})\overrightarrow{p_v p_u} \tag{2.2}$$

where c_ϱ is a repulsive force constant and c_σ is an attractive force constant. These constants can be thought of as scaling factors and are determined empirically. The ideal edge length between the two nodes is l. The positions of the nodes u and v in two-dimensions are p_u and p_v with $\overrightarrow{p_u p_v}$ being the vector from p_u to p_v. In this force model, f_{attr} can act as a repulsive force when the ratio of the distance to the ideal spring length is less than one and the logarithm becomes negative. This property is not true for subsequent spring-electrical techniques described.

In each iteration, the algorithm evaluates Equation (2.1) for every pair of nodes in the graph except those directly adjacent across an edge. It then evaluates Equation (2.2) for every pair of nodes directly adjacent across an edge. The result is stored inside an array of force vectors $F_v(t)$. Each node of N is then moved a small distance δ along the direction of the force vector. Starting from a random configuration, the process is repeated iteratively until equilibrium is reached. In Eades, adjacent nodes are excluded from repulsive force calculations to avoid oscillations about the ideal edge length. These oscillations would occur when f_{attr} in Equation (2.1) frequently changes sign. Since repulsive force calculations are required for every pair of nodes for each iteration, the algorithm has a worst-case complexity of at least $O(|N|^2)$.

Fruchterman and Reingold [33] improve on the forces of the spring-electrical model to make both forces linear in magnitude, leading to faster convergence of the system. Reusing the notation described above, the repulsive forces of Fruchterman and Reingold are described using Equations (2.3) and (2.4).

$$f_{\text{rep}}(u,v) = \frac{l^2}{||p_v - p_u||}\overrightarrow{p_u p_v} \tag{2.3}$$

$$f_{attr}(u,v) = \frac{||p_u - p_v||^2}{l}\overrightarrow{p_v p_u} \tag{2.4}$$

These forces are evaluated using a similar algorithm to that of Eades, but with some changes. First of all, the repulsive force do not exclude directly adjacent nodes. Secondly, sufficiently distant pairs of nodes are not considered in repulsive force calculations. These calculations can be omitted by embedding the graph in a grid, as shown in Figure 2.1(a), and considering only adjacent cells. The authors also present a way to constrain the drawing to a fixed drawing area using a physically based analogy. When a node encounters the boundary of the drawing area, it moves along the component of the force parallel to the boundary as shown in Figure 2.1(b). Finally, the distance a node is displaced in a given iteration is made iteration-dependent so that more drastic movements are allowed in early stages of the algorithm and are tapered in later stages.

The GEM algorithm [30] further improves the evaluation speed of the repulsive and attractive forces by allowing them to be computed using only integer arithmetic. Additionally, the forces are quadratic, providing even faster convergence, as shown in Equations (2.5) and (2.6).

$$f_{\text{rep}}(u,v) = \frac{l^2}{||p_v - p_u||^2}\overrightarrow{p_u p_v} \tag{2.5}$$

$$f_{attr}(u,v) = \frac{||p_u - p_v||^2}{l^2 \cdot \Phi(v)}\overrightarrow{p_v p_u} \tag{2.6}$$

$$f_{grav}(u,v) = \Phi(v) \cdot \gamma \cdot (\frac{\sum_{w \in N} p_w}{|N|} - p_v) \tag{2.7}$$

The value of $\Phi(v)$ for the node v is $1 + \frac{deg(v)}{2}$ and essentially slows the motion of high degree nodes in the plane and draws them toward their barycentre. The new gravitational force, shown in Equation (2.7), pulls high degree nodes towards their barycentre with γ as a force constant. This

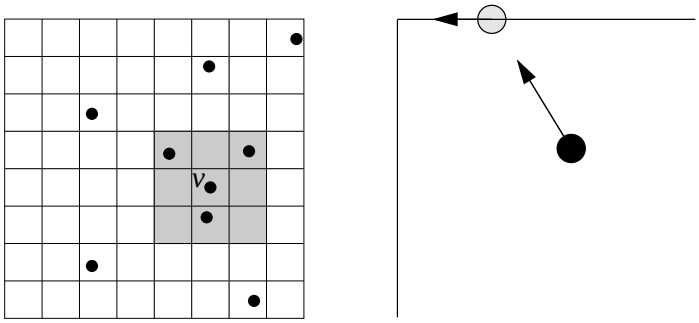


Figure 2.1: Fruchterman and Reingold [33] optimizations. The grid embedding is shown in **(a)**. The repulsive force calculations that involve the node v are only computed for nodes present in the shaded box. The drawing boundary is shown in figure **(b)**. When a node hits the boundary, it travels along the parallel component of the force as shown.

gravitational force works because the gravitational force is multiplied by $\Phi(v)$, resulting in a higher gravitational force with increasing degree of the vertex. The gravitational force serves a similar purpose as the bounding area of Fruchterman and Reingold.

Unlike all previous spring-electrical algorithms, the initialization step of GEM is not random. In Frick's implementation, nodes are introduced into the layout in breadth-first search order originating from a node near a graph center estimation. The node is placed close to its parent in the breadth-first search tree and a single iteration of the force-directed approach is applied to find the initial position of the node.

GEM improves on previous oscillation and rotation control systems for convergence. In many graphs, convergence is delayed by nodes oscillating about their optimal position or rotating in the plane around equally optimal ones. To increase the speed of convergence, the force at the current iteration is compared to the force in the previous iteration. If the forces are in opposite directions, as shown in Figure 2.2(a), oscillation is occurring and if they are perpendicular, as shown in Figure 2.2(b), rotation is occurring. In both cases, this motion does not improve the final drawing, and the force applied to the node is diminished.

In the computer networking and bioinformatics domains, domain scientists have adapted spring-electrical approaches to suit their more specific tasks and data. Usually, these graphs are very tree-like in structure with

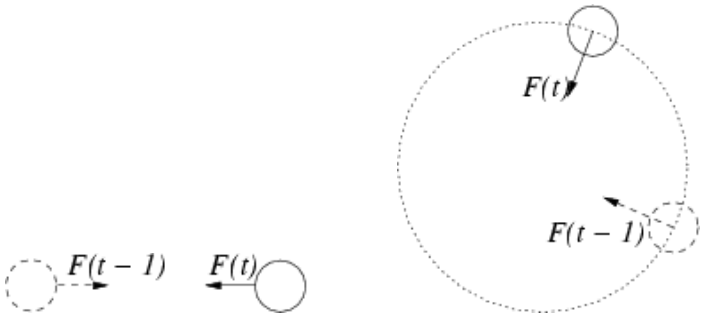


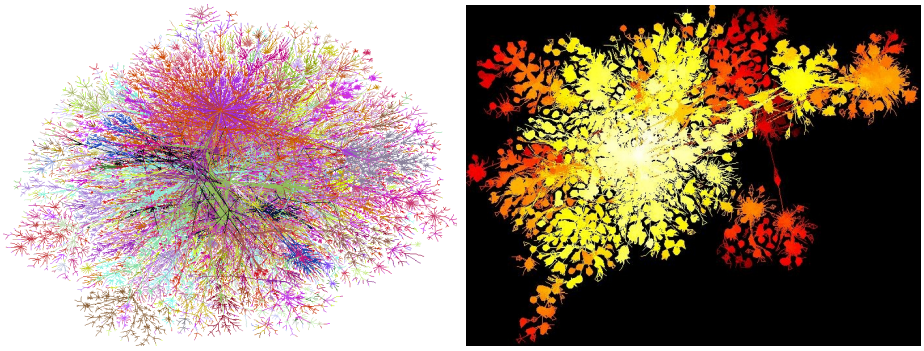
Figure 2.2: GEM oscillation and rotation control. Oscillations are dampened when forces in opposite directions as shown in **(a)**. Rotations are dampened when forces are nearly orthogonal as shown in **(b)**. The force vector $F(t)$ is the force on the node at time t.

only a small proportion of non-tree links. As such, the approaches use a spanning tree in conjunction with spring-electrical techniques to compute a final layout.

Cheswick *et al.* [16] presented a method for drawing substantial portions of the Internet router backbone. The input for the algorithm is acquired using traceroute packets from a source machine. Outgoing paths were tracked to determine the connectivity between servers.

As the focus of the paper was not on graph drawing techniques, few algorithmic details are given, but an overview of the approach is clear. To draw the graph, the authors use a modified force-directed approach with two optimizations. The first optimization eliminates sufficiently distant nodes from the repulsive force calculations. The second optimization is to use a spanning tree as a skeleton for the force-directed layout. The nodes of the test servers are laid out in the centre of the drawing. Iterations of force-directed layout are applied until this graph reaches equilibrium. Nodes are added to the layout in breadth-first order using a spanning tree centred at the traceroute packet source. The edges that connect these nodes to previously placed nodes are added. The result has been described as a "smashed peacock on a windshield" by Dave Presotto [16]. A picture of the system output supplied by the authors is shown in Figure 2.3(a).

When drawing protein homology maps, Adai *et al.* [3] use the approach of Cheswick *et al.* [16] with a few changes to adapt the approach to protein homology maps. In their system, Large Graph Layout or LGL, the spanning tree used as a skeleton is a spanning tree of minimum cost computed on an



(a) Cheswick *et al.* [16] (b) Adai *et al.* [3]

Figure 2.3: Drawings of two domain-inspired, force-directed approaches. Printed in this thesis with permission of the authors.

edge-weighted graph by a protein similarity score. The e-values measure the similarity of proteins. If two proteins are similar enough, the edge between them is included in the graph.

To draw the graph, the root vertex can be chosen arbitrarily, be user specified, or be chosen based on graph centrality. As in Cheswick *et al.* nodes are introduced into the layout in breadth-first order. LGL culls the repulsive forces of sufficiently distant nodes by embedding the graph in a grid. Two nodes share a repulsive force if they are present in the same or adjacent cells. This optimization is essentially that of Fruchterman and Reingold [33]. LGL was tested on several protein homology maps and the layouts grouped proteins into families of related function. A picture of the system output provided by the authors is shown in Figure 2.3(b).

Energy-Based

Energy-based techniques explicitly define an **objective function**. This objective function formalizes the characteristics of a good layout. The drawing algorithm drives positions of the nodes toward a local minimum or maximum of this objective function and terminates when the net energy changes little between iterations. An objective function $U(u, v)$ defines the optimal placement of the node u with respect to the node v.

Kamada and Kawai [46] is the first of these techniques and probably the most commonly in use. Their objective function drives all shortest path distances between pairs of nodes in the graph to straight lines. Let $d_G(u, v)$ be the graph-theoretic distance between nodes u and v and let l be some

ideal Euclidean length for the edge in the two-dimensional plane. Let c be a scaling factor applied to the final layout. The objective function for Kamada and Kawai is given by Equation (2.8).

$$U(u,v) = \sum_{u \in N} \sum_{v \in N} \frac{c}{d_G(u,v)} (||p_u - p_v|| - l d_G(u,v))^2 \tag{2.8}$$

Kamada and Kawai has the disadvantage that it consistently requires $O(|N|^3)$ time to compute the matrix containing all $d_G(u,v)$ values. In addition, this matrix requires $O(|N|^2)$ space as the shortest path distance matrix must be stored during layout to avoid $O(|N|^4)$ running time complexity. Although Kamada and Kawai has this limitation, heuristics can improve the running time of this approach so that it does not have to compute the full adjacency matrix.

Davidson and Harel [18] apply the process of simulated annealing to yield a nice drawing of a graph. **Simulated annealing** is an optimization method where an energy function is minimized using local improvements. Solutions of higher energy are accepted with higher probability in early iterations and with decreasing probability in later iterations to avoid local minima. In their work, this optimization function has terms to drive the drawing toward a uniform distribution of nodes, short edges, bounded layout size, and a low number of edge crossings. Weights on each of these terms are set to increase or decrease the importance of each property.

Noack [59] introduced an energy-based model for graph layout that automatically reveals cluster structure within a graph. In this approach, the LinLog energy function is minimized:

$$U(u,v) = \sum_{(u,v) \in E} ||p_u - p_v|| - \sum_{u \in N} \sum_{v \in N} \ln ||p_u - p_v|| \tag{2.9}$$

The node positions computed using the LinLog function reflect the **cut ratio** of the graph. This ratio is proportional to the number of edges that would be broken by separating the two disjoint subsets of nodes from each other. Thus, nodes participating in highly connected clusters are placed closer together, while nodes that are loosely connected are placed farther apart.

Noack further generalizes the LinLog energy model to what he calls the **r-PolyLog energy models**. The LinLog energy model is the 1-PolyLog model. Taking an energy-based view of Fruchterman and Reingold [33], it can be shown equivalent to the 3-PolyLog model. The objective function

for a general r-PolyLog model is defined as:

$$U(u,v) = \sum_{(u,v)\in E} \frac{1}{r}||p_u - p_v||^r - \sum_{(u,v)\in N\times N} \ln||p_u - p_v|| \tag{2.10}$$

As the value of $r \to \infty$, the distribution of edge lengths becomes more uniform. The author suggests that some of the r-PolyLog energy models could be applied to display different types of clusterings.

Constraint-Based Graph Drawing

The primary limitation of force-directed graph drawing approaches is that there are no guarantees of node position or separation. Constraint based approaches add hard constraints in addition to the soft constraints of energy based techniques, ensuring certain criteria are met. When these constraints are fully imposed, the graph drawing process is only limited to states where no constraints are violated. Constraint-based graph drawing techniques use quadratic programming, a numerical technique employed by constraint solvers in other areas of computer science. These relatively new techniques are based on the stress majorization [35] approach.

Dwyer *et al.* [21] present the first of these techniques. In their system, the energy-based stress majorization process is used to drive the layout of a directed graph to a good solution. A directed graph can be interpreted as a hierarchy, where levels are defined by ensuring the majority of directed edges point downwards. Constraints, in this case, are the horizontal bands imposed by the levels of the directed graph. Quadratic programming is used to ensure that nodes in the graph are constrained to the bands as defined by the hierarchy.

In further work Dwyer *et al.* [22] generalize the process of constraint-based graph drawing. In the work, they define ways to constrain nodes in graphs to fix node positions, ensure nodes are contained within a given area as in Figure 2.4, ensure orthogonal ordering, and ensure no node or cluster overlap. The algorithm is implemented in a dynamic system where these constraints can be imposed interactively at runtime as the user explores the graph. Constraints can then be imposed gradually, allowing the graph to pass through local minima energy states at the beginning, but still ensure that the hard constraints are satisfied at the end of the layout process.

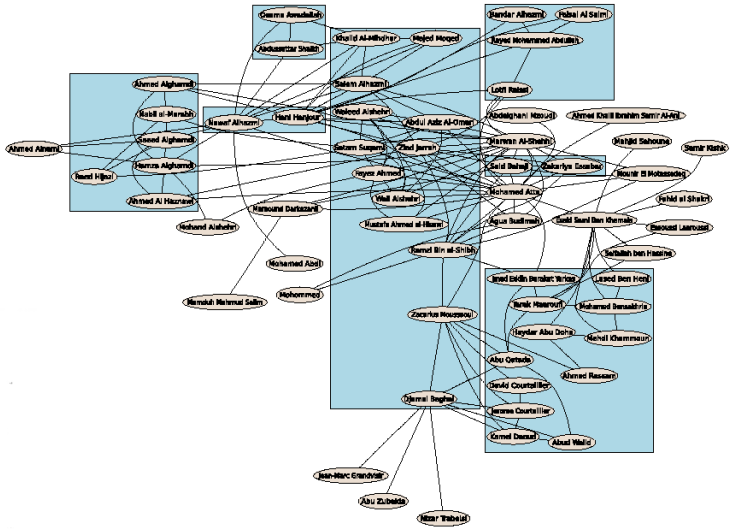


Figure 2.4: Drawing produced by IPSep-CoLa of Dwyer *et al.* [22]. Blue boxes require contained nodes to be within the specified area. Image used with permission of the author.

Area-Aware Variants

The above-described, physically-inspired algorithms assume point nodes and often perform poorly on nodes with significant area. Harel and Koren [43] introduce several ways to adapt the spring-electrical and energy-based techniques to deal with nodes of significant area or to make them **area-aware**.

In this work, the authors propose two variants of the Fruchterman and Reingold [33] algorithm to draw the graph. Their first modification employs bounding ellipses instead of bounding circles to better approximate the shape of nodes. In their second approach, they only consider the distance between the boundaries of the nodes u and v rather than the distance between the node centres. Both modifications of the Fruchterman and Reingold spring embedder have slow convergence rates to a nice drawing. The authors attribute this problem to the lack of maneuverability of nodes as they take significant area. To alleviate this problem, they suggest running regular Fruchterman and Reingold and gradually introducing the area constraints as the algorithm progresses.

The authors also propose an area-aware variant of the Kamada and Kawai [46] algorithm. To make Kamada and Kawai area-aware, they ex-

press the graph as a weighted graph where the weight on an edge is equal to the sum of an ideal edge length l and the sum of the radii of the incident nodes. Area-aware Kamada and Kawai has a faster convergence rate than area-aware Fruchterman and Reingold, but empirically, the algorithm is prone to more node-node overlaps.

In order to achieve a good layout, the authors then suggest a combined method that uses area-aware Kamada and Kawai to achieve an initial layout and then area-aware Fruchterman and Reingold to refine the drawing.

Discussion

Force-directed graph drawing approaches produce layouts of high quality on graphs of hundreds of nodes. The main disadvantage to all of the above-described methods is that they require, in worst case, $O(|N|^2)$ time per iteration to compute the repulsive forces.

Noack [59] was one of the first to suggest that uniform edge lengths for edges in a graph layout may not be ideal for understanding cluster structure in the graph. In fact, the LinLog energy function strives for non-uniform edge lengths in order to make the cluster structure in the graph more prevalent. This argument is more in line with a feature-based approach: an approach we continue to explore and expand in this thesis.

2.1.2 Multilevel Algorithms

A multilevel graph drawing algorithm circumvents the quadratic worst case complexity of force-directed graph drawing algorithms by considering a hierarchy of coarse approximations of the input graph that is used for layout. These techniques consist of two phases: a **coarsening phase** where a coarsening operator is applied recursively to the graph to construct the hierarchy of coarse graphs and a **layout phase** where the computed hierarchy is used to draw the original graph. The coarsening operator replaces subgraphs in the graph by a single metanode n. The recursive application of this operator defines a hierarchy of coarse graphs specified by metanode containment relationships. The node n is the parent node of all nodes in the subgraph it contains. Conversely, the nodes of the subgraph are the children of n. This section discusses several of examples of multilevel algorithms with their coarsening and layout phases. A drawing produced by a recent approach of this class is presented in Figure 2.5.

In Walshaw [76], the coarsening operator is an estimate of the solution to the maximal matching problem. The maximal matching problem selects

the largest independent set of edges in the graph such that no two edges are incident to the same node. The approach uses a heuristic that is not guaranteed to return the maximal set, but returns sets of edges close to the maximum. They lay out each level of the hierarchy, using the spring embedder of Fruchterman and Reingold [33], proceeding from the coarsest level of the hierarchy to finest. The location of a node at a coarser level defines the location of a subgraph at a finer level. The authors set longer edge lengths at coarser levels of the hierarchy to reduce the likelihood of subgraph overlap at finer levels.

Harel and Koren [44] recursively apply an approximate solution to the k-centres problem as a coarsening operator. 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 fact that the Euclidean distance in the layout between two nodes should be proportional to graph-theoretic distance. The algorithm draws each level of the hierarchy using Kamada and Kawai [46] and lengthens edges at higher levels of the hierarchy to deter overlapping subgraphs.

Gajer, Goodrich, and Kobourov [34] coarsen the input graph by applying a filtration to the node set. The filtration operator constructs maximal independent sets of nodes at each level i such that the graph theoretic distance between any two nodes of the subset is at least $2^{i-1} - 1$. They also draw the graph by laying out the levels of the hierarchy from coarsest to finest using the Kamada and Kawai [46] algorithm. Nodes at coarser levels define the locations of subgraphs at finer levels in the plane.

The Fast Multipole Multilevel Method, or FM^3, algorithm [40] is the first multilevel 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. Fruchterman and Reingold [33] is used to lay out the graph from coarse to fine levels. 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 empirical evaluation of FM^3 demonstrates that FM^3 yields higher visual quality results than previous work [41]. A result of FM^3 on a social network dataset is shown in Figure 2.5.

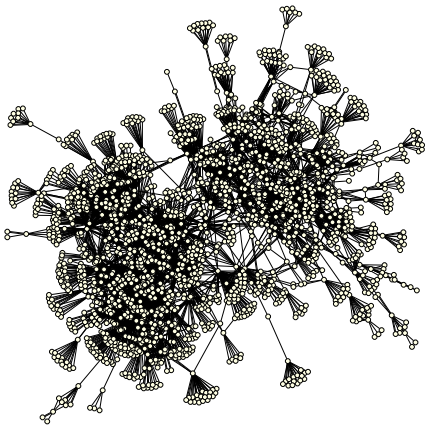


Figure 2.5: Drawing produced by the FM^3 [40] multilevel algorithm. The drawing is of a social network. Multilevel graph drawing techniques produce a final image of a large graph with all nodes and edges visible. Images reproduced with permission of the author.

Discussion

Multilevel algorithms drastically reduce the running time from hours to a matter of minutes when compared to pure force-directed techniques on graphs of thousands of nodes. In addition, the algorithms improve the quality of the final drawing as they help avoid local minima encountered in force-directed approaches. Evidence for both of these improvements is given in Hachul and Jünger's empirical study [41], where several multilevel algorithms are compared to the force-directed approach of Fruchterman and Reingold [33].

However, these heuristics treat all nodes and edges in each subgraph with the same graph drawing algorithm and, thus cannot take advantage of connectivity features found in the graph. Harel and Koren [44] assume that graph-theoretic distance should be proportional to Euclidean distance in the plane. This assumption is safe in many circumstances. However, in many tree drawings, some leaves are placed close together even though these nodes are far apart in terms of graph-theoretic distance in order to create a compact spacial layout. Similarly, for a complete graph, some nodes connected by a single edge need to be on the other side of the drawing. Gajer, Goodrich, and Kobourov [34] have a similar drawback. In their approach, the filtration does not perform well on graphs of high connectivity. Several

of the approaches presented in this thesis explicitly detect some of these cases to improve the visual quality and speed of the drawing.

A second disadvantage of these approaches is they typically use only force-directed approaches at each level of the hierarchy. Force-directed approaches can be slow in the worst case with a $O(|N|^3)$ running time. With extra information about the type of subgraph present at each level, hierarchy levels can be drawn more quickly in specific cases.

2.1.3 Matrix Approaches

There has been some work on expressing graphs as matrices and using linear algebra techniques to compute drawings. These methods produce layouts of high visual quality, but, from a recent empirical evaluation [41], it appears that these methods only work on graphs with certain a connectivity. In these matrix approaches, either the Laplacian matrix or the shortest path distance matrix representation of the graph is used to compute drawings.

Laplacian Approaches

The **Laplacian matrix** L of a graph is defined as:

$$L = D - A \tag{2.11}$$

The matrix D is the **degree matrix** of the graph where each entry d_{ii} is the degree of node i. The matrix is zero everywhere else. The matrix A is the **adjacency matrix** of the graph where an entry a_{ij} is one if and only if there exists an edge between nodes i and j. Otherwise, the entry is zero.

Although typically characterized as a energy-based method, the Tutte [73] algorithm was the first algorithm to exploit the Laplacian matrix for graph layout. Graphs drawn with Tutte must be planar, meaning they can be embedded in the plane without edge crossings. They also need to be 3-connected, meaning the removal of any two nodes does not disconnect the graph. These restrictions ensure that there exists a boundary polygon of a set of nodes M, where $M \subseteq N$. The coordinates of the nodes in M can be fixed on a polygon of $|M|$ nodes. The positions of the remaining nodes lie inside that polygon and their coordinates are determined by computing the global minimum of a system of equations defined by L. Let the remaining nodes be labeled $|M|+1, \ldots, |N|$. Let $\mathbf{x}$ and $\mathbf{y}$ be vectors where x_i and y_i for $i = 1, 2, \ldots |N|$ are the x and y coordinates of node i in the embedding plane.

The positions of the remaining nodes, $i = |M| + 1, \ldots, |N|$, are obtained by solving the two systems of equations:

$$L\mathbf{x} = 0 \tag{2.12}$$

$$L\mathbf{y} = 0 \tag{2.13}$$

The imposed restrictions ensure that there is a unique solution for all x_i and y_i with $i = 1, 2, \ldots |N|$. This solution is determined by deleting the rows $i = 1, 2, \ldots, |M|$ of L in Equations (2.12) and (2.13) as their coordinates are specified by the embedding polygon. These terms are solved, summed, and subtracted from both sides of the system of equations as the values of x_i and y_i are known.

The ACE algorithm [49], shown in Figure 2.6, solves for the eigenvectors of L associated with the smallest eigenvalues 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 via algebraic multigrid techniques and computing the eigenvectors of the coarsest matrix. The final layout is recursively computed by using the eigenvectors obtained at a coarser level of the matrix hierarchy as an estimate for the eigenvectors one level down. In this way, ACE could be considered multilevel. The computed eigenvectors are mapped to the x and y positions of the nodes to produce the final drawing. The ACE algorithm has a strong connection to the Hall energy [42] where these eigenvectors are used to lay out a set of points.

Shortest Path Distance Matrix Approaches

Instead of considering the Laplacian, the High Dimensional Embedder algorithm or HDE [50] considers part of the pairwise, shortest path matrix between nodes in the graph. The algorithm is related to a rich family of mathematical approaches that have been explored as solutions to problems ranging from flattening curved surfaces [65] to texture mapping in computer graphics [82]. 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

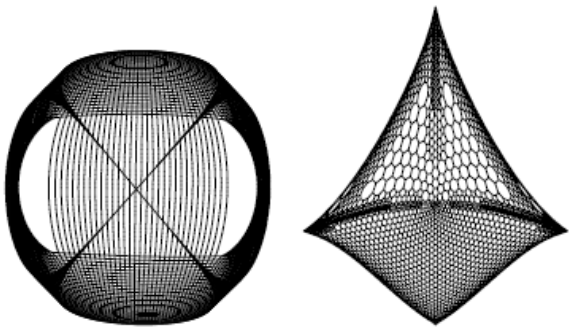


Figure 2.6: Two graphs drawn with the ACE algorithm [49]. The approach uses a hierarchy of coarse Laplacian matrices to speed up the computation of a final layout. Images reproduced with permission of the author.

scaling (MDS) maps the d-dimensional embedding down to a two or three-dimensions. Figure 2.7 shows a result of the algorithm.

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 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 has a running time of $O(d(|N|\log|N| + |E|))$ or $O(d(|N| + |E|))$ depending on whether Dijkstra's algorithm or breadth-first search is used.

Discussion

ACE and HDE are many orders of magnitude faster than the multilevel techniques presented in Section 2.1.2. On graphs of thousands of nodes, frequently they take less than one second. As Tutte needs only to solve a system of linear equations and compute a boundary polygon for the graph, it is also extremely fast.

However, these matrix-based approaches only work well on graphs with a

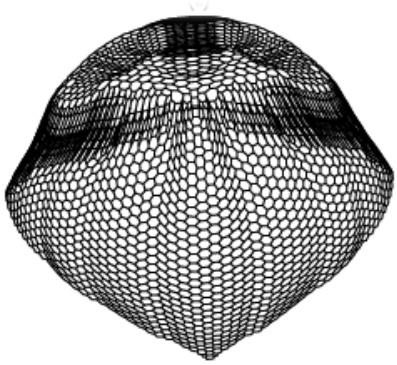


Figure 2.7: A graph drawn with the HDE algorithm [50]. The approach computes the dominant eigenvectors of the shortest path matrix and uses them for node positions. Images reproduced with permission of the author.

certain connectivity. For Tutte, this is the set of 3-connected, planar graphs. Linear time algorithms exist to transform a planar graph into a 3-connected planar graph, but additional dummy nodes need to be added. For ACE and HDE, the set of graphs is not known, but there is some evidence that suggests these approaches are also restricted to a subset of general directed graphs. A recent empirical study [41] suggests that these graphs need to have mesh-like properties. Also, Stefan Hachul, in his PhD thesis [39], presents an argument as to why HDE will not perform well on graphs with many biconnected components.

2.1.4 Connectivity Feature Approaches

Connectivity features have been exploited previously in graph drawing. Two previous approaches search for specific connectivity features in the graph at a single level, and both of these approaches use different layout algorithms depending on the type of connectivity feature detected.

One of the older approaches along these lines is the work of Sugiyama and Misue [70]. The algorithm takes as input a directed graph and a graph hierarchy on top of that directed graph. The approach draws the levels of the hierarchy in several passes, optimizing the layout using accepted directed graph drawing aethetics in the following priority order: vertex closeness, edge crossings, edge-node crossings, and edge straightness. The resulting drawing of the hierarchy is a compromise between these possibly conflicting criteria.

Six and Tollis [67] decompose the graph into biconnected components and lay out each component using a radial tree layout algorithm that is area-aware. Bridge edges appear as edges in the tree. Bridge nodes are either drawn with a component or are drawn between components. Each biconnected component is drawn with a circular layout algorithm that places all the nodes on the circumference of a circle.

Niggemann and Stein [58] describe an algorithm based on regression learning. The algorithm recursively clusters an input graph. For each subgraph in the recursive clustering, the algorithm constructs a feature vector containing statistics about the subgraph, including the number of connected components, biconnected components, and clusters found. A mapping for a query feature vector is found through a machine learning algorithm applied to a large database of graphs. The machine learning algorithm draws all example graphs with several layout algorithms and evaluates each drawing using traditional graph drawing aesthetic criteria to select the best drawing method. Although the work produces some visually convincing results, the largest graph drawn was one thousand nodes. No explicit performance numbers were given. However, drawing a full database of example graphs, evaluating each layout according to graph drawing aesthetics, and applying regression learning to the results is most likely a computationally expensive process, suggesting that this preprocessing step is a limitation.

Discussion

Some work has been completed on drawing graphs by exploiting the connectivity features they contain. Sugiyama and Misue do process a hierarchy of connectivity features for directed graphs. However, the approach lacks a decomposition phase as it requires a hierarchy of connectivity features as input. Also, the approach applies the same drawing algorithm at each level of the hierarchy. One of the primary differences between the features presented in their work and the features presented in this thesis is that different drawing algorithms are used depending on the type of connectivity feature detected.

Six and Tollis are able to exploit biconnectivity to draw large graphs and clearly display areas that are not biconnected. However, they only use circular layout when drawing each individual biconnected components and do not test for other types of connectivity features.

Although Niggemann and Stein do produce some visually convincing results, regression learning is slower when compared to efficient connectivity feature detection algorithms because connectivity feature algorithms do not

need to apply machine learning techniques to a database of example graphs.

In this thesis, we expand to a larger set of connectivity features detected and use a wider range of drawing algorithms tuned for the type of connectivity feature detected.

2.1.5 Discussion

Graph drawing is a very active area of research, and this section has covered only the algorithms most relevant to this thesis. An interested reader can find more information on this subject in the many surveys and textbooks written about the subject [19, 45, 47].

Few of the algorithms described above adhere to a feature-based approach to graph visualization, especially in multilevel graph drawing. This thesis presents two multilevel, feature-based algorithms, TopoLayout and SPF, that are presented in Chapters 3 and 4 respectively. These algorithms are graph drawing algorithms in the sense defined above: they compute a layout for the input graph and display the drawing in its entirety.

The process of graph visualization does not end with the area of graph drawing. Even if graph drawing algorithms use two-dimensions, where occlusion is less of a problem, the fact remains that as the size of the graph increases, visual quality degrades due to the number of items on the screen. Larger dataset sizes increase the difficulty of conveying full graph structure all at once. Thus, interactive techniques have been created to allow graph drawing algorithms to scale to larger dataset sizes.

In this thesis, we also explore several steerable, feature-based approaches in addition to the TopoLayout and SPF graph drawing algorithms. Foundations for these steerable techniques have been previously explored in the information visualization and graph drawing literature and are discussed in the next section.

2.2 Interactive Exploration of Graphs and Graph Hierarchies

This section discusses various interactive techniques that can be used to explore graphs. By allowing enabling user interaction, it is possible to present overviews of the data that are easier to understand and present details on demand as the user requests them. Most of these algorithms use established graph drawing techniques discussed in Section 2.1 or the many surveys presented in the discussion section.

Many interactive techniques for graph drawing exist, but this thesis is primarily focused on feature-based visualization in a multilevel context. Thus, we focus our discussion of interactive techniques around multilevel approaches. The section begins by presenting spanning tree methods, in Section 2.2.1, where the visualization uses a carefully selected spanning tree to drive the visualization process. Section 2.2.2 presents interactive visualization techniques that use attribute data to construct overviews of the data. Techniques that use multilevel hierarchies as a visualization technique are presented in Section 2.2.3. Finally, Section 2.3 discusses the limitations of previous work.

2.2.1 Spanning Tree Methods

Spanning tree methods have appeared previously in the graph drawing and information visualization literature. They draw a carefully-selected spanning tree and use it as an overview for visualization purposes. If the spanning tree is selected in a way that is familiar to the user, it can be useful for the purposes of visualization.

The H3 system [55], shown in Figure 2.8, relies on preprocessing with domain-specific knowledge to find an appropriate spanning tree for the graph. Node positions depend only on the chosen spanning tree, and the drawing of non-tree edge subsets is toggled based on user selection.

Boutin and Hascoët [14] characterize *tree-like* graphs as having non-tree links only below a threshold graph-theoretic distance. They impose this definition by filtering a potentially dense general graph by removing edges that do not fit their constraint. An interactively chosen focus node is used as the spanning tree root.

Discussion

All the above methods use filtering of graph edges to highlight structure and reduce visual clutter. However, the greatest strength of these techniques is their greatest weakness: much of the high-level structure is not shown. As a result, it is difficult to understand structures such as shortest paths because critical links completing a path may be filtered out. In some visualization tasks, a high-level overview of the entire graph is critical to understanding the data. For example, in a large computer network, users may want to highlight how subnetworks are interconnected. In systems biology, users may want to understand the connectivity between processes in the body.

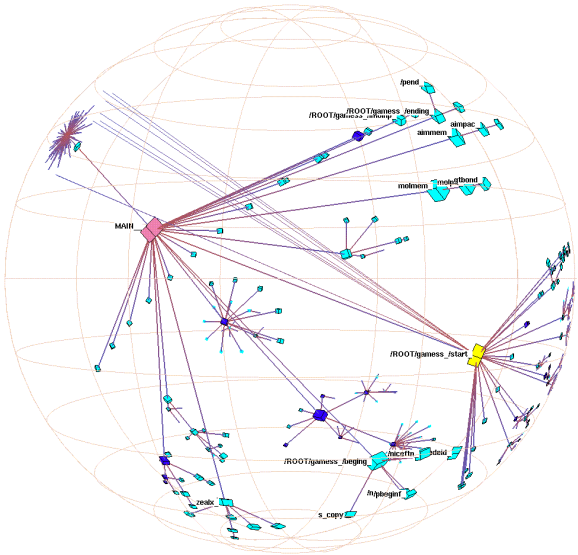


Figure 2.8: Visualization of a Fortran call graph using the H3 system [55]. H3 is a spanning tree method for graph visualization. It uses a carefully selected spanning tree and filtering on the edges of the graph in order to facilitate understanding. Image reproduced with permission of the author.

This thesis aims to provide systems that are able to highlight these high-level interactions between subgraphs.

2.2.2 Interactive Exploration Using Attribute Data

A variety of interactive graph drawing techniques have encoded spacial position using attribute data associated with the graph. These techniques do not consider graph connectivity for the purposes of accentuating the classification of nodes or edges by attribute. An exemplar system is shown in Figure 2.9.

Kosak, Marks, and Shieber [51] describe a predicate logic and a genetic algorithm approach for graph layout by attribute data. In the first approach, a set of predicate logic rules is written for the nodes in the graph, specifying clusters of nodes by label, relative positions of nodes relative to each other, and the importance of various aesthetic criteria for their placement. A Prolog interpreter applies these rules to determine a layout that is a compromise between these conflicting criteria. Their second approach is a

genetic algorithm where the fitness function encodes these predicates. The genetic algorithm is run on a massively parallel machine.

Pretorius and van Wijk [62] describe a system to map multidimensional attribute data associated with the nodes and edges of a graph to the spacial position of the nodes. In their approach, nodes are placed in the high-dimensional space defined by their attribute data. This high-dimensional drawing is mapped down to two-dimensions using principal component analysis or PCA. The resultant positions of the nodes in these drawings accentuate directions of maximal variance in the attribute being investigated, rather than the connectivity of the graph.

The PivotGraph system of Wattenberg [79] determines the spacial position of nodes using using high-dimensional attribute data. They employ a dimensionality reduction technique whereby nodes are divided into equivalence classes. Nodes are placed in the same equivalence class if and only if they have the same value for the dimension being examined. These equivalence classes can be used to influence spacial position or simplify the graph structure. When the dimensionality of the graph has been reduced to two-dimensions, the final positions of the nodes are encoded using the values of those dimensions on the x and y axis.

Semantic substrates for graph layout, introduced by Shneiderman and Aris [66], shown in Figure 2.9, allows nodes to be placed based purely on attribute data. In this approach, only categorical data is used to place nodes in regions the user specifies. Nodes are placed along horizontal and vertical axis based on attributes. Attributes can also be used to divide the nodes into separate areas of the screen. Intelligent edge filtration techniques display underlying graph connectivity to encourage understanding of the graph with the attribute as the driving feature of the visualization.

Discussion

The primary advantage of these approaches is that they elucidate structures in the attributes associated with the nodes and edges of the graph that approaches based purely on graph connectivity would miss. However, one could argue that these techniques place too little emphasis on graph connectivity for some tasks. For example, if a user wanted to see a path through a large computer network, the connectivity of the network is important and should have some influence on the spacial positioning of the nodes. Many of the chapters of this thesis demonstrate a compromise between attribute data and graph connectivity over the course of the visualization process.

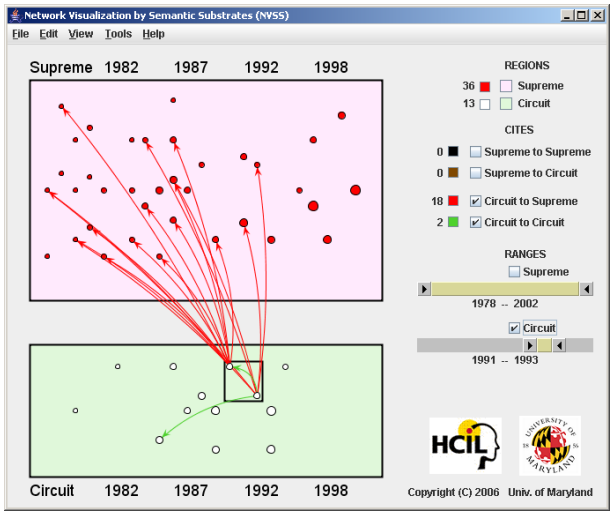


Figure 2.9: Visualization of citations in court cases using Semantic Substrates [66]. This technique is indicative of attribute-based drawings of graphs. Node positions are determined by date and interaction is used to clarify graph structure. Image reproduced with permission of the authors.

2.2.3 Graph Hierarchy Based Exploration

This section of previous work is further subdivided into three categories. Systems in the first category require existing layouts of the entire underlying graph before visualization can begin. Systems in the second category are steerable systems, or systems that lay out the graph on the fly over the course of the visualization. Systems in the third category allow some modification of the graph hierarchy in addition to steerable exploration.

Existing Layout Required

These systems are some of the oldest graph hierarchy exploration techniques in existence. All of these techniques assume a complete layout of the input graph, the leaves of the hierarchy, before visualization of the data can begin. The output of two of the more recent systems of this class are presented in Figure 2.10.

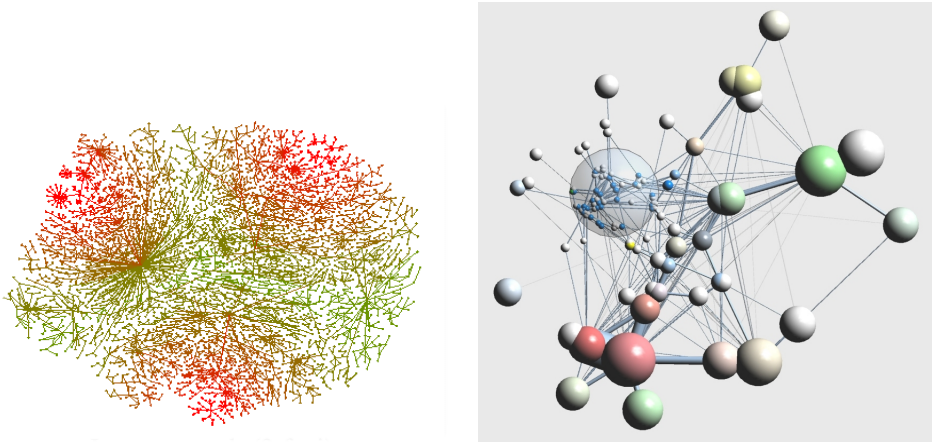
Eades and Feng [26] introduced one of the first techniques for visualizing graph hierarchies. In their approach, all levels of the graph hierarchy are extruded out onto separate planes in a three-dimensional space. Edges of the hierarchy tree are then drawn between the planes in three-dimensional

space. The user is able to interactively rotate the graph and hierarchy in order to see an interesting view. The authors provide an example of their approach with a hierarchically clustered graph where each level is drawn with the Tutte [73] algorithm.

Gansner, Koren, and North [36] introduce the concept of a topological fisheye view to a graph hierarchy. The topological fisheye displays a hybrid graph of several hierarchy levels with progressively coarser graphs displayed at increasing Euclidean distance from a focus centre. The algorithm constructs a hierarchy of coarse graphs recursively using an approximate solution to the maximal matching problem, similar to that of Walshaw [76], but also exploiting the Euclidean positions of the nodes in the layout. These node positions are exploited through a Delaunay triangulation or relative neighbourhood graph. The edges that connect two nodes of a graph-theoretic distance of two or less are considered along with the edges of the input graph. After specifying the focus centre, nodes are assigned a desired coarsening level in the hierarchy based on Euclidean distances. Inconsistencies between nodes at different levels are resolved by selecting the finer level in the hierarchy.

Abello, Kobourov, and Yusufov [1] also define a compound fisheye view for large graphs. The compound fisheye is essentially the same idea as the topological fisheye. Their approach defines a focus centre and displays fine levels of the graph hierarchy near the focus centre and coarse levels with increasing distance. Like Gansner *et al.* [36], their method assumes an initial drawing of the graph is given. The graph hierarchy is computed using graph-theoretic or Euclidean distance information. In their system, if a Euclidean distance clustering is requested they use a binary space partition of the drawing. If graph-theoretic clustering is requested they use Markov clustering [74]. Their work differs from Gansner *et al.* [36] by providing a tree map view of the cluster hierarchy next to the fisheye view. A linked view shows the level of the hierarchy that is currently being expanded by the compound fisheye view in the tree map. Through linked highlighting, the method provides a way to visualize open metanodes.

The work of van Ham and van Wijk [75] describes a way to visualize a small world graph hierarchy. In their approach, they improve the clustering spring embedder of Noack [59] to draw a clustered graph. In their variant, they perform simulated annealing on the r-PolyLog force model, choosing a more uniform force model at early stages and decreasing r in later stages to encourage clustering. The visually clustered drawing is then used to construct a hierarchy. The original nodes of the graph are the leaves and each original node is considered its own cluster initially. Recursive pairwise



(a) Gansner *et al.* [36] (b) van Ham *et al.* [75]

Figure 2.10: Two large graph visualizations that use graph hierarchies to simplify the input graph. Both of these techniques require a layout of the entire graph before visualization can begin. In **(a)**, a Topological Fisheye view of an Internet graph is presented. Red areas are shown in fine levels of detail, closer to the leaves of the hierarchy, where green areas are coarse levels of the hierarchy. In **(b)**, size encodes how coarse the metanode is. The focus centre of the visualization is the translucent bubble near the centre of the figure. Coarser levels are drawn with increasing distance from this focus centre. Images reproduced with permission of the authors.

merging of clusters is performed by considering the average Euclidean or graph-theoretic distance between elements in each pair of clusters. The hierarchy is a binary tree. It is visualized using hierarchy cuts, providing several focus areas.

Balzer and Deussen [13] apply these techniques to the domain of software engineering. Their system takes a graph and an assignment of coordinates to the nodes of the graph in three-dimensions as input. It constructs a graph hierarchy top down using spacial positions of the nodes and clustering algorithms. Level-of-detail techniques smoothly transition the views of the graph between the various levels of the graph hierarchy.

Discussion

All of these techniques have the advantage that the layout of the underlying graph is computed once, before the visualization begins. However,

this preprocessing step can be very expensive and can delay the exploration process as most graph drawing algorithms are computationally expensive. Another approach would be to draw the graph on the fly using the graph hierarchy as the user explores the data. This way, computational power is directed towards areas of the data that are of interest to the user with areas of non-interest receiving minimal computational effort. This is the approach presented in both the Grouse and GrouseFlocks systems described in chapters 5 and 6 respectively.

Steerable Exploration

In steerable systems, the layout of the graph is computed on the fly, as users explore their data. Steerable systems do not require a pre-existing layout of their graphs, allowing exploration to begin immediately. Rather, they compute layouts of subgraphs on the fly when cut metanodes are opened and become open metanodes above the hierarchy cut. By computing the layout on the fly, exploration can begin immediately without a costly layout preprocessing step. A drawing produced by a recent steerable system is presented in Figure 2.11.

Di Giacomo [20] *et al.* present a system for visualizing a graph and an associated hierarchy both created by a search engine query. In their system, a search query produces a graph: a set of documents, the nodes of the graph, and an edge exists if documents are sufficiently semantically related. The strength of the relationship is encoded with an edge weight and a clustering algorithm is recursively applied to the graph, forming a graph hierarchy. Hierarchy cuts are visualized through orthogonal graph drawing algorithms as the user explores the hierarchy.

ASK-GraphView [2], shown in Figure 2.11, is a powerful steerable system with multiple linked views. It computes hierarchies using a feature-based decomposition, detecting trees, biconnected components, and clusters. The system uses force-directed placement for all metanodes. After the base decomposition, it automatically modifies the hierarchy by imposing thresholds on hierarchy depth and the number of children in each metanode to ensure interactive visualization.

Discussion

Steerable systems allow for the visualization of a fixed graph and a fixed associated hierarchy. These systems allow hierarchy based visualization to scale to larger datasets by eliminating the expensive preprocessing layout

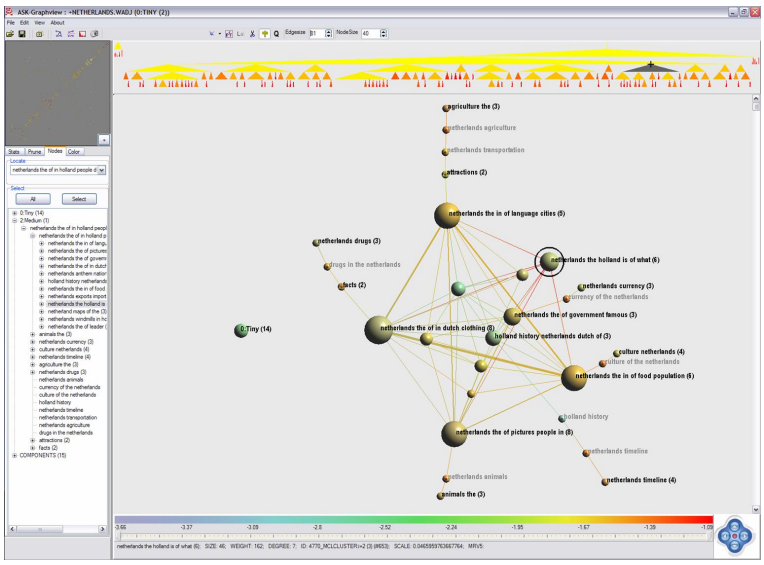


Figure 2.11: Steerable hierarchy visualization using ASK-GraphView [2]. Unlike previous visualization systems, ASK-GraphView draws the contents of metanodes as the user opens them in the visualization. Views of the graph hierarchy are visible on the top and left part of the screen. A matrix view is visible in the top left corner. Image reproduced with permission of the authors.

step. However, there is still a cost for hierarchy generation that must be paid before exploration can begin. Also, only one hierarchy that is generated by detection of connectivity features, is visualized on top of the input graph.

Current steerable graph exploration systems have a number of limitations that are addressed in this thesis. DA-TU and ASK-GraphView employ force directed algorithms that are computationally expensive on large subgraphs, limiting scalability or causing the system to resort to coarsening when it may not be necessary. Di Giacomo must generate a new input graph for each query. Generating a new input graph does not allow for the visualization of graphs where connections are given like in many software systems, computer networks, and biological systems. Grouse, in Chapter 5, partially addresses the concerns by using appropriate algorithms dependent on the connectivity of the metagraph associated with each metanode, and by allowing the visualization of a hierarchy of connectivity features on top of a static input graph.

In the next section, steerable systems are used in order to allow users to explore both a graph hierarchy and the set of graph hierarchies that can be created on top of an input graph.

User-Specified Hierarchy Editing

Three systems in the literature allow users to manually modify graph hierarchies in a limited way, mostly by manual selection. Only one of them handles the visualization of the currently created graph hierarchy in conjunction with graph hierarchy modification utilities.

The DA-TU system [27] of Huang and Eades provides an interface for the interactive visualization of graph hierarchies in two-dimensions. Metanodes can be created and destroyed, but the user must navigate to the appropriate hierarchy cut and manually select all nodes to change the hierarchy. To visualize the current graph hierarchy, the system uses a modified version of a force-directed approach that biases the hierarchy cut towards its hierarchy structure. However, the force-directed algorithm must be applied to the entire cut, rather than to only the sections of the hierarchy that have changed. This approach does not scale to large graph sizes.

The work of Auber and Jourdan [11] supports interactive hierarchy editing. The paper's primary focus is on fast algorithms for replacing subgraphs with metanodes. The algorithm executes in linear time with respect to the number of nodes and edges in the graph with constant memory requirements. Metanodes can be formed by manual selection by the user. The system does not provide steerable hierarchy exploration by progressively adjusting the

layout after the graph hierarchy has been modified, so navigation to understand the new hierarchy is not supported. We use the work of Auber and Jourdan in GrouseFlocks presented in Chapter 6.

The Clovis system [61] supports interactive clustering of an input graph based on querying the attribute values associated with the nodes and edges of the input graph. The graph hierarchy is superimposed on the input graph using containment. Attribute based queries can also affect any aspect of the appearance of a node or edge except position. The entire graph is shown at all times and clusters of nodes are shown using overlays. The user controls when clusters are redrawn manually by selecting the appropriate cluster.

Discussion

These tools provide good ways of modifying or creating an existing hierarchy by either manual selection or attributes on the nodes and edges of the graph. However, these systems do not integrate graph hierarchy exploration in a scalable manner into their approaches. This is a limitation since the user has little assistance in understanding the hierarchy they have created, making subsequent modification steps difficult. This limitation is addressed in the GrouseFlocks system presented in Chapter 6.

2.3 Discussion

Through user interaction, interactive graph exploration systems allow graph drawing algorithms to scale to larger datasets. The approaches dynamically filter the input dataset, whether it be based on a carefully selected spanning tree, attribute data associated with the graph, or through a multilevel hierarchy imposed on the data. By filtering, areas of high-level and mid-level structure are visible while low-level structure can be acquired on demand.

The steerable, feature-based approaches presented in this thesis exploit graph hierarchies and attribute data for visualization. In Section 2.2.2, none of the techniques presented are based on graph hierarchies. It would be advantageous for us to consider graph hierarchies for the purposes of visualizing attributes in a feature-based context, especially when elucidating high and mid-level structure. Conversely, in Section 2.2.3, these approaches primarily focus on graph connectivity and consider attribute data in a very limited way. An explicit focus on attribute data may lead to more insightful methods for visualizing an input graph.

Additionally, it should be noted that all of the approaches that exploit graph hierarchies for visualization, except ASK-GraphView [2], require ei-

ther a layout or hierarchy precomputation or both before visualization can begin. Steerable exploration of a graph hierarchy has been done with ASK-GraphView and DA-TU, but the technique of steerable hierarchy creation in conjunction with steerable hierarchy exploration has not been explored. It would be advantageous to allow the user to steerably explore a graph hierarchy and even the space of graph hierarchies with minimal precomputation. Chapter 5 describes a method for steerable exploration of a graph hierarchy of connectivity features. Chapter 6 explores the possibility of steerably exploring the many possible graph hierarchies that can be constructed on top of a graph where nodes have attributes associated with them.

Chapter 3

TopoLayout: Multilevel Graph Layout by Connectivity Features

TopoLayout is a multilevel, feature-based algorithm. The features it detects are the connectivity features as defined in Section 1.1.2. Previous multilevel algorithms exploit force-directed approaches almost exclusively and exploit few types of connectivity features. In many situations, once a connectivity feature has been identified, an algorithm that can produce drawings of a higher visual quality with lower asymptotic complexity may be used to draw the feature. Our motivation was to capitalize on these situations in a multilevel context.

We implemented a system that realizes the TopoLayout algorithm. Its decomposition phase is is a set of connectivity feature detection algorithms. The phase is computed offline and the connectivity feature detection algorithms are applied in a fixed order recursively. The drawing of the graph is also computed offline. It computes a final layout for the graph through one pass of the hierarchy as the multilevel algorithms do in Section 2.1.2. Previous multilevel algorithms [34, 40, 44, 49, 76] used edge contraction based on collections of nodes of distance one or two from each other. The TopoLayout approach detects connectivity features that can be drawn with specific graph drawing algorithms.

The contribution of TopoLayout is a multilevel graph drawing algorithm based on connectivity features. It also introduces a pass for reducing edge crossings in the final drawing. We implemented TopoLayout and compared it empirically to previous systems. TopoLayout was published in an InfoVis 2005 poster [4] and in a *Transactions on Visualization and Computer Graphics* article [7]. An implementation of the algorithm is available online[1].

This chapter is structured as follows. Section 3.1 describes an overview of TopoLayout and how the four phases described above interact. Section 3.1.1

[1] `http://www.cs.ubc.ca/labs/imager/tr/2006/Archambault_TopoLayout_TVCG/`

describes the decomposition phase of the algorithm and how it is recursively applied to produce a graph hierarchy of connectivity features. The pass that determines a final position for the nodes in the graph is described in Section 3.1.2. Algorithmic complexity numbers for all algorithm phases are given in Section 3.2. An empirical evaluation of TopoLayout against other graph drawing algorithms is given in Section 3.3 and the results of the evaluation are discussed in Section 3.4 and 3.5. A discussion of algorithm robustness is presented in Section 3.6. Finally, Section 3.7 discusses the contributions of TopoLayout and its future directions.

3.1 Algorithm Overview

The TopoLayout framework consists of four main phases as shown in Figure 3.1. The **decomposition** phase is the same as the coarsening operator of multilevel 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. The latter three phases are executed in that order during a single postorder traversal of the graph hierarchy.

Many of the algorithms used in these phases are directly drawn from previous work, but two are novel. In the feature layout phase, a new weighting scheme for HDE [50] is devised. The crossing reduction phase is new to multilevel algorithms and is a novel algorithm. All other algorithms are straightforward applications of the literature.

3.1.1 Decomposition

The decomposition phase consists of a series of connectivity feature detection algorithms that are applied to the input graph. Upon detection of a connectivity feature, the feature is collapsed into a single node. The process is applied recursively to the graph, constructing the feature hierarchy.

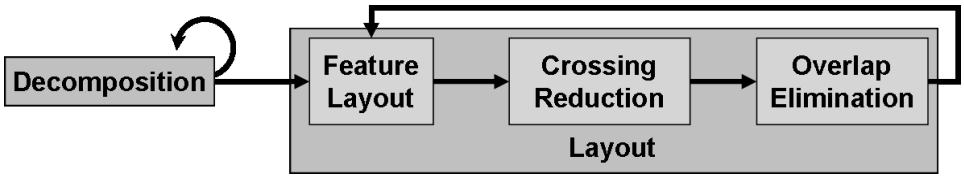


Figure 3.1: TopoLayout algorithm phases. Decomposition is applied first, recursively generating a graph hierarchy. Feature layout draws each each subgraph in the graph hierarchy and crossing reduction and overlap elimination improve drawing quality. The latter three phases are applied recursively in one postorder traversal of the graph hierarchy.

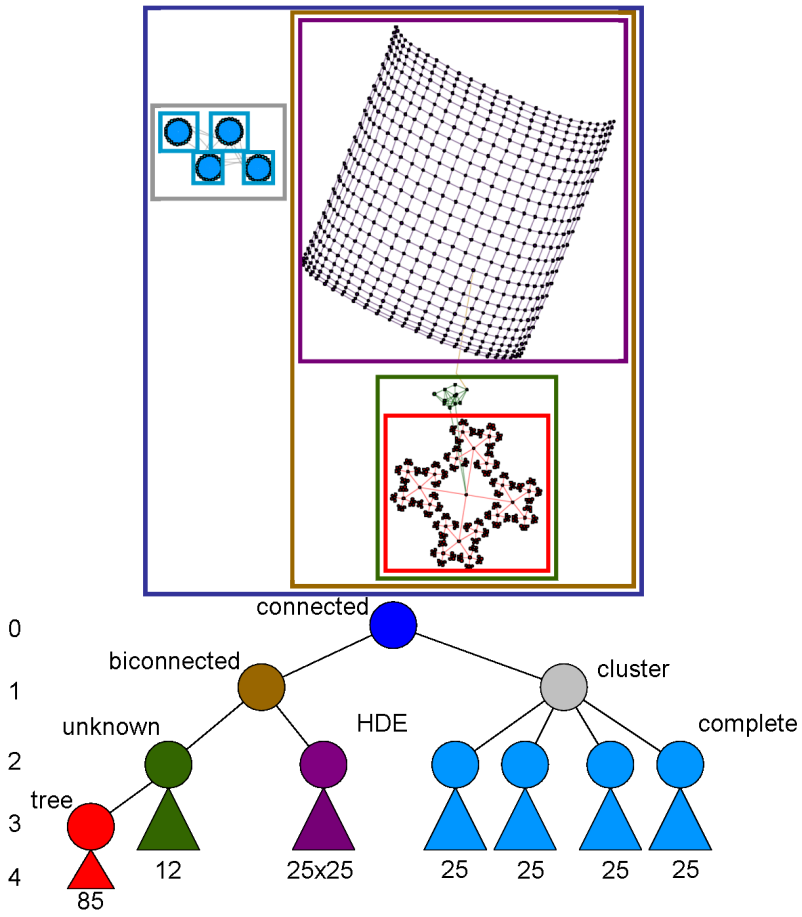


Figure 3.2: Feature hierarchy after decomposition, with connectivity feature type encoded by colour. The graph is constructed by hand to show all types of connectivity features exploited by the algorithm. **Top**: Layout annotated with bounding boxes to show hierarchy structure: metanodes encompass the subgraphs of their children. **Bottom**: Diagram of feature hierarchy, with levels enumerated and nodes labeled by feature type. Large sets of hierarchy leaves are replaced with triangles labeled by the number of leaves they represent.

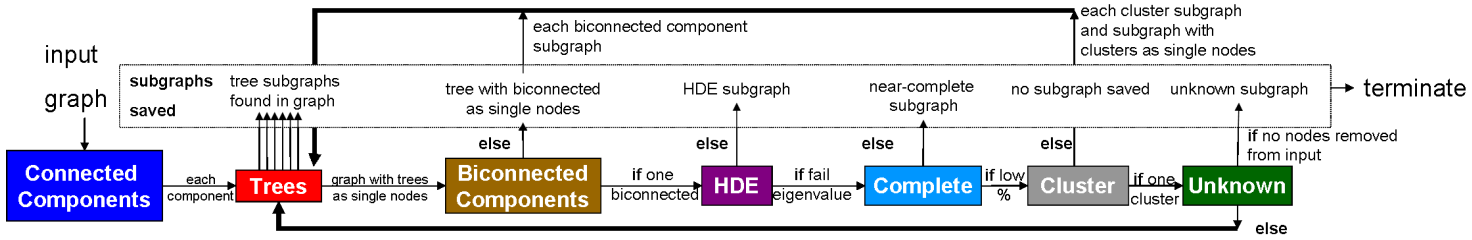


Figure 3.3: Decomposition phase for TopoLayout. Detection algorithms in boxes coloured by feature type as in Figure 3.2. If a clause on a horizontal is true, the algorithm transitions along the horizontal arrow. Otherwise, the algorithm follows the vertical arrow to save some subgraphs and recursively decompose others. Bold arrows indicate the recursive cases.

Decomposition Algorithm

The decomposition phase recursively constructs the feature hierarchy. An example feature hierarchy is shown in Figure 3.2. Figure 3.3 describes the decomposition algorithm as a box diagram along with a colour scheme. For each box in the diagram, the algorithm is presented with a subgraph of the original input graph that is decomposed into one or more metanodes.

The first step of the decomposition phase replaces each connected component with a metanode. 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. In the box diagram, connected component detection is dark blue.

To detect connected components, the decomposition phase uses a standard algorithm in the literature that executes a series of depth-first searches to compute spanning trees for each component [12]. The approach traverses the list of nodes in the graph and performs depth-first search whenever an unmarked node is encountered. The depth-first search marks nodes as they are visited. As each node is visited at most twice and each edge at most once, the algorithm works in $O(|N| + |E|)$ time.

Next, the algorithm segments out the trees present in each connected component, using a special case of graph peeling [2]. Each tree is saved as a subgraph and replaced by a single metanode. Tree detection is red in the box diagram. The graph with all trees removed and replaced by metanodes is passed to biconnected component detection.

To detect trees, the algorithm begins 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, the algorithm performs a depth-first search. When the algorithm visits a node of degree one, it is removed and the depth-first search continues. 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.

The next phase of decomposition detects biconnected components. 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 metanodes is saved as a subgraph. Biconnected component detection is tan in the box diagram. This subgraph must be a tree as explained in Section 3.1.2. If only one biconnected component is present, it is passed to HDE detection.

A good description of a standard biconnected component detection algorithm is also given by Baase and Van Gelder [12]. 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.

If a layout of the subgraph with HDE has the properties described in the next paragraph, it is saved and area-aware HDE is used for the subgraph in the final layout. HDE component detection is purple in the box diagram. If the layout does not have these properties, the graph is passed to complete detection.

To determine if HDE is a suitable layout, an algorithm analyzes 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 [53]. The algorithm determines if there is enough variance in the data, or if its largest eigenvalue is above a minimum threshold value. In the TopoLayout system, a value of 100 was determined empirically. This minimum variance is required as some projections of low variance place many of the nodes on top of each other. Next, the algorithm compares the percentage of variance accounted 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, HDE is used 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, the HDE detector uses a breadth-first search and runs in $O(d(|N| + |E|))$ time.

If a subgraph is complete, it is saved. This phase is not a clique detector, but simply checks to see that all N^2 edges are present in the subgraph. It does so by computing the square of the number of nodes and comparing it to the number of edges. This phase is cyan in the box diagram. If the subgraph is not a clique, it is passed to cluster detection.

Complete graphs are detected by taking the ratio of the number of edges in the graph to the number of possible edges given the number of nodes. This method could easily be adapted to detect near-complete graphs by considering a threshold below 100%. If the number of nodes and edges is known, the ratio is computed in $O(1)$ time.

The next phase of decomposition detects clusters in the graph. 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 that results from replacing each cluster with a metanode. The cluster detection phase is silver in the box diagram. If there is only one cluster, the subgraph is labeled unknown.

Clusters are detected using an approach based on the strength metric described by Auber *et al.* [10]. This approach 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, the algorithm partitions 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 involving (u, v) is the number of elements in $W(u, v)$. The algorithm determines 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. Thus, the algorithm detects clusters in $O(r|E|)$. For near-complete graphs, the performance of the algorithm would degrade to $O(|N|^3)$, but, in practice, the algorithm can be run on large graphs.

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. Unknown components are green in the box diagram.

We experimented with several orderings of the decomposition algorithms. Our rationale for applying the detection algorithms in the order presented in Figure 3.3 is as follows. Connected components of the graph should be detected first, since if there are multiple components, they can be laid 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 further decomposition of the graph using strength clustering, HDE is checked for an appropriate algorithm for layout. As cluster detection is the most expensive part of the decomposition phase and the HDE check is quick, it makes sense to check if HDE would work well on this subgraph. Finally, cluster detection provides a partition of the graph into highly connected subgraphs when more meaningful connectivity features cannot be found.

```
layout (subgraph s)
  for all metanodes c ∈ s
    c.size ←boundingBox (layout (c.subgraph));
  end for
  layOutFeature (s);
  reduceCrossings (s);
  eliminateOverlaps (s);
```

Algorithm 3.1: Pseudocode for the feature layout phase.

3.1.2 Layout

During the feature layout phase of level i of a hierarchy, the features at level $i+1$ contained by all the metanodes at level i must be drawn first to determine the screen-space bounds of the metanode. 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 3.1, draws the connectivity feature at level i using an appropriate layout algorithm, rotates metanodes 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. The layout phase uses four types of layout algorithms: tree, circular, HDE, and force-directed. This section also describes passes to reduce the number of node-edge and edge-edge crossings and to eliminate all node-node overlaps.

Area-Aware Tree Layout

In TopoLayout, area-aware tree drawing algorithms are used to draw both trees and biconnected component trees. The algorithm draws biconnected component trees in a way to highlight bridge nodes and edges in the graph. Bridge edges appear as edges in the tree between two components. Bridge nodes are drawn with one of the methods suggested by Six and Tollis [67]. The algorithm places the bridge node directly between the two components.

TopoLayout can use any tree layout algorithm that is area-aware for drawing. The algorithm uses the bubble tree algorithm [37] for trees of low depth and high branching factor and an area-aware version of the Walker algorithm [15] 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.

Area-Aware Circular Layout

Circular layout is used to 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.

Area-Aware HDE

The area-aware HDE algorithm is used to lay out subgraphs found by the HDE detector. Area-aware HDE is the standard HDE approach [50] with weighted edges where the eigenvectors of the shortest path matrix are computed over d iterations. 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.

Area-Aware GEM

When no other algorithm is applicable, area-aware GEM is used. This algorithm is similar to the algorithms developed by Harel and Koren [43] who adapted Fruchterman-Reingold [33], Kamada-Kawai [46], and combinations of these algorithms. Area-aware GEM is a modified version of the GEM algorithm [30] 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:

$$f_{\text{repulsive}}(n_i, n_j) = \frac{l + \lceil \mathbf{r_i} + \mathbf{r_j} \rceil}{\|p_i - p_j\|^2}(p_i - p_j) \tag{3.1}$$

$$f_{\text{attractive}}(n_i, n_j) = \frac{\|p_i - p_j\|^2}{l + \lceil \mathbf{r_i} + \mathbf{r_j} \rceil}(p_j - p_i) \tag{3.2}$$

The bold terms in (3.1) and (3.2) are terms added to make GEM area-aware. The ceiling of the sum of the radii is taken so that the forces are still

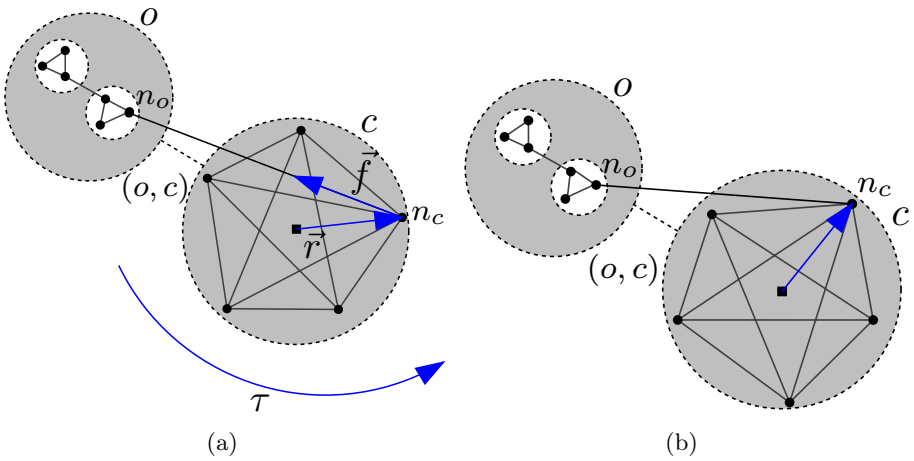


Figure 3.4: Reducing crossings with torque. **(a)** Computing the torsional force τ on c exerted by the edge (n_o, n_c). **(b)** Applying τ results to rotate c. Dashed nodes and edges are metanodes and metaedges. Solid nodes are leaves in the hierarchy. The square box is the centre of node c.

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

Crossing Reduction

Our crossing reduction phase is a heuristic that rotates metanodes in the hierarchy, reducing crossings of edges in the original graph connecting nodes in different subgraphs as shown in Figure 3.4. The heuristic does not guarantee a drawing free of node-edge or edge-edge crossings, but it reduces their number and also shortens edge length between subgraphs. The approach is similar to that of Symeonidis and Tollis [71] 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 [46] in graph layout. In contrast, the approach presented in this chapter is similar to GEM [30] and includes oscillation control.

Let o and c be metanodes in a subgraph at level i of the graph hierarchy. Let n_o and n_c be leaves in the graph hierarchy. The algorithm uses 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 metanodes, each with their own relative coordinate frames. For the moment, assume the locations of the nodes n_o and n_c are known in the coordinate frame of the subgraph at level i. We 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 (3.3).

$$\tau = \frac{\pi}{2}\mathrm{sg}(\vec{r} \times \vec{f})(\vec{r} \cdot \vec{f}) \tag{3.3}$$

Analogous to that of force-directed graph drawing techniques, the solution presented here is an incremental approach. The average value of τ is computed for all edges in the list of edges contained in the metaedge (o, c). The process is repeated, computing an average τ for each metanode in the subgraph containing o and c, using their incident metaedges. Once the average τ is computed for all metanodes in the subgraph, it is applied to the cumulative rotation of each metanode.

The approach presented here for damping oscillations around equally good orientations is analogous to the approach of GEM [30] for damping oscillations around equally good positions in graph layout. The algorithm stores the torque for each metanode 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, the algorithm is causing nodes to oscillate around an optimal orientation, and a damping factor is applied. Currently, this factor is the fraction of completed iterations to the N_i iterations that will be executed, where N_i is the number of metanodes 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 metanode that contains it. This information can be saved in the decomposition phase with no asymptotic runtime penalty when the algorithm constructs metanodes. Each metaedge 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. The algorithm traverses 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 $O(LN_i|E|)$.

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 the final layout, the algorithm performs a pass to eliminate these overlaps.

Several algorithms were experimented with to reduce or eliminate node overlaps in the drawing. In all cases, overlap reduction was tried 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. The former approach was chosen, because a single pass on the final drawing causes overlap between connectivity features.

First, the naive approach of considering every pair of nodes was tested 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.

The Cluster Buster algorithm of Lyons *et al.* [54] was implemented. This algorithm computes the Voronoi diagram of the nodes in the graph 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.

The best results were obtained from implementing the fast node overlap removal algorithm without Lagrange multipliers [23, 24]. 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.

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

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	$O(N_i \log N_i)$

Figure 3.5: Time complexity of TopoLayout framework components, for each hierarchical level.

axis aligned nodes, the axis aligned bounding box of the rotated metanode is computed.

3.2 Algorithm Complexity

The worst-case complexity of TopoLayout is $O(|N|^3)$ if no connectivity features are found and area-aware GEM is used. However, in practice, the algorithm typically performs better than this worst case.

Figure 3.5 shows the time complexity of the algorithms in TopoLayout. The number of operations performed on each subgraph of the hierarchy is described in the following way: 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. The value of d is fifty in this work. The value of L is the number of levels the algorithm must traverse up the hierarchy to compute the level i positions of n_o and n_c when computing torques.

3.3 Empirical Evaluation

The TopoLayout framework was implemented using the Tulip [9] graph visualization system, and we have tested it against other multilevel 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 multilevel algorithms were tested against TopoLayout. The code for GRIP[2], ACE[3], and HDE[4] was available online and was incorporated into the Tulip framework. Stefan Hachul kindly supplied the FM[3] code. The code was also incorporated into Tulip for testing. Harel and Koren's multilevel approach [44] was not tested. The source code for this implementation was unavailable. The observed running times and visual quality results were the same as those presented in Hachul and Jünger empirical study [41]. The benchmark datasets used in this empirical evaluation have been subsequently used in other empirical evaluations of graph drawing systems in the literature [31].

TopoLayout automatically colours connectivity features in the graph, using the scheme defined in Section 3.1.1. Since the other graph drawing algorithms do not detect connectivity features automatically, colouring sections of the graph using connectivity features is a fair comparison and demonstrates another advantage of our approach.

Our experiment was divided into two phases. *Synthetic Data* primarily consisted of benchmark datasets taken from the graph drawing literature. These preliminary tests provided a baseline for the comparison of multilevel algorithms. *Real World Data* mostly consisted of datasets deemed real world in previous empirical evaluations. The empirical evaluation also considers two additional datasets that came from real world data sources.

3.3.1 Synthetic Data

All but one of the synthetic graphs came from the Hachul and Jünger empirical evaluation [41] of multilevel algorithms. All types of graphs in the evaluation were used at the middle size of the three sizes tested. `Crack` is a standard graph drawing dataset, part of the Walshaw Graph Partition Archive[5]. It was categorized as a *real world* graph in their study. The

[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-ary`, `Snowflake`, `Spider`, and `Flower` datasets are each of the medium sized *challenging artificial* graphs of their study. The `6-ary` 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 that 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 tested, `bi_walsh`, did not appear in the Hachul and Jünger empirical evaluation. 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 provide some evidence that TopoLayout is able to segment out the mesh-like components and draw each of them with HDE. Also, it provides some evidence that other multilevel algorithms, including HDE and ACE, have difficulty drawing this dataset. The results of this part of the empirical evaluation are shown in Figures 3.6 through 3.11. Several zoomed insets are included with each drawing until nodes are visible at a given size.

3.3.2 Real World Data

In addition to synthetic data, data considered real world in other empirical evaluations was tested. The first three datasets are *challenging real world* graphs in Hachul and Jünger's empirical evaluation [41]. The `ug_380` and `dg_1087` graphs are from the *AT&T Graph Library*[6]. 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, two more were added. `UBC` is the hyperlink structure of the department of computer science at the University of British Columbia's web site acquired using a breadth-first search cut off at about 40,000 nodes. `IMDB 1999` is a subset of the Internet Movie Database[7]. 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*. This actor was chosen because he has a relatively low branching factor in that year. The results of this part of the empirical evaluation are shown in Figures 3.12 through 3.16. Several zoomed insets are included with each drawing until nodes are visible at a given size.

[6] `www.graphdrawing.org`
[7] `www.imdb.com`

3.4 Discussion of Empirical Evaluation Results

Since most of the data for these tests came from the Hachul and Jünger empirical evaluation [41], the results of this evaluation were compared with their findings. The evaluation reproduced their results except for the few differences indicated. 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, only two-dimensional layouts are used. Places where this fact makes a major difference between the results of the two studies when the drawings of each of the datasets are discussed.

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 that are *mesh-like* in structure. As a result of this finding, the remainder of the evaluation focuses on GRIP, FM^3, and TopoLayout and will only show the results for these three algorithms on the real world data.

3.4.1 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 few 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. The visual quality is assessed by presenting side-by-side comparisons of drawings produced by TopoLayout and its competitors.

`Crack`

All three algorithms produced drawings of similar visual quality for this mesh dataset as shown in Figure 3.6. This result differs from the Hachul and Jünger empirical evaluation in that GRIP does not have any folds in the layout.

`6-ary`

The results on `6-ary` are shown in Figure 3.7. On this 6-ary tree of depth five, TopoLayout produced the drawing that clarified the most symmetry,

followed by FM^3, and GRIP. In the TopoLayout drawing, both high-level and low-level structure of the tree are depicted as the tree is 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 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.

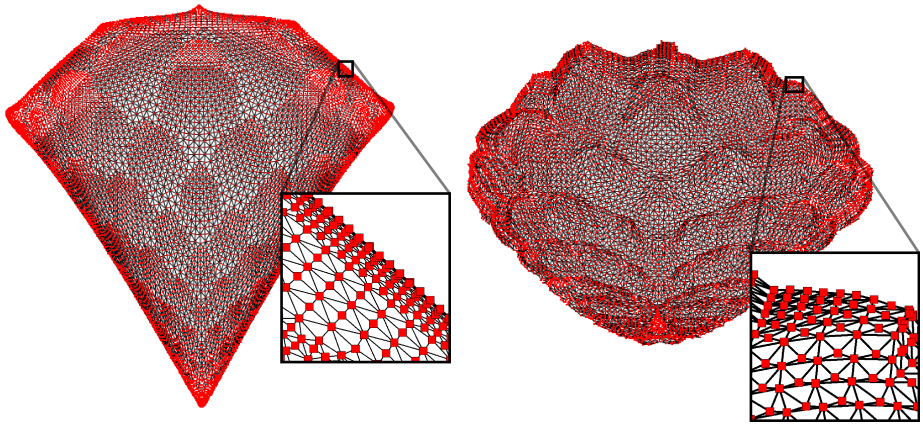
`Snowflake`

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. The results on `Snowflake` are shown in Figure 3.8. 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.

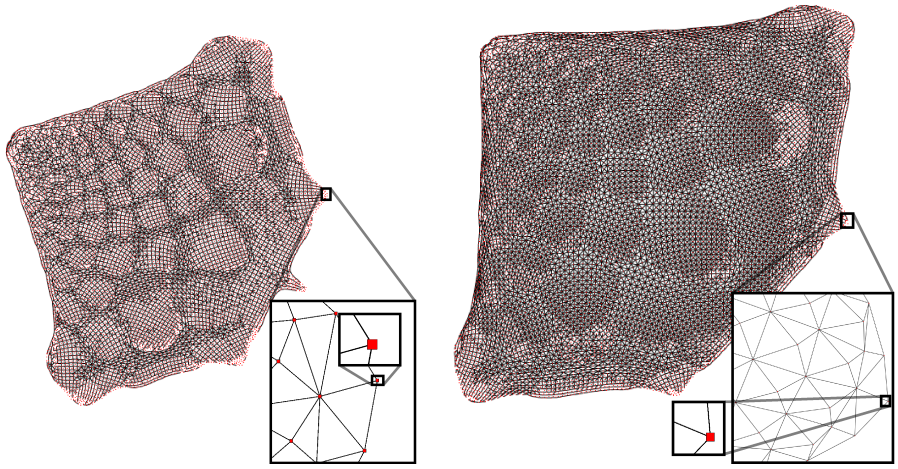
`Spider`

All three algorithms drew this dataset with a similar level of visual quality as shown in Figure 3.9, but this level of visual quality was not very high for any result. The high-level structures of the well-connected head and the eight long paths or legs are visible in two of the 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.

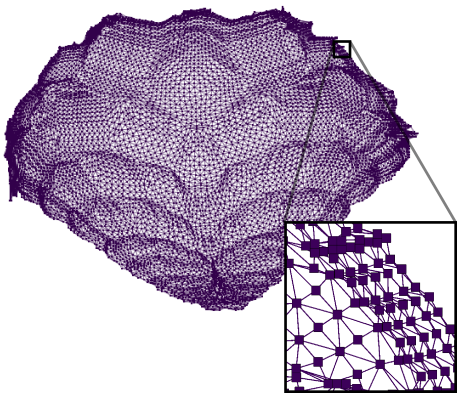
The ACE layout has such significant overlap of nodes that they are difficult to distinguish from each other in the insets of Figure 3.9(a). In the cores of Figures 3.9(c) and 3.9(e) the nodes are placed side by side, leading to long red and green boxes respectively.



(a) ACE, 0.35s (b) HDE, 0.14s

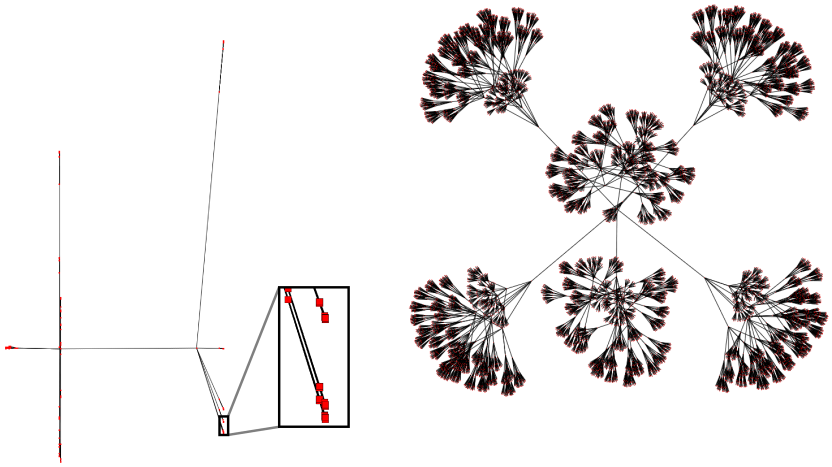


(c) GRIP, 2.43s (d) FM^3, 21.99s



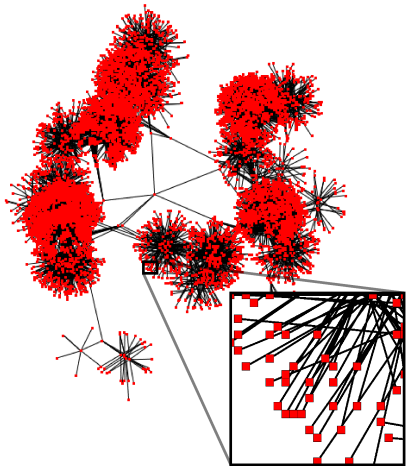
(e) TopoLayout, 3.35s

Figure 3.6: Layouts of the `Crack` dataset, with $10,240$ nodes and $30,380$ edges, using ACE, HDE, GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.

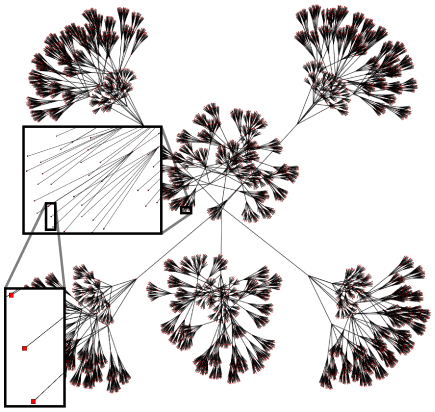


(a) ACE, 0.72s

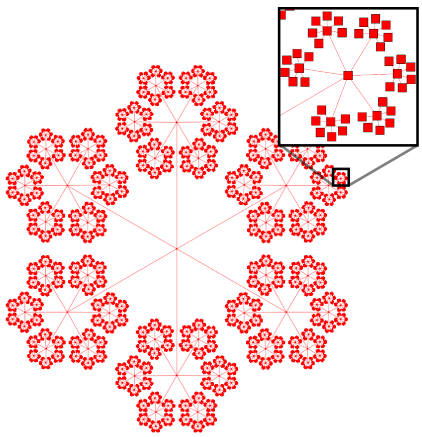
(b) HDE, 0.08s



(c) GRIP, 1.02s

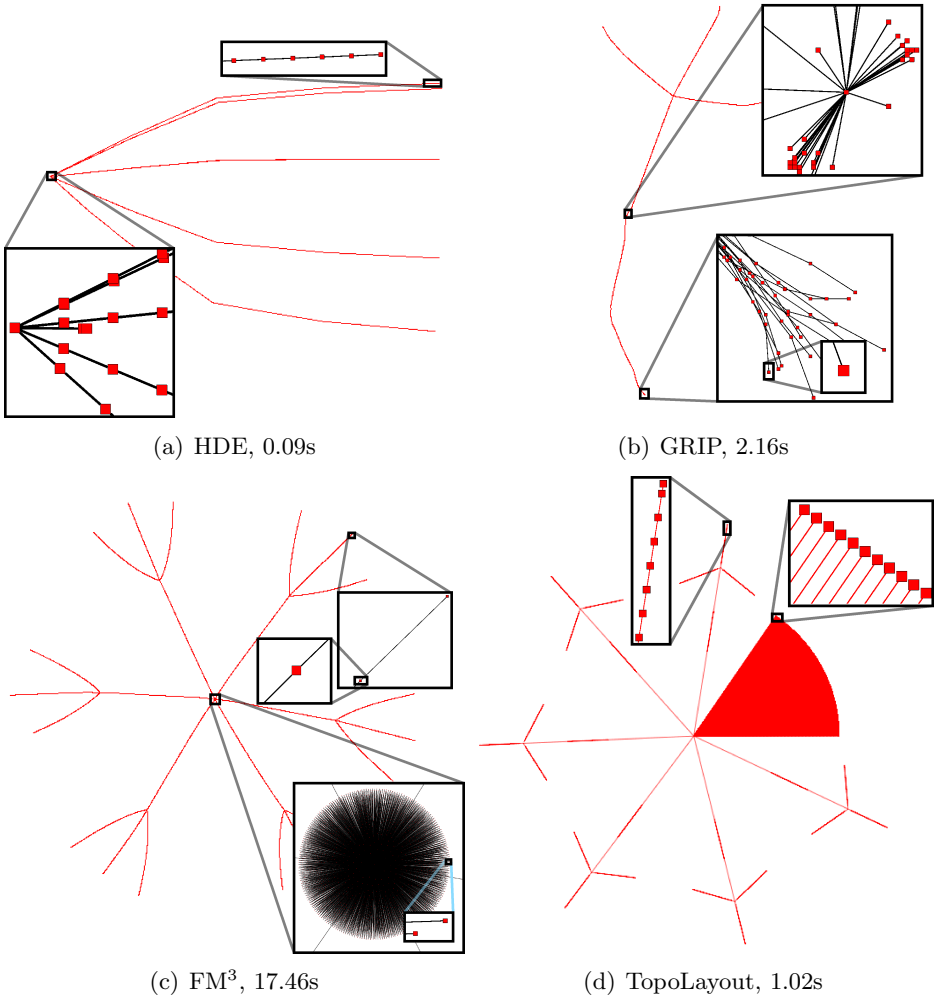


(d) FM^3, 17.09s



(e) TopoLayout, 0.97s

Figure 3.7: Layouts of the `6-ary` dataset, with 9,331 nodes and 9,330 edges, using ACE, HDE, GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.



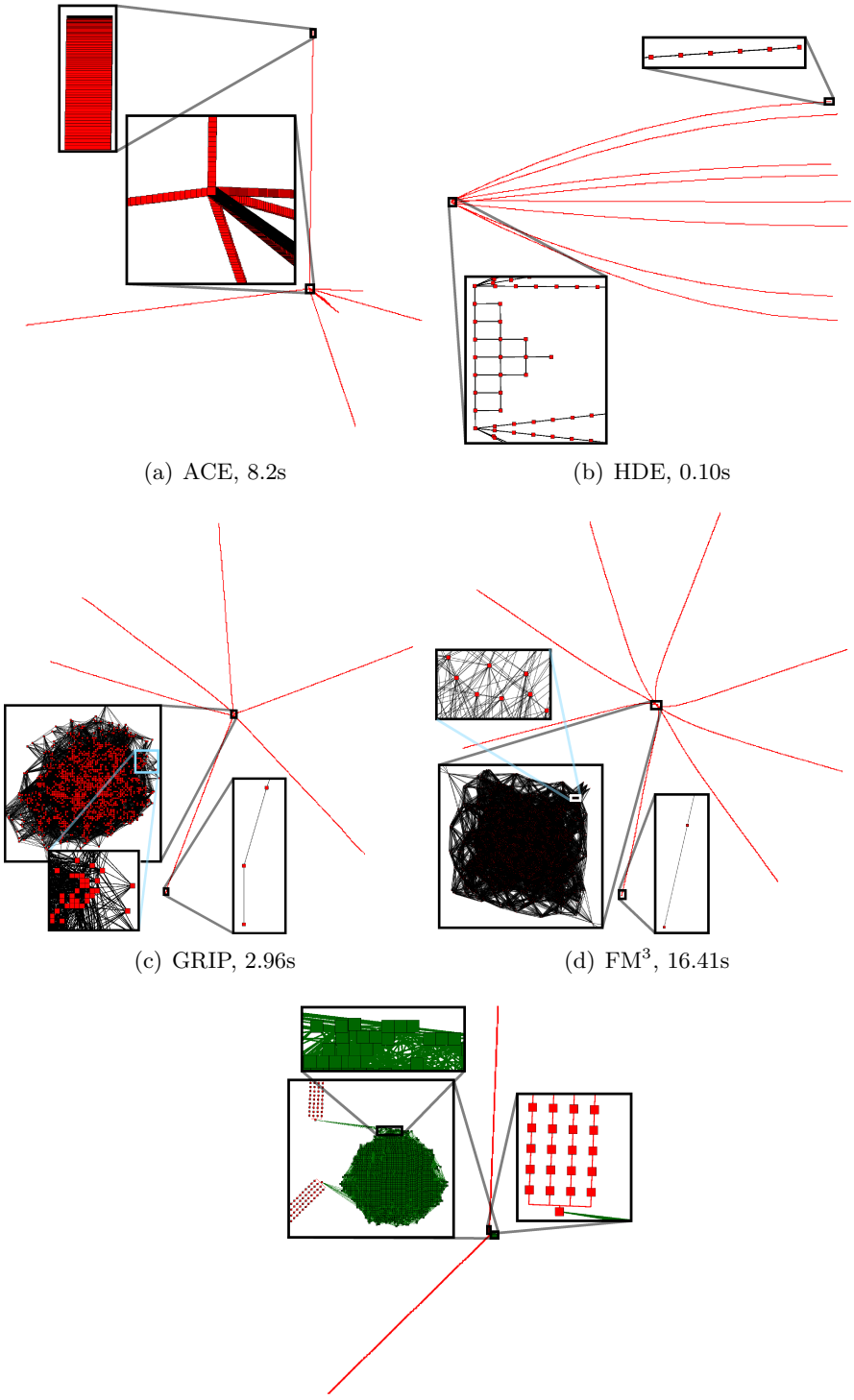
(a) HDE, 0.09s

(b) GRIP, 2.16s

(c) FM^3, 17.46s

(d) TopoLayout, 1.02s

Figure 3.8: Layouts of the `Snowflake` dataset, with 9,701 nodes and 9,700 edges, using HDE, GRIP, FM^3, and TopoLayout. ACE was not able to generate a drawing of this dataset in four hours of execution time. Times in seconds to lay out each graph indicated below each drawing.



(a) ACE, 8.2s

(b) HDE, 0.10s

(c) GRIP, 2.96s

(d) FM^3, 16.41s

(e) TopoLayout, 403.88s

Figure 3.9: Layouts of the `Spider` dataset, with 10,000 nodes and 20,000 edges, using ACE, HDE, GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.

`Flower`

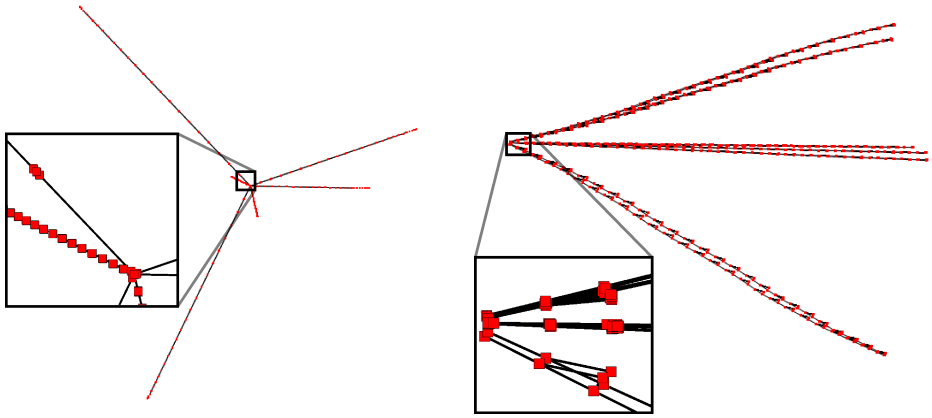
Figure 3.10 shows the results for `Flower`. TopoLayout and FM^3 are able to draw most of the high-level and low-level structures in this dataset while GRIP is only able to draw parts of both. TopoLayout draws each of the K_{30} cliques with circular layout, for a visual indication that the graphs are 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 shows that a feature-based approach has promise, 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.

`bi_walsh`

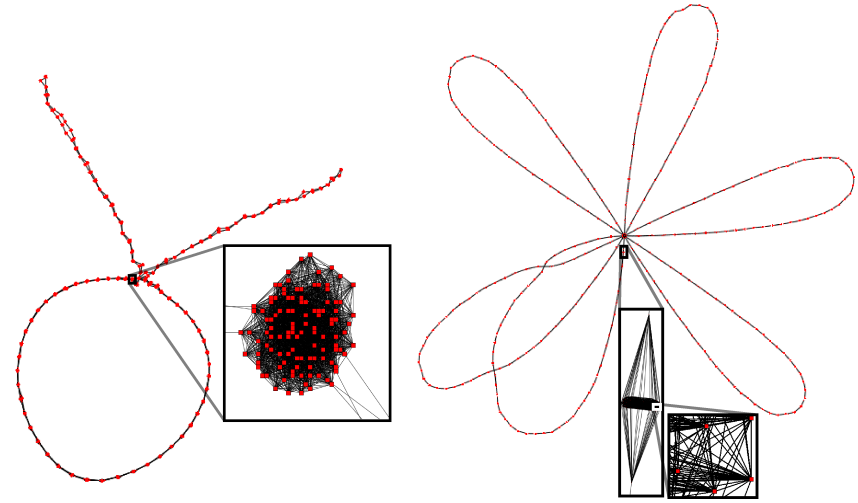
The results on `bi_walsh` are shown in Figure 3.11. TopoLayout and FM^3 draw the high-level biconnected structure of this dataset well. TopoLayout detects the high-level biconnected structure present in this graph and draws it using bubble tree. It uses HDE to draw each of the thirteen mesh-like datasets 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.

3.4.2 Real World Data

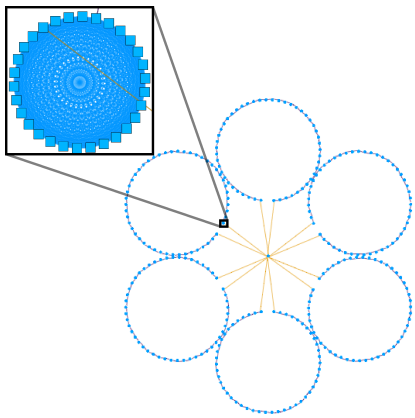
On the real world datasets, the TopoLayout running times were usually of a similar order of magnitude as FM^3. The only exception was `IMDB 1999` where TopoLayout took just over a minute and FM^3 took two seconds. GRIP was faster than all algorithms, but it frequently yielded results of lower visual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



(a) ACE, 0.15s (b) HDE, 0.15s

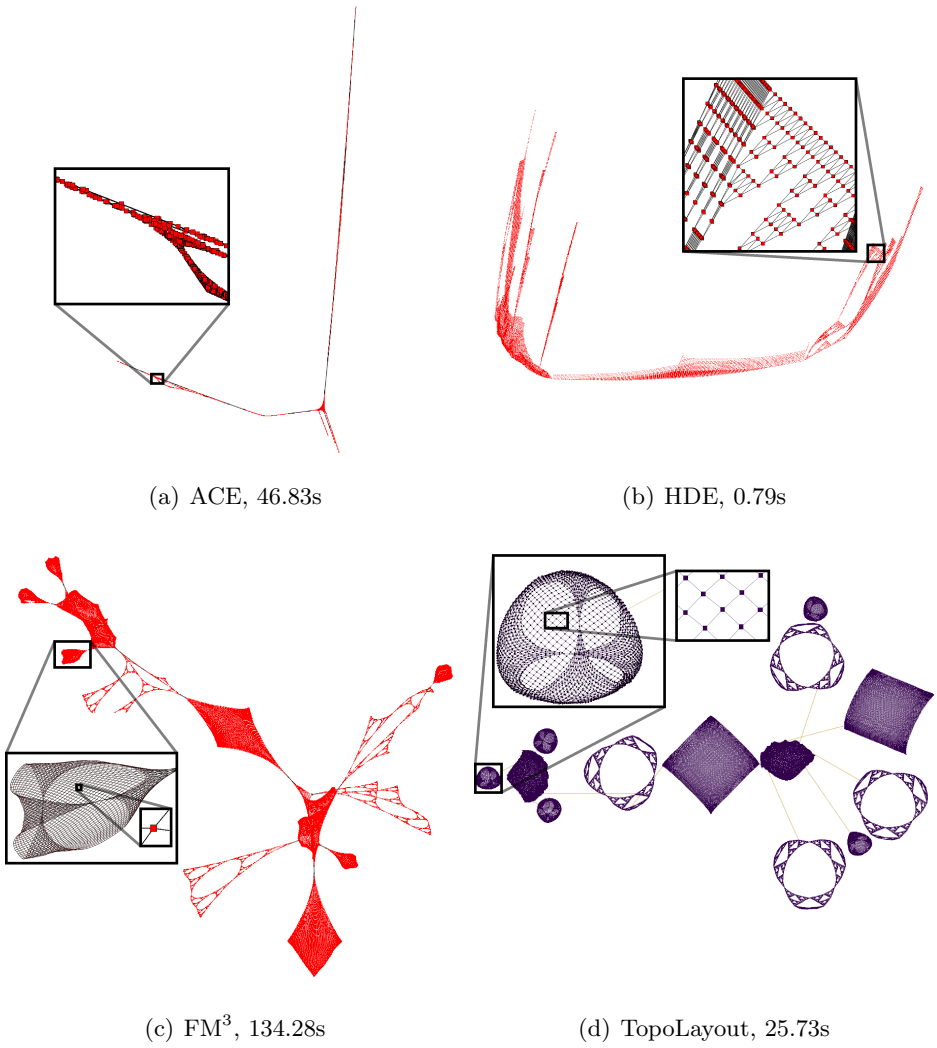


(c) GRIP, 4.40s (d) FM^3, 11.37s



(e) TopoLayout, 70.00s

Figure 3.10: Layouts of the `Flower` dataset, with 9,030 nodes and 131,241 edges, using ACE, HDE, GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.



(a) ACE, 46.83s

(b) HDE, 0.79s

(c) FM^3, 134.28s

(d) TopoLayout, 25.73s

Figure 3.11: Layouts of the `Bi_Walsh` dataset, with 77,251 nodes and 183,945 edges, using ACE, HDE, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing. GRIP did not produce a drawing because of an execution-time error.

`ug_380`

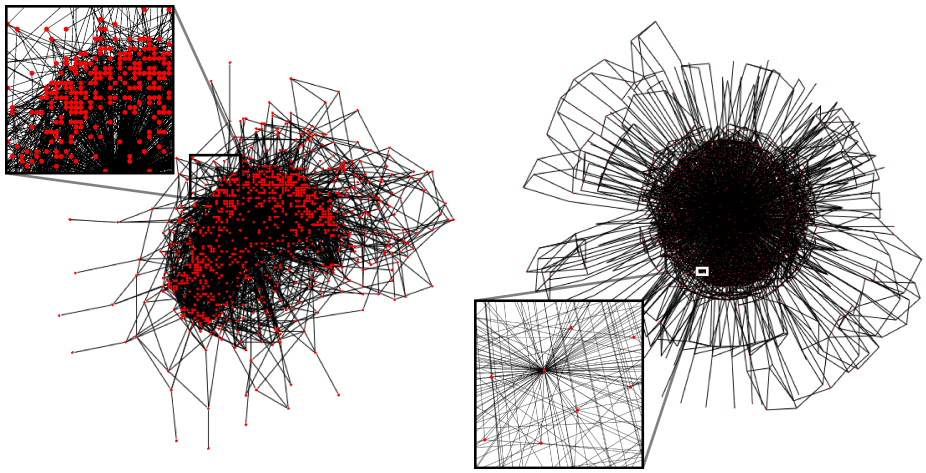
This dataset contains a single node of very high degree with some interesting structure at some graph-theoretic distance from this central node. Figure 3.12 shows the results for this dataset. The high-level structure could be improved in all three drawings. However, TopoLayout is able to depict some interesting 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 connectivity 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 the high degree node in the drawing. The connectivity 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 that 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.

`dg_1087`

This dataset is a tree with a very high degree node at the root. Figure 3.13 shows the results for this dataset. 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, as much of it is hidden by many overlaps. It is very hard to see the low-level structure of this dataset using GRIP.

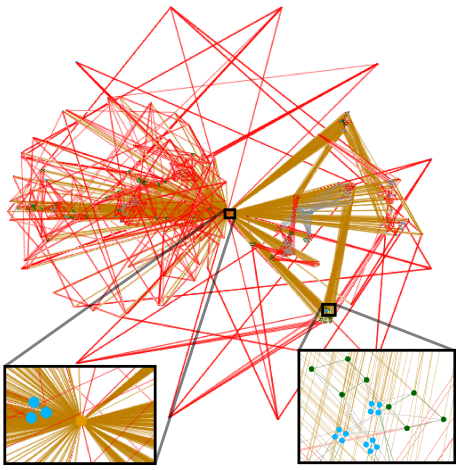
`Add32`

In this TopoLayout drawing, a feature-based approach on real world data shows promise. Figure 3.14 shows the results. 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, the drawing depicts 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



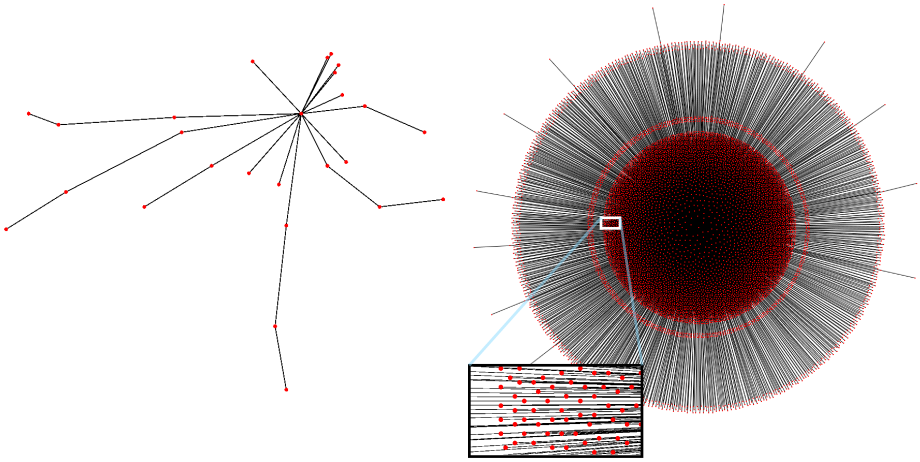
(a) GRIP, 0.22s

(b) FM^3, 2.12s



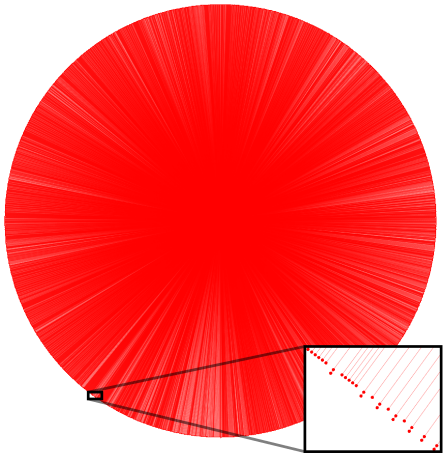
(c) TopoLayout, 10.23s

Figure 3.12: Layouts of the `ug_380` dataset, with $1,104$ nodes and $3,231$ edges, using GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.



(a) GRIP, 3.29s

(b) FM^3, 17.48s



(c) TopoLayout, 0.78s

Figure 3.13: Layouts of the `dg_1087` dataset, with 7,602 nodes and 7,601 edges, using GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.

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.

`UBC`

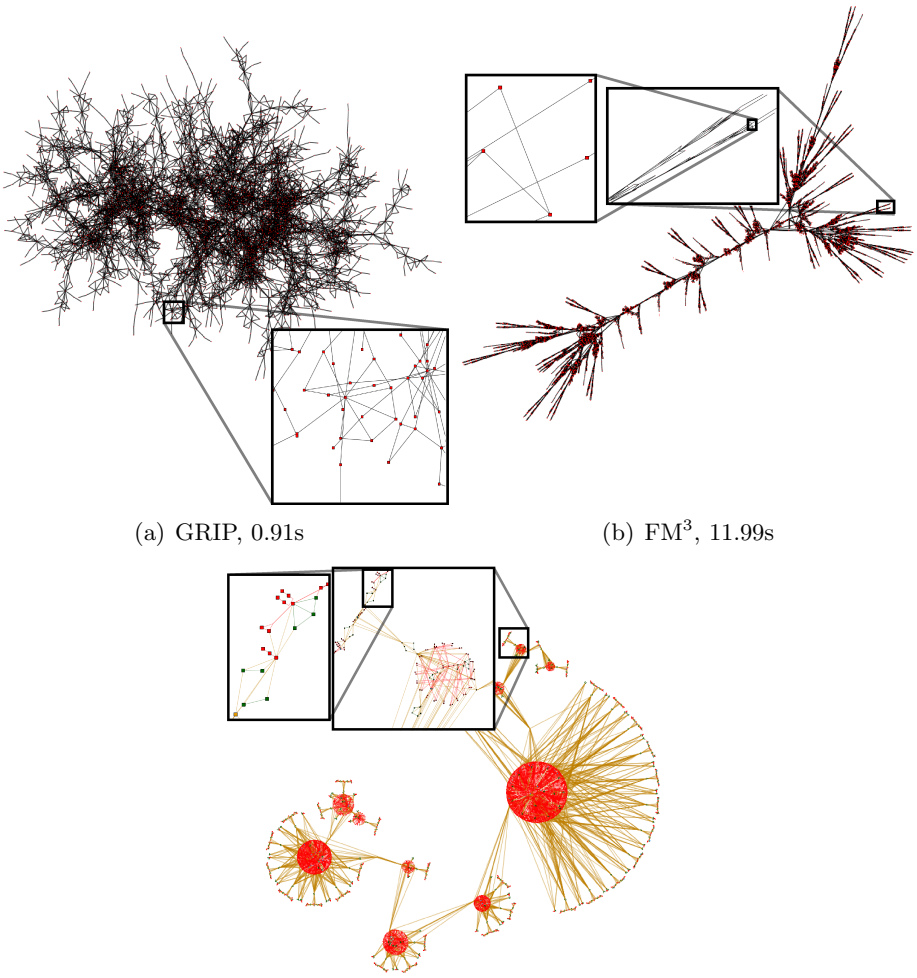
Figure 3.15 shows the results for FM^3 and TopoLayout on this dataset. TopoLayout is able to draw the high-level tree structure of this dataset seen in the red and tan components of the drawing. As the dataset was acquired using a breadth first search of weblinks, this high-level tree backbone of the dataset is expected. 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.

`IMDB 1999`

This IMDB subset is a very hard dataset to draw because of its high connectivity. None of the algorithms produce convincing results in revealing the high-level structure in the dataset as shown in Figure 3.16. TopoLayout is unable to reveal high-level structure, because much of it is obscured by large swathes of edges. The algorithm 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. The circular placement of the nodes highlights these subgraphs as complete. 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.

3.5 Edge Crossing Reduction Evaluation

We provide some support for the claim that the crossing reduction phase reduces the number of edge crossings in the final layout by counting the number of crossings in the entire drawing before and after this phase has executed. The effectiveness of the crossing reduction phase on eleven examples ranges from 0.47% more crossings to to 32% less crossings in the final

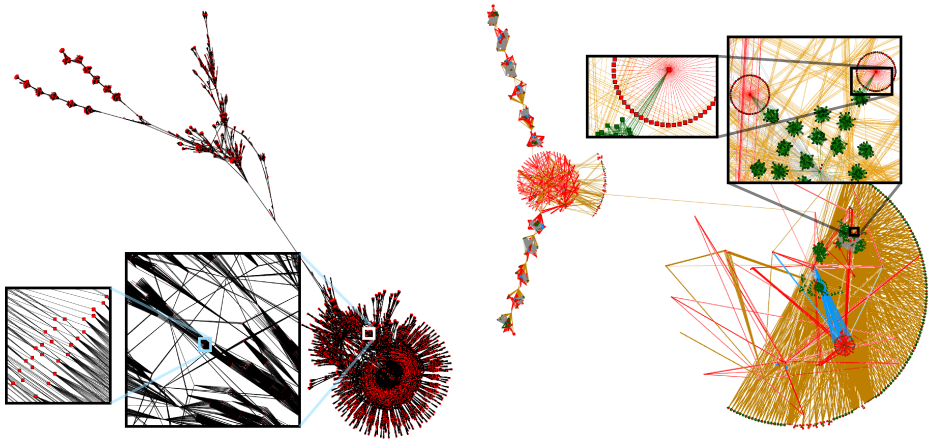


(a) GRIP, 0.91s

(b) FM^3, 11.99s

(c) TopoLayout, 14.02s

Figure 3.14: Layouts of the `Add32` dataset, with 4,960 nodes and 9,462 edges, using GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.



(a) FM^3, 84.56s (b) TopoLayout, 220.79s

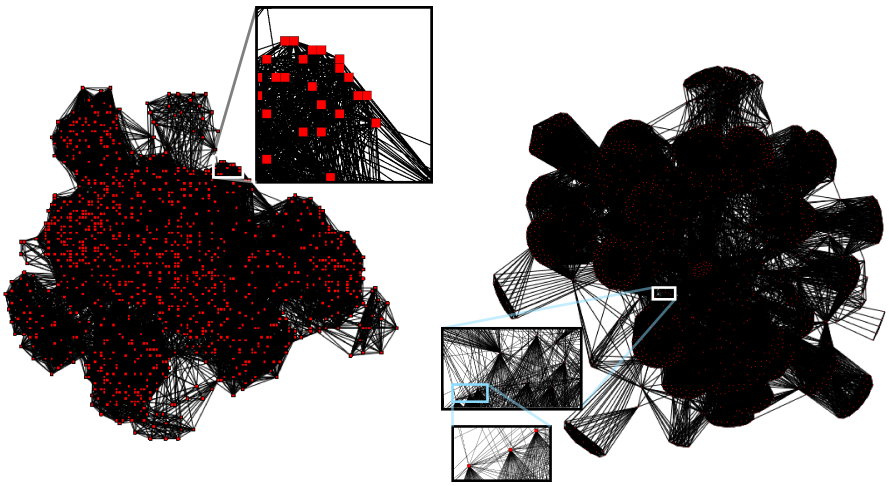
Figure 3.15: Layouts of the `UBC` dataset, with 40,011 nodes and 191,659 edges, using FM^3 and TopoLayout. Times in seconds to lay out each graph indicated below each drawing. GRIP did not produce a drawing due to an execution-time error.

drawing. The results of the tests are shown in Figure 3.17. We categorize our results into four types: no change, minor increase, minor decrease, and significant reduction.

To compute these numbers, exactly one run of TopoLayout was executed. The **before** column lists the number of edge crossings in the final layout when the execution of the crossing reduction phase is disabled. The **after** column lists the number of edge crossings in the final layout when the crossing reduction is applied. The far right column is the percentage increase or decrease in the total number of crossings after the crossing phase was applied with negative percentages indicating a decrease in the number of crossings. The layout of the metagraphs in all metanodes of the hierarchy is the same for both the before and after trials. Therefore, only the crossing reduction phase can influence the number of crossings presented in the table.

`Crack`, `6-ary`, `Snowflake`, and `dg_1087` saw no change in the number of crossings. `Crack` is a HDE component and the remaining three datasets are trees. In these cases, TopoLayout places the entire graph inside a single metanode and draws it with an appropriate algorithm. No torsional forces are executed between metanodes because only a single metanode exists.

Three of the datasets saw very little change in the number of crossings.



(a) GRIP, 0.78s

(b) FM^3, 2.64s

(c) TopoLayout, 75.62s

Figure 3.16: Layouts of the `IMDB 1999` dataset, with 1,181 nodes and 31,527 edges, using GRIP, FM^3, and TopoLayout. Times in seconds to lay out each graph indicated below each drawing.

`Add32` and `Spider` saw a small increase while `Flower` saw a small decrease in the number of crossings. On `Add32` and `Spider`, we conjecture the reason for a small increase in the number of crossings is due to the fact that both datasets have a large number of trees in them. The root node of a tree is usually placed close to the centre of the metagraph layout. Therefore, any edge connected to this root node intersects the metanode for approximately the same amount of distance independent of orientation. As a result, it is more difficult to reduce edge crossings by changing the orientation of the metanode. Therefore, a small increase in the number of crossings can and does occur.

On `Flower`, a small decrease in the number of edge crossings is realized. In this dataset, six chains consisting of the complete graph K_{30} are joined at a single instance of K_{30}. A change in orientation of a complete graph does not resolve any of the crossings in its circular drawing, and the majority of the crossings in this graph exist in the circular drawings of the complete graphs. Therefore, any change in the crossings that can be resolved is subsumed in this statistic. We conjecture this reason is the cause for a small decrease in the number of crossings on `Flower`.

For all remaining drawings, the number of crossings in the final drawing was reduced by the crossing reduction phase. The reduction ranges from 4% to 32%.

The crossing reduction phase only has rotation at its disposal to reduce crossings between metanodes in the graph hierarchy once the layouts of the metagraphs have been fixed. We hope to reduce the number of crossings in the drawing by using this heuristic, but in some cases it can increase the total number of crossings as we have seen from the results. In future work, it may be possible to reduce crossings further by changing the layout slightly. Also, it would be interesting to determine if some edge crossings at higher levels of the graph hierarchy are more important perceptually than others. For example, it seems reasonable that a single crossing between two of the loops in `Flower`, Figure 3.10 in the results section, is much worse than a single crossing between two edges of one of the complete graphs. In future work, perceptual experiments with users would be required to determine metrics more in line with quantifying high-, mid-, and low-level structure in terms of the number of edge crossings.

Dataset	Before	After	+/-%
Crack	11,730	11,730	0.0
6-ary	0	0	0.0
Snowflake	0	0	0.0
Spider	3,144,468	3,145,008	+0.017
Flower	8,293,970	8,265,540	-0.34
bi_walsh	44,613	42,779	-4.1
ug_380	135,237	91,144	-32.0
dg_1087	0	0	0.0
Add32	342,856	344,483	+0.47
UBC	17,521,900	16,277,740	-7.1
IMDB 1999	21,351,255	18,504,048	-13.0

Figure 3.17: Table of edge crossing numbers before and after the crossing reduction phase. The left column is the dataset name. Before and after are the number of edge crossings with and without crossing reduction on a single layout of the graph. The far right column is the percentage increase or decrease in the number of crossings after the crossing reduction phase has occurred.

3.6 Robustness

We now discuss how the order of the nodes in the input affects the final drawing. In the decomposition phase of TopoLayout, both tree and biconnected component decompositions are unique and therefore are unaffected by node order. The clustering algorithm used can be affected if two edges have the same number of cycles pass through them and HDE detection can be affected as detection begins from a random node in the graph. In the drawing phase, the only randomly seeded layout algorithm is area-aware GEM. The remaining algorithms in the drawing phase can also be affected by node order. In this section, we supply a pair of examples to show that on two random orders of nodes and edges the effect is small.

Figure 3.18 supports the idea that the levels of structure depicted and symmetries in the drawings are not greatly affected by the supplied input order of the nodes. Both datasets used HDE detection and clustering as well as other algorithms in TopoLayout. The running times to produce these drawings were roughly the same as reported in the results section of this chapter. The input presented to TopoLayout was the same dataset, but with the order of the nodes and edges scrambled randomly. Although

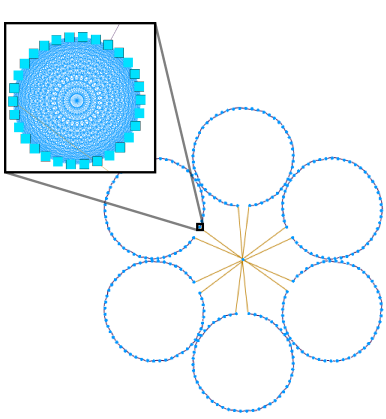
the coordinates of a given node in the drawing may be very different, TopoLayout produces drawings where the visual quality of the two drawings is similar.

3.7 Discussion

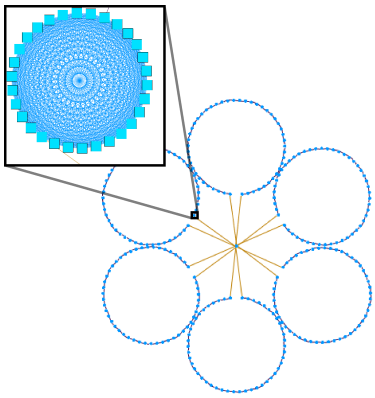
TopoLayout realizes a feature-based approach to graph visualization by providing a decomposition phase that divides the input graph recursively by connectivity features. An empirical evaluation of TopoLayout on eleven datasets shows the circumstances where it is better able to present connectivity features in the graph at various levels of structure. In terms of running time, the approach is competitive with multilevel approaches on these eleven datasets. However, in nearly every dataset, TopoLayout is unable to depict mid-level structure in the final drawings. In future chapters of this thesis, we will partly address this issue.

We have provided some evidence through the eleven datasets used in the empirical evaluation of Hachul and Jünger that the connectivity features detected by TopoLayout appear in graphs other authors have found interesting. However, a fundamental limitation of the TopoLayout decomposition phase is that if the connectivity features being sought by the algorithm are not present, the running time and visual quality of the algorithm deteriorates. This limitation is a limitation of any feature-based approach to graph visualization: if the feature being sought is not found, the algorithm must resort to default graph drawing behaviour.

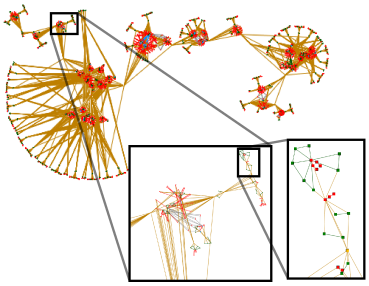
The TopoLayout framework is used as a basis for the remaining chapters of this thesis. These systems improve upon this primary limitation of TopoLayout by developing an algorithm tuned to a specific type of graph in chapter 4 and by allowing for steerable graph layout in chapters 5, 6, and 7.



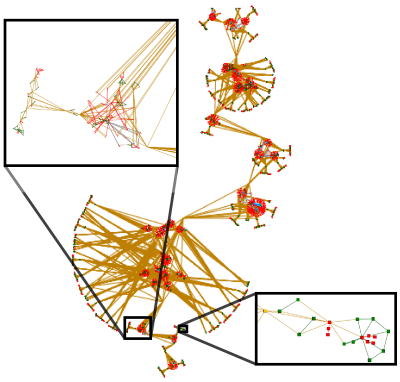
(a) Flower Run 1



(b) Flower Run 2



(c) Add 32 Run 1



(d) Add 32 Run 2

Figure 3.18: We show two different runs of TopoLayout on `Flower` and `Add32`, each with a randomized input order for nodes in the graph. Although the random reordering of the nodes can significantly affect the absolute spacial coordinates of a given node, it has very little effect on the levels of structure depicted in the drawing.

Chapter 4

Smashing Peacocks Further: Drawing Quasi-Trees from Biconnected Components

The TopoLayout multilevel algorithm is a general approach for drawing undirected graphs. However, if more information about the graph is available beforehand, we can create a more specific pipeline for the type of data presented to the algorithm. In Smashing Peacocks Further, or SPF, we modify the TopoLayout decomposition and drawing pipeline to better handle **quasi-trees**, defined as graphs with many biconnected components. This work was specifically motivated by the problems of visualizing protein homology maps [3] in bioinformatics and Internet tomography maps [16] in computer networks. In these two domains, similar algorithms have been developed that use trees as a basis for graph layout. These approaches use a specified spanning tree as scaffolding for a force-directed layout. The purpose behind using SPF is to decompose into biconnected components in order to depict the high-level tree structure using a tree drawing algorithm and the force-directed approach of previous work on the individual biconnected components.

The approach uses a single biconnected component decomposition to divide a large graph into biconnected components that are of a more manageable size. These components are drawn with LGL [3], using the part of the predefined spanning tree that the component contains. Once the extent of each biconnected component is determined, the biconnected component tree is drawn using the area-aware RINGS algorithm.

The decomposition phase of SPF detects biconnected components of the graph as described in Section 1.1.2. This phase is computed offline and biconnected component decomposition is applied only once to the large graph, creating a hierarchy of two levels. The drawing of the graph is also computed offline. It computes a layout of the final graph once, using a single postorder traversal of the graph hierarchy.

The contribution of SPF is an algorithm that tunes the TopoLayout pipeline for drawing quasi-trees. Additionally, the approach extends the previous RINGS algorithm in order to make it area-aware. We implemented SPF and compared it empirically to previous systems. Additionally, we extended some traditional graph drawing metrics to a multilevel graph drawing context in order to evaluate the high-level tree structure in the drawings. SPF was published at *InfoVis 2006* and appeared in a special issue of *IEEE Transactions on Visualization and Computer Graphics* [5]. A video[8] compares drawings produced by SPF to drawings produced by three competitive algorithms on two large datasets. An implementation of the algorithm is available online[9].

This chapter is structured as follows. Section 4.1 presents an overview of the SPF algorithm and its components. The decomposition phase, biconnected component drawing phase, and high-level tree drawing phase are discussed in Sections 4.1.1, 4.1.2, and 4.1.3 respectively. Empirical testing of SPF on data from two application domains that have quasi-tree structure is presented in Section 4.2. A discussion of algorithm robustness is presented in Section 4.3. Finally, a discussion of the contributions of SPF is presented in Section 4.4.

4.1 Algorithm Overview

The inputs to SPF are a graph and an unrooted, minimum spanning tree. The graph can be weighted or unweighted. For weighted graphs, Kruskal's algorithm [68] is used to compute the spanning tree, and for unweighted graphs, breadth-first search [12] is used. For unweighted graphs, all spanning trees are of equal cost. Spanning trees computed from a breadth-first search are a logical choice as they tend to be representative of the shortest path distance between a pair of nodes in the graph [68], keeping areas of the graph close in terms of graph theoretic distance also close in terms of Euclidean distance in the drawing. The algorithm roots the input spanning tree at its tree centre, namely the node at the midpoint of the tree's diameter.

The drawing algorithm runs in three phases:

1. Decompose the graph into biconnected components.

2. Draw each component using an improved version of LGL.

[8] `www.cs.ubc.ca/labs/imager/video/2006/spf`

[9] `www.cs.ubc.ca/labs/imager/tr/2006/Archambault_SPF_InfoVis2006/`

3. Draw the biconnected component tree using an area-aware version of the RINGS [72] algorithm.

The first phase decomposes the input graph into biconnected components. In the second phase, the algorithm draws each biconnected component with an improved version of LGL and sets the sizes of the nodes in the biconnected component tree to the extents of the biconnected components. The third phase draws the biconnected component tree using an area-aware version of the RINGS [72].

4.1.1 Decomposition into Biconnected Components

In SPF, the graph is decomposed into biconnected components using a standard algorithm from the literature [12]. The biconnected component tree is constructed as shown in Figure 4.1. Bridge edges, such as the edge between components C_3 and C_4, are edges in this tree. Bridge nodes appear as nodes in the tree. If a bridge node shares two or more edges with a component, the bridge node is duplicated and placed into those adjacent components as shown with C_1 and C_2. This duplication keeps nodes directly adjacent to the bridge node together during layout of the biconnected component, but the duplicated bridge nodes do not appear in the output drawing.

4.1.2 Biconnected Component Drawing with Optimized LGL

Once the graph is divided into biconnected components, the algorithm uses the improved version of LGL, **optimized LGL**, to draw each biconnected component. The roots are computed from the spanning tree as shown in Figures 4.1(b) and 4.1(c). If the root of the input spanning tree is present in a biconnected component, the root is chosen as the root of the biconnected component. Otherwise, the node from which the spanning tree entered the component is used. The edges of the input spanning tree present in the biconnected component are used as the spanning tree for the biconnected component.

The first optimization to LGL improves running time in some cases and leaves visual quality unchanged. Recall that LGL uses a grid to cull repulsive forces of sufficiently distant nodes. To compute the repulsive forces, LGL marches through each cell of the grid. Computing repulsive forces in this manner can be costly in the early stages of the algorithm when many cells are empty. Instead, the algorithm keeps a list of nodes already placed by the spanning tree, determines the position of the node in the grid, and computes

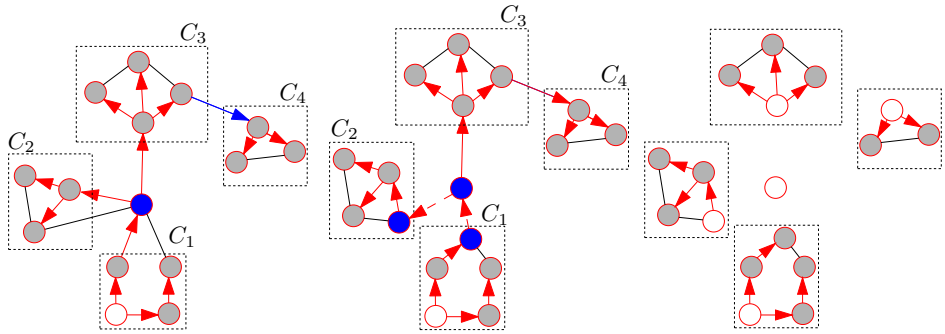


Figure 4.1: Decomposition of the graph into biconnected components preserves the input spanning tree. Spanning tree edges are shown with directed red edges, and the roots are shown in white. **(a)** Identify the bridge nodes and edges, shown in blue, in the decomposition phase. **(b)** Bridge nodes are duplicated and placed into adjacent components. **(c)** Break spanning tree up into individual biconnected components, and find the root for each new component.

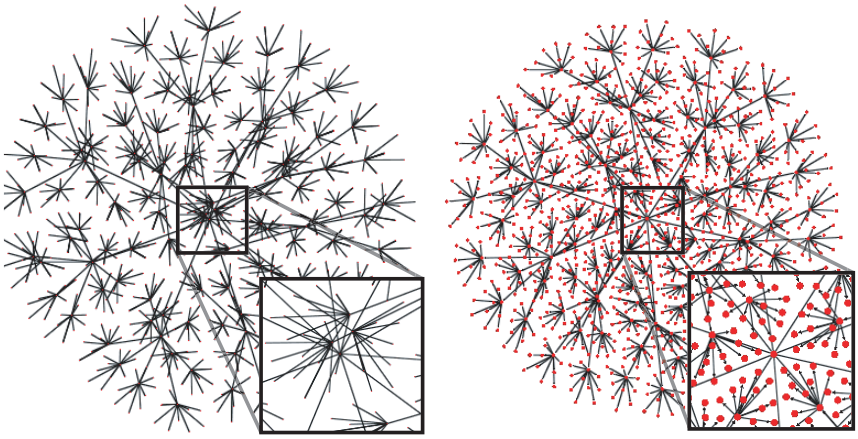
the repulsive forces directly. The algorithm marks the cell to ensure that the repulsive forces between nodes in a cell are computed only once. The drawing remains unchanged, because exactly the same set of repulsive forces are computed as when marching through the grid.

The second optimization improves the visual quality of the final drawing of the graph, as shown in Figure 4.2. This optimization consists of two parts that influence initial placement of the nodes in the layout. In LGL, nodes are placed into the layout using the input spanning tree as a skeleton according to $\vec{S}$:

$$\vec{S} = c(\hat{M} + \hat{P}) + x_{\text{parent}} \tag{4.1}$$

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

The first part improves placement by reducing the scaling factor c to the sum of the size of the parent node, the size of child node, and the average size of the nodes in the graph. Reducing c yields more compact drawings. In Figure 4.2(a) the nodes are spread far apart and can hardly be seen while in



(a) LGL (b) optimized LGL

Figure 4.2: Comparison of the final layout of a ten-ary tree of depth three between **(a)** LGL and **(b)** optimized LGL. Repulsive forces are diminished in optimized LGL to roughly the size of a node.

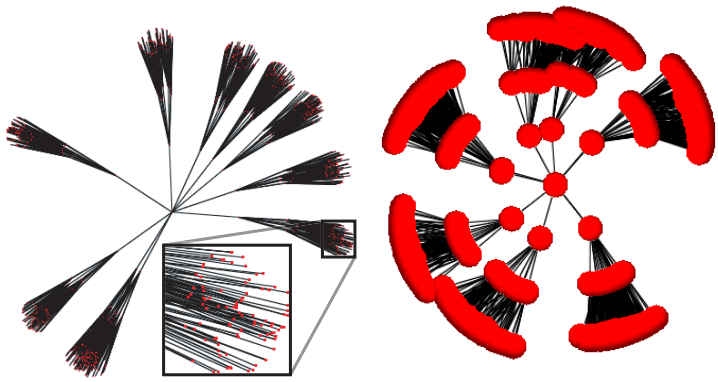
Figure 4.2(b) they are easily visible along with the connectivity. The second part places nodes on small directed fans in the direction of the vector $\vec{S}$ as shown in Figure 4.3(b) instead of circles as shown in Figure 4.3(a). Placing the nodes on fans inhibits them from being directed back inwards, preserving low-level structure. This problem is shown qualitatively in Figure 4.2(a). In contrast, the problem is less prevalent in optimized LGL as shown in Figure 4.2(b).

4.1.3 Biconnected Component Tree Drawing with Area-Aware RINGS

Once each biconnected component has been laid out, the algorithm has computed the screen-space extents required by each of these metanodes. In order to draw the biconnected component tree, the algorithm needs a tree drawing algorithm that is area-aware.

This section begins by motivating the need for area-aware RINGS. Next, it describes the original RINGS algorithm as presented by Teoh and Ma [72]. Then, it describes the area-aware adaptation of RINGS and discusses how to use the algorithm to draw the biconnected component trees.

For large quasi-trees, the degree of nodes inside the biconnected component tree can be very high. Figure 4.4 shows the few previously existing

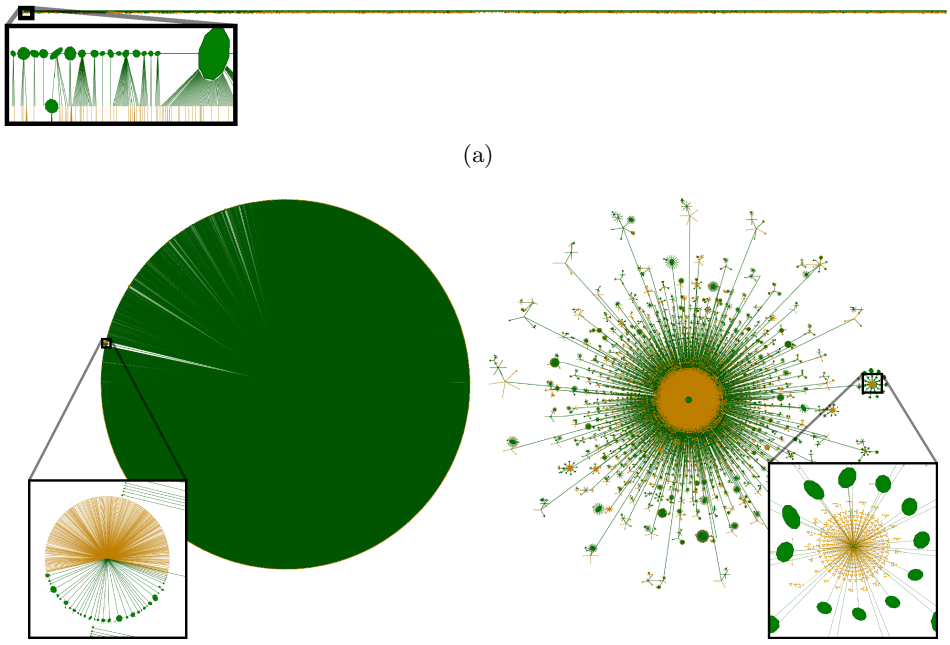


(a) LGL (b) optimized LGL

Figure 4.3: Comparison of the initial placement of nodes between **(a)** LGL and **(b)** optimized LGL, on a ten-ary graph of depth tree. Fan placement also places nodes closer to each other at the beginning.

tree layout approaches suitable for this data. The area-aware version of the Walker [15] layout, shown in Figure 4.4(a), lays out the children of the high-degree node on a horizontal line, and the details of their structure are too small to be seen without zooming. Similarly, Figure 4.4(b) shows that bubble tree [37] lays out the high-degree node in the centre of a circle so large that the children are too small to be seen individually on the circumference. In contrast, we argue qualitatively that the RINGS algorithm [72] provides a much more compact drawing and a better result at the price of introducing node-edge overlaps and edge crossings. By adapting it to be area-aware, the result shown in Figure 4.4(c) is achieved. In this approach, the node-edge overlaps of area-aware RINGS results in metanode-edge overlaps between the metanodes of the biconnected component tree and the edges of the graph in the final drawing.

In RINGS [72], each child of a root is placed in a circle enclosing its entire subtree as shown in Figure 4.5(a). The blue node at the centre of the drawing is the root while the grey circles are the children of the root and their subtrees. The root is placed at the centre of the drawing, and the algorithm sorts its children by their number of children; that is, by their number of grandchildren with respect to the root. The subtrees are placed onto concentric rings inward towards the root in order from the child with the most to least children. Each of the subtrees on a ring is given an equal-sized enclosing circle of radius r. The radii R_1 and R_2 are as shown in



(a)

(b) (c)

Figure 4.4: Layouts of the `Net05` dataset biconnected component tree using: **(a)** area-aware Walker's algorithm [15], **(b)** bubble tree [37], and **(c)** area-aware RINGS. With area-aware RINGS, a significantly more compact drawing is obtained at the price of introducing edge-node overlaps.

Figure 4.5(a). Let $N_{\text{GrandPlaced}}$ be the number of grandchildren placed in the current ring. Let $N_{\text{GrandTotal}}$ be the total number of grandchildren to place with respect to the root. A new ring is started on the inner, yellow circle when:

$$\frac{N_{\text{GrandPlaced}}}{N_{\text{GrandTotal}}} > \frac{{R_2}^2}{{R_1}^2} = \frac{(1-\sin(\frac{\pi}{n}))^2}{(1+\sin(\frac{\pi}{n}))^2} \tag{4.2}$$

Given n children and their subtrees, the right hand side of Equation (4.2) is the ratio of the circle areas with radii R_1 and R_2 respectively. This ratio does not depend on r and only requires that $R_1 = 2r + R_2$. The process is repeated for the remaining subtrees.

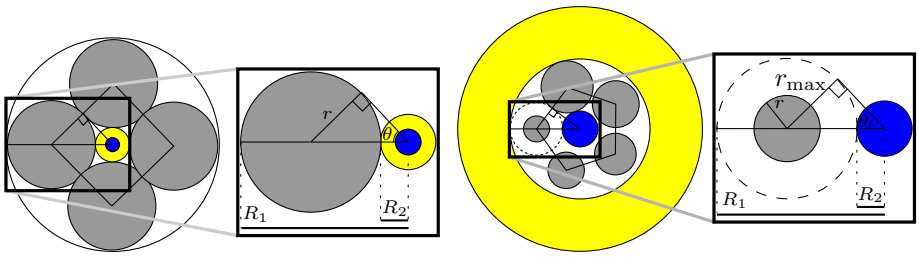
The left hand side of Equation (4.2) demonstrates that the size of the nodes in the tree is not considered, only the number of children. Therefore, RINGS assumes a uniform node size. Moreover, the node size should be much smaller than r to ensure that the entire subtree is fully contained by the enclosing circle, since only grandchildren, and not the entire subtree, are considered in this ratio. In this work, the subtree could contain a large number of nodes of substantial size. Therefore, the algorithm cannot choose such an r.

In the area-aware variant of RINGS, instead of counting the number of grandchildren, the algorithm determines the area needed to lay out each subtree by drawing the tree bottom-up. At a leaf node, the algorithm uses the bounding circle of the node. At an interior node, the algorithm uses the bounding circle of its subtree. The subtrees are placed into concentric rings outward from the root in order from the subtree that requires the least area, to the subtree that requires the most area, as shown in Figure 4.5(b). The algorithm keeps track of the largest enclosing circle radius in $r_{\max}$. The radii R_1 and R_2 are as shown in Figure 4.5(b) with $R_1 = 2r_{\max} + R_2$. A new ring is started on the outer, yellow circle when:

$$\frac{A_{R_2}}{A_{R_1}} < \frac{{R_2}^2}{{R_1}^2} = \frac{(1-\sin(\frac{\pi}{n}))^2}{(1+\sin(\frac{\pi}{n}))^2} \tag{4.3}$$

where A_{R_1} and A_{R_2} are the areas of the circles with radii R_1 and R_2. Since the algorithm has drawn the tree from the bottom up, the algorithm has computed the actual areas of these circles and can compute the ratio directly. Where Equation (4.2) compares ratios of grandchildren, Equation (4.3) compares ratios of the areas needed to draw the nodes and subtrees.

Another optimization of area-aware RINGS comes into effect when the number of subtrees to be placed on a ring is two or fewer as shown in



(a) RINGS (b) Area-Aware RINGS

Figure 4.5: Comparison of subtree placement between **(a)** the RINGS algorithm and **(b)** Area-Aware RINGS. The blue node is the root of the current subtree. The grey circles are subtrees that have filled the white ring. The yellow circle indicates where the next ring is started.

Figure 4.6. The root node of a subtree is usually placed at the centre of its bounding circle. Drawing the subtree in this way leads to wasted space as shown in Figure 4.6(a) and 4.6(c). In the case of a root node with a single child subtree, the algorithm places the root and its child subtree tangent to each other as shown in Figure 4.6(b). In the two-child subtree case, the algorithm places the root and its two subtrees tangent to each other in a triangle as shown in Figure 4.6(d). When the triangle connecting the three centres is acute, the smallest bounding circle is the outer circle tangent to all three enclosing circles. This circle is known as the outer Soddy circle [17, 48]. When the triangle is obtuse, the bounding circle is the circle enclosing two largest circles as shown in Figure 4.6(e).

Additionally, we argue qualitatively from Figure 4.6 that area-aware RINGS improves on how chains are handled by increasing information density. A chain is defined as a linear sequence of nodes each having exactly one child. Without this improvement, each node of the chain is the case shown in Figure 4.6(b), resulting in the long lines of nodes as seen Figure 4.7(a). In this optimization, each node of the chain is treated as though it was a direct child of the node that began the chain. The chain spirals around the node that began it, as shown in Figure 4.7(b). Unlike other cases in area-aware RINGS, the nodes of a chain are not sorted by size.

The algorithm uses a subtle three-dimensional depth effect to create perceptual layering when drawing edges. Edges are placed below the nodes, with edges to the outer rings placed more deeply than edges to the inner ones. When browsing the layout in Tulip [9], edges can be raised if a path between two nodes needs to be visible. Edge colour for edges between biconnected

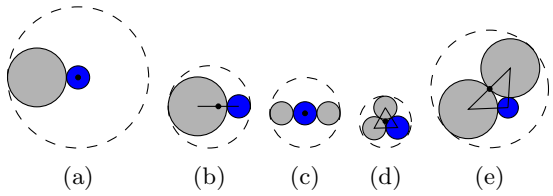


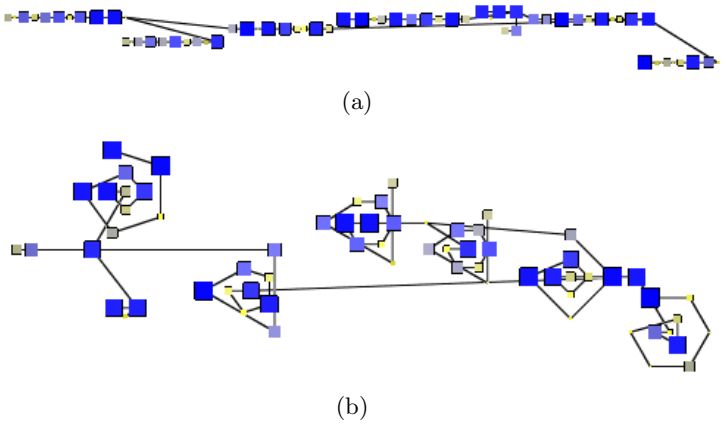
Figure 4.6: Comparison of special cases for small subtree layout. **(a)** One subtree in RINGS. **(b)** One subtree in area-aware RINGS. **(c)** Two subtrees in RINGS. **(d)** Two subtrees in area-aware RINGS. When the triangle connecting the three centres of the root and subtrees is obtuse, the centre of the bounding circle is placed at the centre of the sum of the two largest diameters as shown in **(e)**.

components is lightened.

The algorithm also chooses an ordering for the metanodes in a ring to reduce edge occlusion, since all metanodes in the same ring have an enclosing circle of at most $r_{\max}$. The chosen ordering of the metanodes in a ring is based on the positions of the nodes inside the metanode at the root. The vector between every node inside the root that attaches the metanode on the ring and the centre of the drawing at the root is computed. The average vector is taken as the vector of ideal placement. Metanodes in a ring are sorted based on the direction of their ideal placement vector. The metanode with the most edges in the original graph to the root is placed in its ideal location on the ring. All other biconnected components are placed in their sorted order. This ordering tries to minimize the amount of area an edge passing between two metanodes intersects a metanode, obscuring part of its structure.

4.2 Empirical Evaluation

SPF was implemented using the Tulip [9] framework, and this section now compares it to three other algorithms in terms of performance, qualitative visual results, and quantitative statistics on two large datasets. `Protein`, the graph shown in Figures 4.8 through 4.11 with 30,727 nodes and 1,206,654 edges, is the unweighted version of the protein homology graph presented in the LGL paper [3]. `Net05`, the graph shown in Figures 4.12 through 4.15 with 190,384 nodes and 228,354 edges, is an Internet tomography dataset similar to those presented in Cheswick *et al.* [16], but generated in 2005



(a)

(b)

Figure 4.7: Demonstration of the chain optimization, with nodes set to random sizes between one and ten units. **(a)** Node drawn in linear chains. **(b)** More compact drawing of chain nodes in a spiral.

by Cheswick's Internet Mapping Project[10]. All benchmarks were run on a 3.0GHz Pentium IV with 3.0GB of memory running SuSE Linux with a 2.6.5-7.252 kernel.

SPF is compared to its most competitive algorithms. FM^3 [40] is a state of the art multilevel graph drawing algorithm. Other previous algorithms, including ACE [49], GRIP [34], and HDE [50], were shown to be less competitive than FM^3 in previous work [41]. LGL is an algorithm developed in the bioinformatics domain for visualizing quasi-trees and optimized LGL is a modified form of LGL developed in this work. TopoLayout [7] was not competitive. After twelve hours, the algorithm was still trying to decompose `Protein` because it used a clustering algorithm whose performance deteriorates as the number of edges becomes large. The TopoLayout drawing of `Net05` was not compact, because it drew most of the biconnected component tree using bubble tree. Its drawing is similar to Figure 4.4(b).

4.2.1 Performance

The FM3 algorithm was the fastest algorithm on both datasets. On `Protein`, it was an order of magnitude faster than all the drawing algorithms, and it was three times faster than SPF for `Net05`. LGL and optimized LGL were the slowest algorithms on both datasets: two times slower than SPF

[10] `www.cheswick.com/ches/map`

on `Protein`, and an order of magnitude slower than the other algorithms for `Net05`. SPF was two times faster than LGL and optimized LGL on `Protein` and an order of magnitude faster than them on `Net05`.

4.2.2 Qualitative Drawing Comparison

FM^3 has difficulty depicting both the high-level and low-level structure in both datasets, as shown in Figures 4.8 and 4.12. The high-level, tree-like structure is unclear in `Protein`. On `Net05`, the high-level tree structure is somewhat visible, but details of it are difficult to see, because it draws the branches along thin lines. It is nearly impossible to see low-level structure in either dataset.

With the slower LGL and optimized LGL, the drawings are improved. The high-level, tree-like structure is apparent in `Protein` throughout the dataset as shown in Figures 4.9 and 4.10. High-level branches are visible without zooming in, as well as more of the tree structure in the insets. However, the drawing of `Net05`, shown in Figures 4.13 and 4.14, only displays the high-level tree structure well at the periphery. Most of the drawing is a featureless core where the tree-like structure is hidden. In terms of low-level structure, protein families are drawn in clusters and fusion proteins between families in `Protein`. In `Net05`, the subnetwork structure is only clear when the nodes lie on the fringes of the drawing as shown in the insets of Figures 4.13 and 4.14.

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

The principal advantage of SPF over previous work is the improved visualization of low-level structure in the graph, because it is not as occluded by the higher-level components. In `Protein`, protein families are clustered together and the fusion proteins are drawn between them as shown in the insets of Figure 4.11. The core of `Protein` is thinned, revealing more internal structure than with previous algorithms. Protein families and fusion proteins are clearly seen in the rings of the drawing. The core of `Net05` is far smaller than with LGL; however, it still contains about 37,000 nodes and suffers from a great deal of node and edge occlusion. Nevertheless, many local network features are resolvable in the context of the entire Internet. Subnetwork structure is illustrated around servers at the University of British Columbia and the City of Baltimore as shown in the insets of

Figure 4.15.

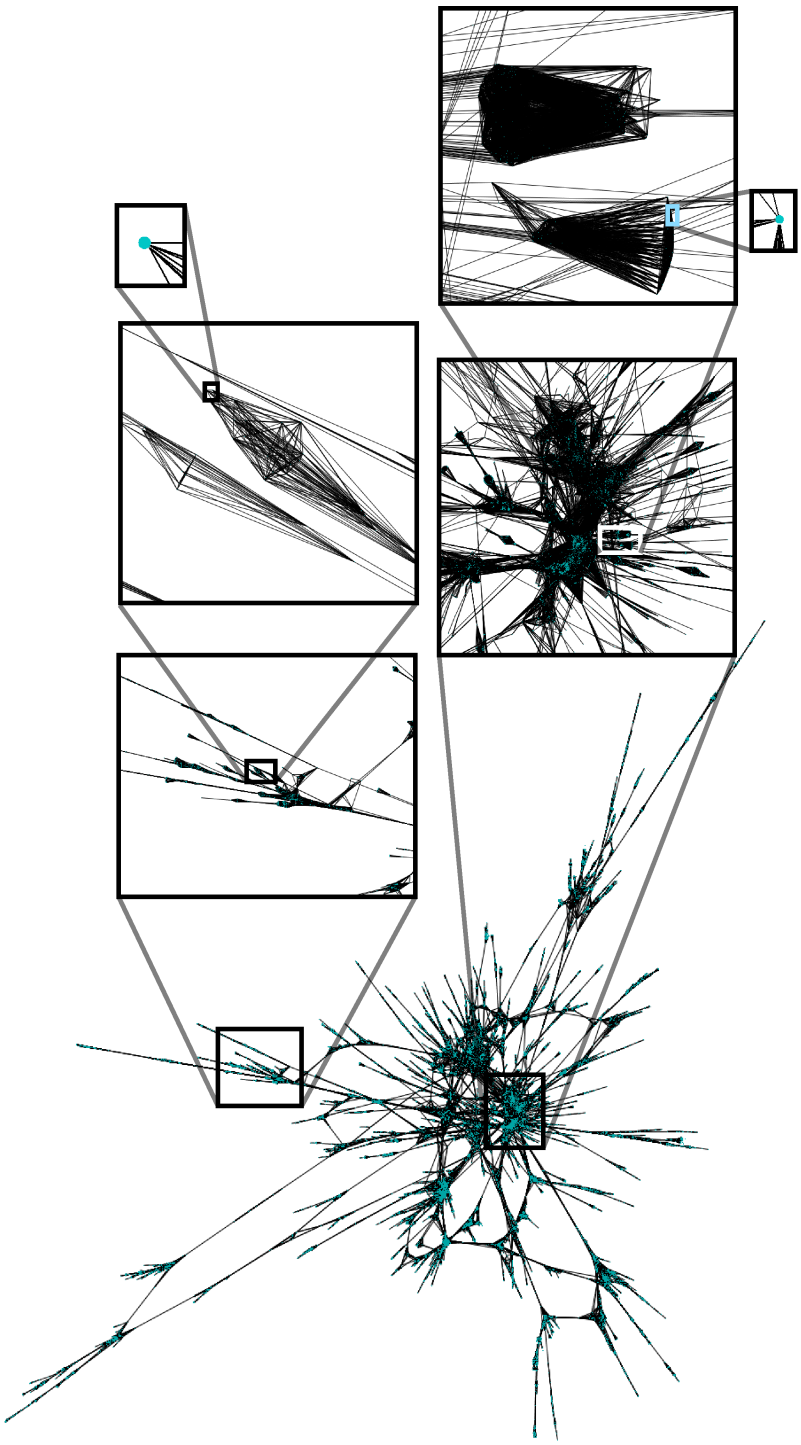


Figure 4.8: Drawing of `Protein`, a protein homology map obtained from the LGL project. Graph contains 30,727 nodes and 1,206,654 edges. Drawing produced by FM^3 in 1.7 minutes. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

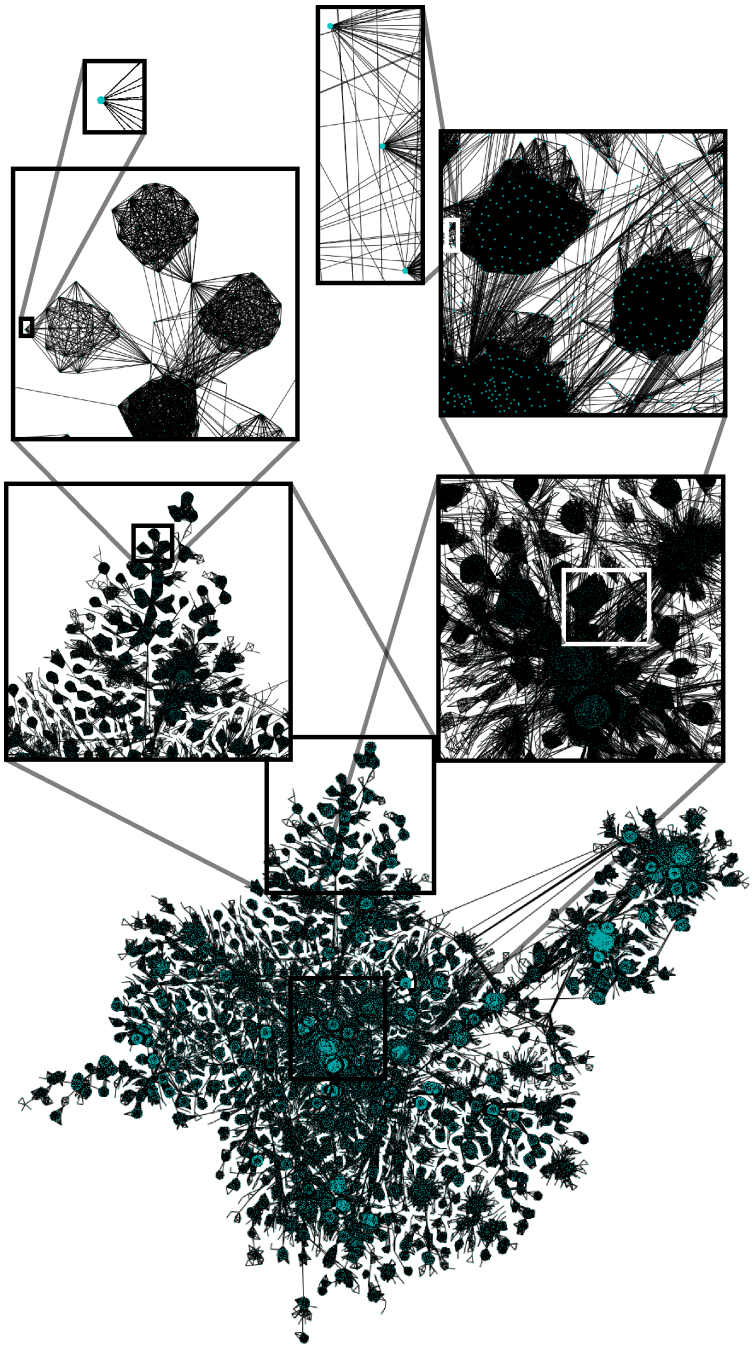


Figure 4.9: Drawing of `Protein`, a protein homology map obtained from the LGL project. Graph contains 30,727 nodes and 1,206,654 edges. Drawing produced by LGL in 1.4 hours. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

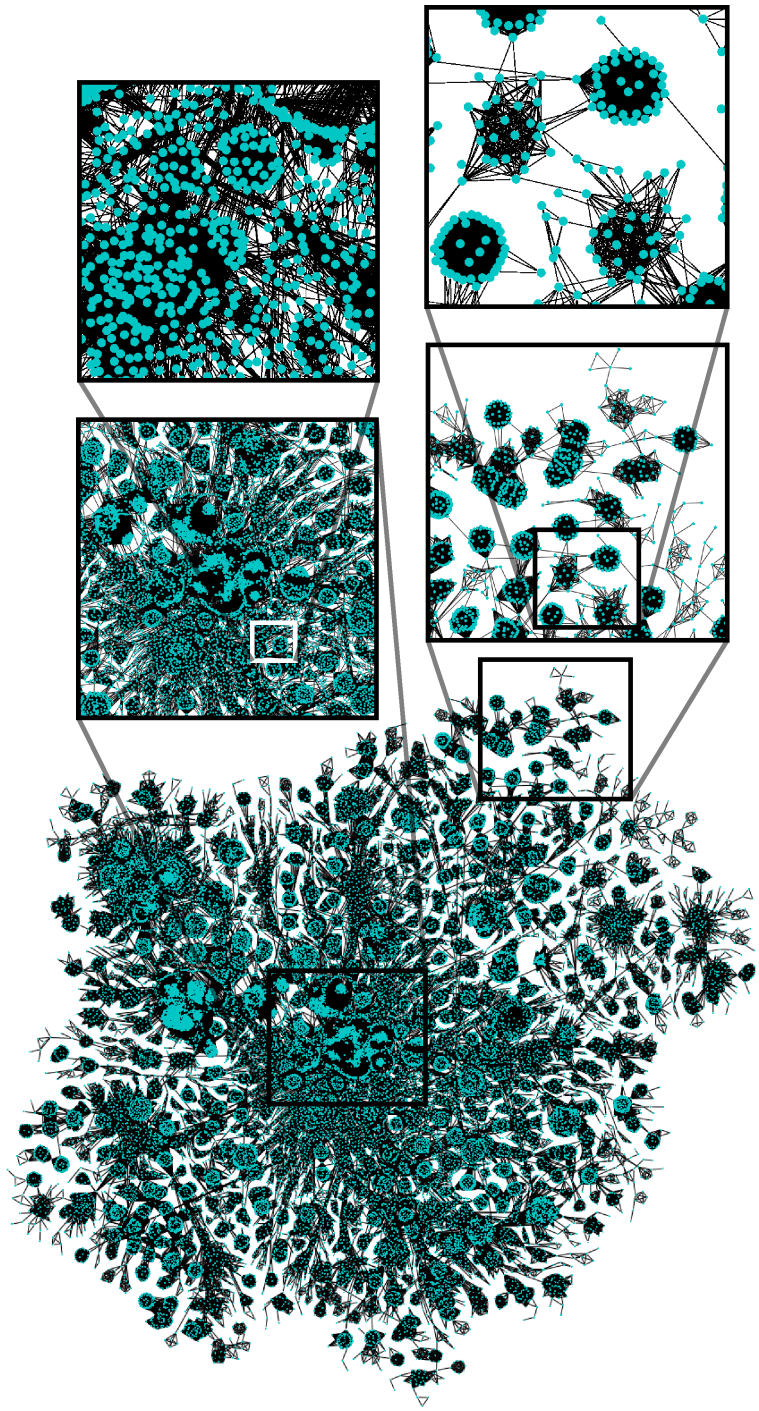


Figure 4.10: Drawing of `Protein`, a protein homology map obtained from the LGL project. Graph contains 30,727 nodes and 1,206,654 edges. Drawing produced by optimized LGL in 1.8 hours. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

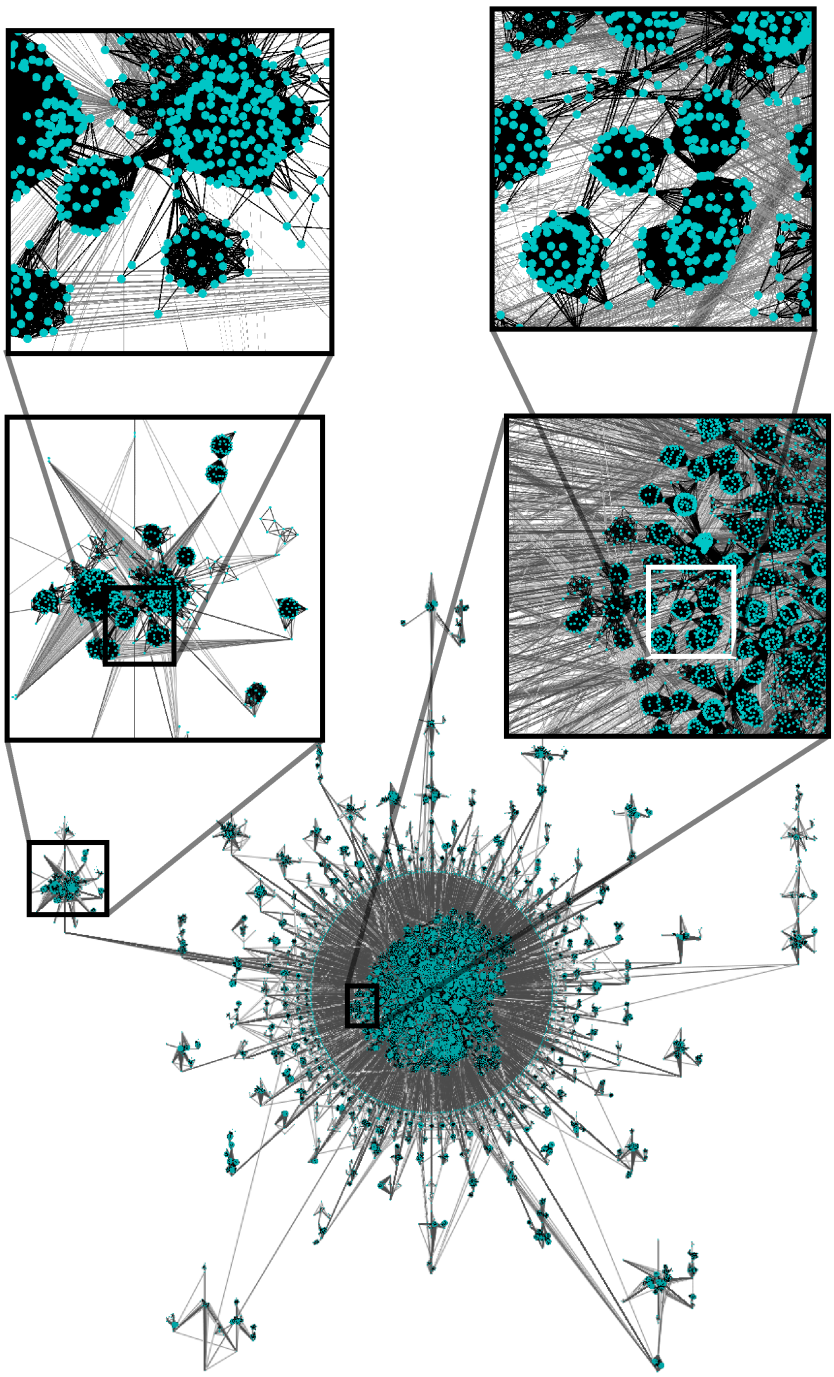


Figure 4.11: Drawing of `Protein`, a protein homology map obtained from the LGL project. Graph contains 30,727 nodes and 1,206,654 edges. Drawing produced by SPF in 43 minutes. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

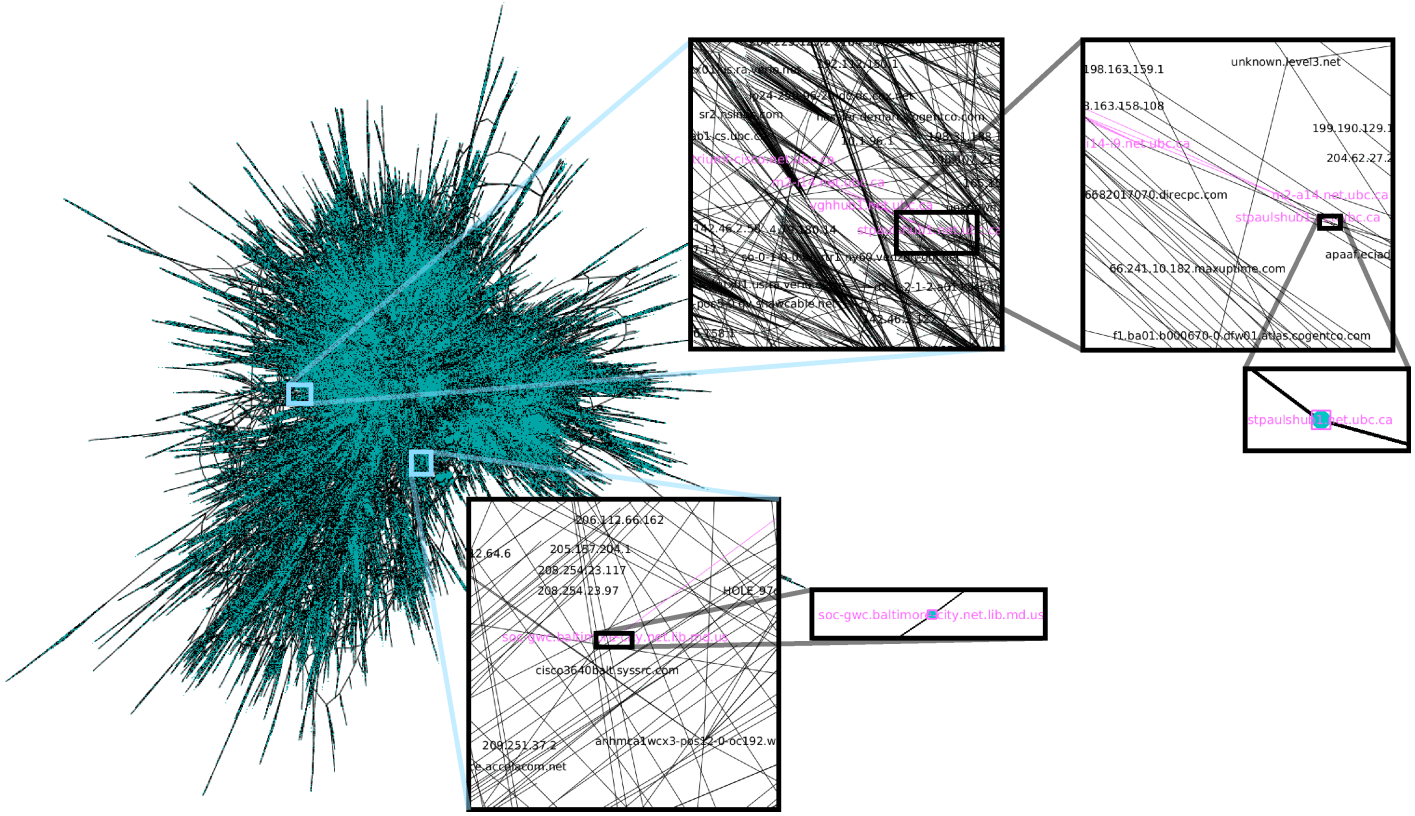


Figure 4.12: Drawing of `Net05`, obtained from the Internet Mapping project. Graph contains 190,384 nodes and 228,354 edges. Drawing produced by FM^3 in 11 minutes. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

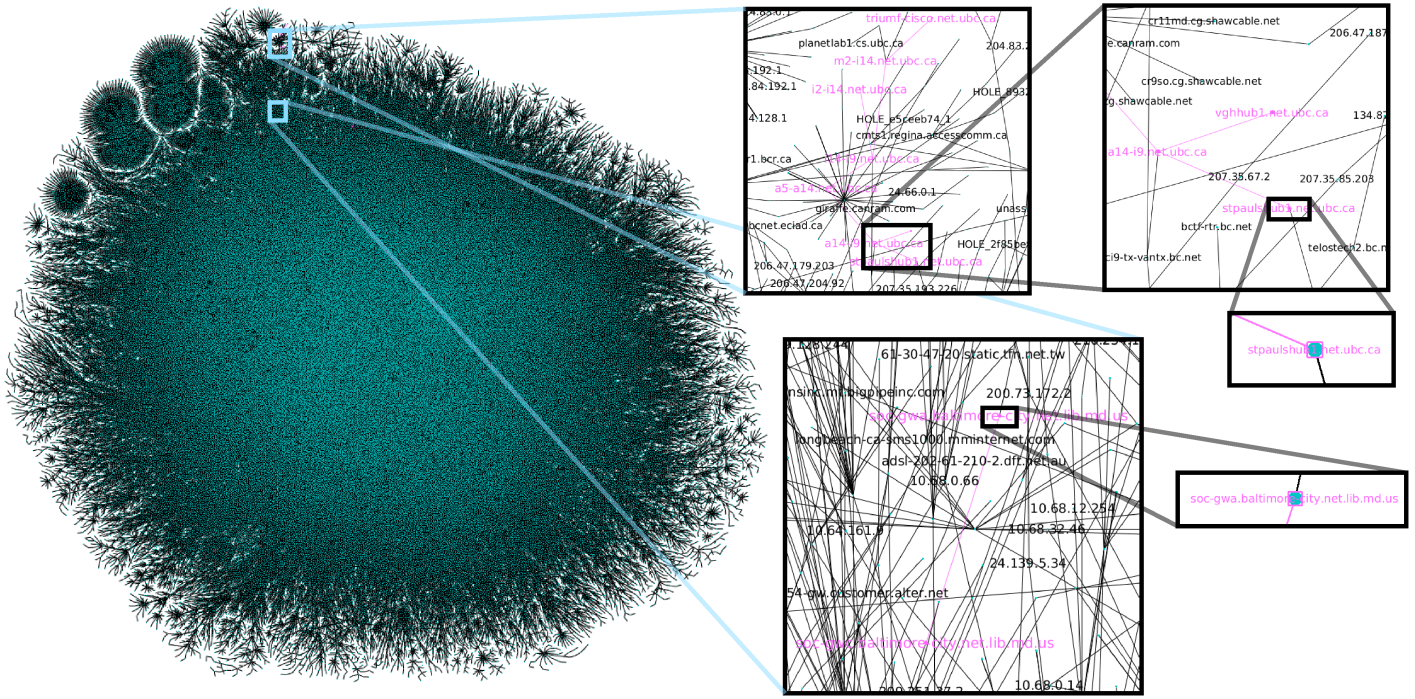


Figure 4.13: Drawing of `Net05`, obtained from the Internet Mapping project. Graph contains 190,384 nodes and 228,354 edges. Drawing produced by LGL in 12 hours. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

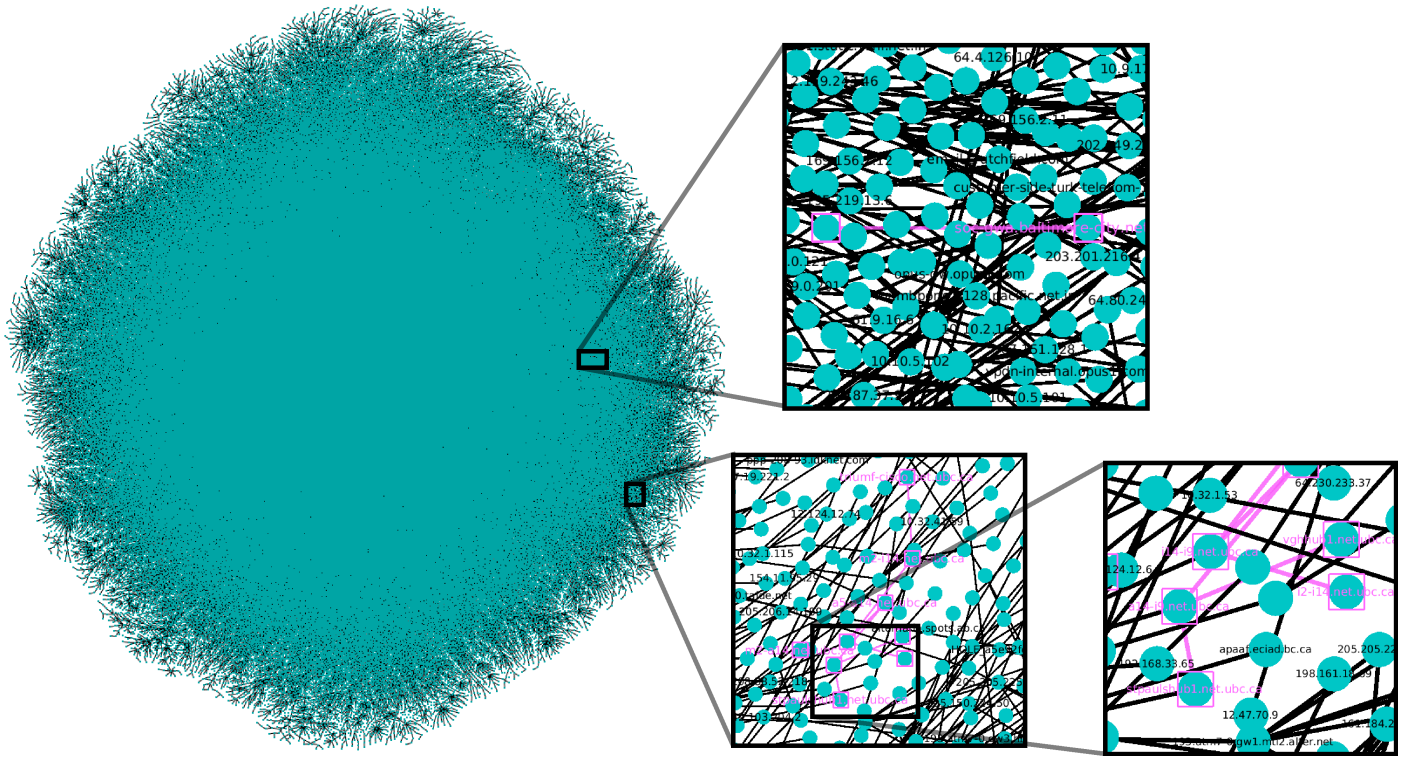


Figure 4.14: Drawing of `Net05`, obtained from the Internet Mapping project. Graph contains 190,384 nodes and 228,354 edges. Drawing produced by optimized LGL in 10 hours. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

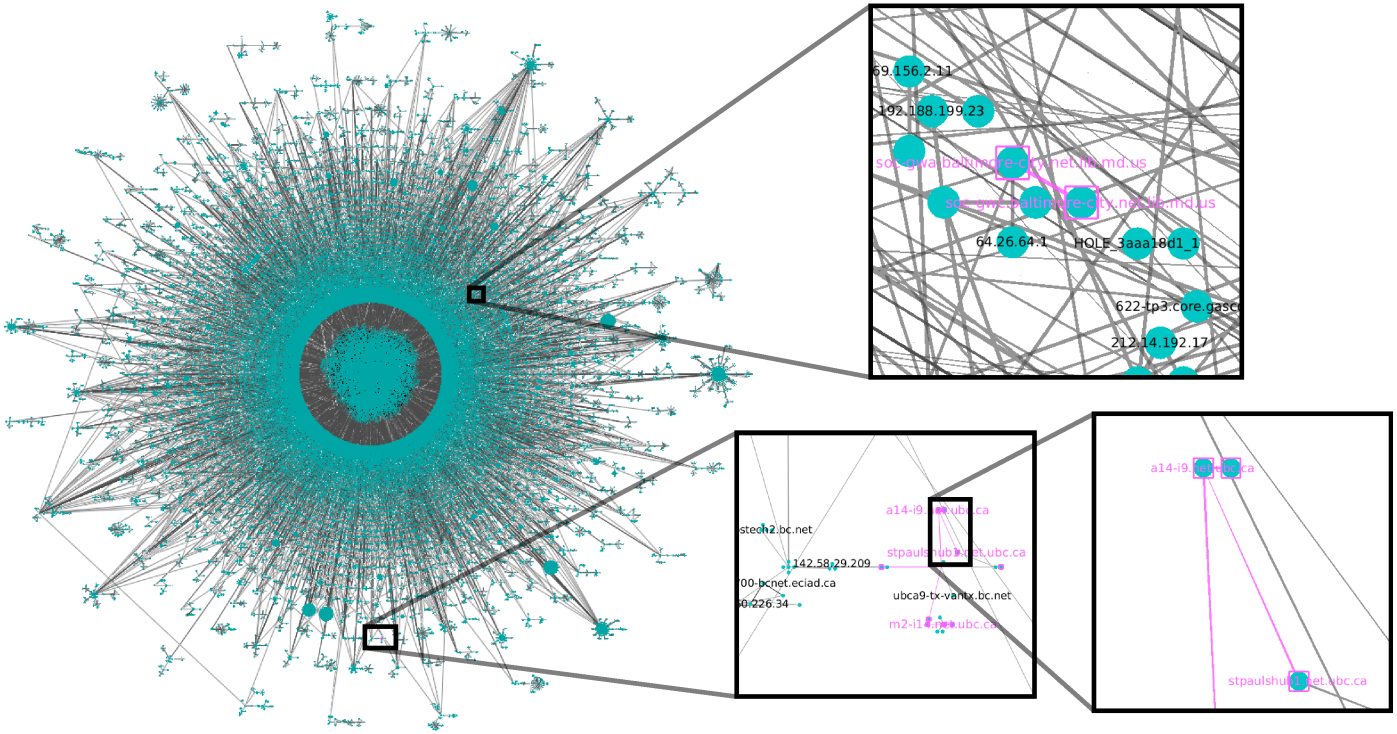


Figure 4.15: Drawing of `Net05`, obtained from the Internet Mapping project. Graph contains 190,384 nodes and 228,354 edges. Drawing produced by SPF in 30 minutes. A video that shows more detail in several areas of this layout is available at `www.cs.ubc.ca/labs/imager/video/2006/spf`.

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

Figure 4.16: Node-node overlaps. The table gives the total number of node-node overlaps, then only the number of major overlaps; that is, those where the overlap covers more than half of the node area. `Protein` has 30,727 nodes and 2,427 metanodes. `Net05` has 190,384 nodes and 167,460 metanodes.

4.2.3 Statistical Analysis

This section provides quantitative statistics for each of the four layouts. The number of node-node overlaps and uniformity of edge lengths are computed, both for the low-level structure of individual nodes and edges, and for the high-level structure of the metanodes containing biconnected components.

Node-Node Overlaps

A node-node overlap is simply the intersection of two nodes in a drawing. For the metanodes, a node-node overlap occurs when the two convex hulls of the biconnected components intersect. A smaller number of node-node overlaps in the metanodes more clearly displays high-level structure because the biconnected components do not overlap.

The node-node overlap statistics are presented in the top of Figure 4.16. The total number of overlaps is shown, and the number of major overlaps where more than half the area of the smallest node is covered. Major overlaps are more interesting than total overlaps as they affect the readability of the drawing more severely.

FM^3 has few node-node overlaps on either dataset. However, in this approach, the nodes are spread very far from each other with respect to the standard node size, leading to poor information density. This problem with

poor information density is that it is difficult to resolve low-level structure in the graph. LGL and optimized LGL incur more overlaps, but have the benefit of better information density. Optimized LGL incurs many more total node-node overlaps as the algorithm reduced the magnitude of the repulsive force constant. However, only a small percentage of them are major. With SPF, although `Protein` has a large total number of overlaps, the number of major overlaps is less than that for optimized LGL. A significant reduction in the number of node-node overlaps is realized on `Net05`. The use of area-aware RINGS to draw the very large number of biconnected components reduces the possibility of low-level node-node overlaps.

In the bottom of Figure 4.16, overlap statistics for the metanodes are presented. Metanode overlaps are more severe because features present in the graph, in this case biconnected component structure, are made difficult to read. FM^3, LGL, and optimized LGL have thousands of major overlaps. These major overlaps make it difficult to see the structure of the biconnected components in their higher level context. In these drawings, many but not all, of the large overlaps of metanodes are with the large, biconnected core that is spread through the drawing. It is important to note that optimized LGL is better able to separate the metanodes than the original LGL. This result supports the optimization of placing nodes on directed fans and reducing the repulsive force constant, keeping the nodes inside each metanode closer together. In contrast to these three methods, SPF succeeds in making these metanodes more evident. It incurs no overlaps at all for the smaller `Protein` dataset, and only one major overlap for the larger `Net05`. This statistic demonstrates an improvement in visualizing higher level structure in the graph.

Uniformity of Edge Lengths

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

The results are presented in Figure 4.17. The overall standard deviation is presented the left hand column. FM^3, LGL, and optimized LGL perform

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

Figure 4.17: Standard deviation of normalized edge lengths, where lower numbers mean more uniformity. Overall is the standard deviation over all edges. Within is the average standard deviation of the edges within each metanode, and Between is the standard deviation of edges that connect the components in the biconnected component tree.

well on this metric where SPF does not. From visual inspection of the SPF drawings, this additional variance is probably due to the long edges introduced by the area-aware RINGS algorithm. However, uniform edge lengths across the entire drawing may not be appropriate for displaying the biconnected structure of quasi-trees. In the drawings produced by LGL and optimized LGL, this property is illustrated through a uniform but featureless core. Noack [59] stated that long edges between the computed clusters of a graph may be required to display cluster structures. This thesis proposes that a more suitable metric for quasi-trees is to consider uniformity within a meaningful group; that is, the edges within a particular metanode, and the edges of the quasi-tree that connect between biconnected components.

Figure 4.17 shows these separate standard deviations in the centre and right columns. Optimized LGL is commensurate with LGL on nearly all numbers. SPF is commensurate with all algorithms in terms of the average standard deviation of edge lengths within each metanode on both datasets. SPF has a slight improvement for the between edges in `Protein`, but a very high standard deviation for `Net05`. This situation follows directly from the size of their biconnected component trees: small for the former, and large for the latter. The many concentric rings used by area-aware RINGS for large biconnected component trees contribute to this increased variance.

4.3 Robustness

We now discuss the robustness of SPF under slight changes to its input. As SPF only decomposes the graph into biconnected components, the decomposition phase is unique, regardless of node input order. However, LGL is a force-directed approach and therefore the positions of the nodes within a biconnected component can vary. Even though the nodes of the biconnected component tree are sorted by size, the size of each of these nodes can vary as well from the computed LGL layouts.

As the decomposition phase is unique, the features detected for a particular graph will be the same, regardless of node order. Although the coordinates of a given node may be very different from one drawing to the next, the levels of structure depicted is the same.

However, SPF uses a biconnected component decomposition. As a result, the addition of a single edge can drastically change the final drawing produced by SPF because the structure of the high-level tree changes as well. The argument can be made that such a change drastically affects the high-level tree structure and should be made apparent, but the chosen decomposition does negatively influence the robustness of SPF.

4.4 Discussion

SPF is a variant of the TopoLayout feature-based graph drawing algorithm tailored for the problem of drawing quasi-trees. According to the empirical evaluation on two real datasets, the algorithm improves on previous systems written in the computer networking and bioinformatics domains, more clearly illustrating the biconnected component structure of the quasi-tree and how those components are interconnected. These arguments are also supported using graph drawing aesthetics metrics on the final drawing and the multilevel structure. The system also improves upon the running time performance of previous approaches.

This work illustrates that a feature-based algorithm can be fruitful on domain specific data. SPF can resolve these features, especially in the Internet tomography dataset, because it relaxes the edge length uniformity and node uniformity constraints imposed by previous graph drawing approaches at high levels of structure.

The principal advantage of SPF is that the decomposition of the graph into biconnected components clarifies low-level structure. The algorithm clarifies this structure by ensuring biconnected component overlap is min-

imized. The principal disadvantage of SPF is that the area-aware RINGS algorithm can place elements of the graph that are close together in terms of graph-theoretic distance far apart in terms of Euclidean distance. In future work, additional area-aware tree drawing algorithms should be investigated so that biconnected components with directly adjacent nodes are placed as close together as possible.

However, the drawings still contain large amounts of node and edge occlusion. Also, it requires minutes of drawing time before exploration can begin. This is primarily due to the fact that simply displaying all the nodes and edges of a graph of hundreds of thousands of nodes may be too complex to draw quickly and intelligibly. In the next chapter, we begin to explore steerable graph visualization systems in order to address some of these limitations. Presented next is Grouse: the first of the steerable algorithms discussed in this thesis.

Chapter 5

Grouse: Steerable Exploration of a Feature-Based Graph Hierarchy

In both the TopoLayout and SPF algorithms, large graphs on the order of thousands of nodes take several minutes to draw. In order to circumvent this limitation, steerable techniques have been invented that allow for the progressive exploration of large graphs with an associated hierarchy [2, 20, 27]. None of these previous approaches, however, exploit specific algorithms tailored to the connectivity feature being drawn. With Grouse, we extend steerable graph exploration techniques in order to work with a hierarchy of connectivity features. This extension provides improved running time and visual quality in many circumstances.

In Grouse, we realize the above algorithm. Steerable exploration systems for a graph and associated hierarchy exist, such as ASK-GraphView [2] and DA-TU [27], but only force-directed algorithms are used. These systems must resort to automatic coarsening in order to reduce the size of a meta-graph. In Grouse, the algorithm draws parts of the hierarchy of connectivity features with the algorithms of TopoLayout in order to improve the speed and visual quality of the exploration.

The decomposition phase of Grouse is exactly the same as that of TopoLayout as described in Chapter 3, but it is computed beforehand and is supplied as input to the system. However, the drawing of the graph is steerable and directed by the user. Steerability is accomplished by modifying the hierarchy cut as described in Section 1.1.5.

The contribution of Grouse is an algorithm for the steerable exploration of a graph and an associated graph hierarchy of connectivity features. Additionally, a re-layout algorithm is presented that ensures nodes in the final drawing closer to their input size. We implemented Grouse and compared

it empirically to previous systems. Grouse was published in *The Proceedings of the 2007 Eurographics/IEEE-VGTC Symposium on Visualization* [6]. A video[11] shows the exploration of two datasets using this algorithm. An implementation of the algorithm is available online[12].

This chapter is structured as follows. Section 5.1 describes Grouse system architecture at a high level. Section 5.2 presents details of each of the system architecture components. Results of the system working on subsets of the Internet Movie Database (IMDB) are presented in Section 5.3. A discussion of algorithm robustness is presented in Section 5.4. Finally, a discussion of the contributions of the Grouse system is presented in Section 5.5.

5.1 Algorithm Overview

The Grouse interface consists of two linked views, as shown in Figure 5.1. On the left is the **tree view** window showing all open, hidden, and cut metanodes and leaves in the graph hierarchy. as a list view widget. It supports tree browsing with the standard interaction of expanding or collapsing items and vertical scrolling. On the right is the **graph view** window, showing the current cut to the graph hierarchy with a node-link diagram. It supports standard pan and zoom through the two-dimensional view. The graph hierarchy used for the visualization is supplied with the input graph. In Grouse, this hierarchy is generated by the decomposition phase of Topo-Layout described in Chapter 3.

The system has visual encodings for open and cut metanodes. Hidden metanodes are not visible in the graph view. As the detail of these nodes is not visible, the algorithm does not need to draw their subgraphs. Open metanodes show the subgraph beneath the metanode in the hierarchy contained within its enclosing circle. Cut metanodes are drawn as an opaque hexagon without further detail. Containment is used to show the structure of the graph hierarchy. The algorithm shows graph relationships by drawing edges from leaf nodes in the usual way and by drawing metaedges between two cut metanodes or a cut metanode and a leaf node, if there is an edge between any leaves beneath the metanodes in the graph hierarchy. Leaf nodes are drawn as grey boxes in the graph view.

The algorithm colours metanodes by the feature type they contain, inspired by Stone's Tableau Software colour scheme [69]. Nodes that contain biconnected component trees are brown, trees are blue, sets of clusters are

[11] `http://www.cs.ubc.ca/labs/imager/video/2007/Grouse/GrouseVid260070p.mov.gz`
[12] `www.cs.ubc.ca/labs/imager/tr/2007/Archambault_Grouse_EuroVis`

yellow, cliques are cyan, meshes are purple, and all other components are green. The saturation of the base colour represents depth in the hierarchy, so that the large disks are less saturated and the smaller ones are more brightly coloured. Grouse supports linked highlighting between views. In the tree view, the current hierarchy cut is shown within the context of the entire hierarchy through colouring the label backgrounds of open metanodes according to their feature type. Nodes are selectable in either view. The nodes are highlighted with a red perimeter in the graph view and a red background label in the tree view.

The main interaction is the opening of a metanode. The subgraph beneath it is inserted into the graph at the location of the metanode, and the newly inserted nodes move in a smooth transition to their locations in the new layout. Their enclosing metanode appears at a depth proportional to its depth in the hierarchy. A cut metanode can be opened by clicking within its hexagon in the graph view or by clicking on its name in the tree view. A hidden metanode can be opened by clicking in the tree view. This action triggers a multi-stage animation, showing the successive opening up of all enclosing metanodes on the path from its ancestor in the cut down to the desired node. All the metanodes present in a subtree below a metanode in the graph view can be opened by holding down the shift key and clicking a closed metanode.

5.2 Algorithms

Grouse is built on the Tulip framework libraries [9]. The approach uses many of its data structures, graph drawing algorithms, and rendering capabilities. In the first section, feature-based layout is described along with how we generalize the crossing minimization phase to work in this steerable system. The second section presents the core algorithm used to support the animated metanode opening operation. Two other algorithmic issues are discussed: minimizing change during a re-layout and morphing node locations and sizes during animated transitions.

5.2.1 Layout and Crossing Reduction

Grouse adapts the feature-based layout of TopoLayout described in Chapter 3 for steerability. As discussed in that chapter, it is a challenge to provide good drawing density that allows structure at many levels to be seen simultaneously. To obtain high visual quality, the algorithm must estimate the size of the graph layout at each level and set a corresponding size for each

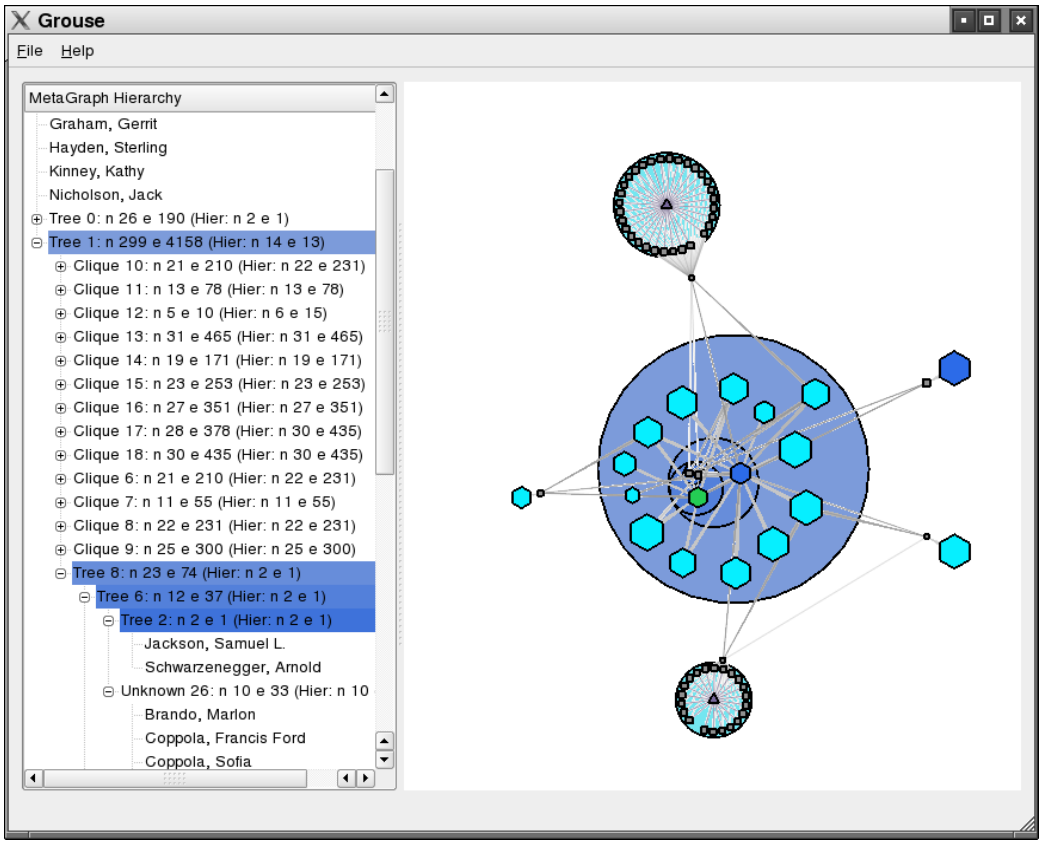


Figure 5.1: The Grouse interface has two linked views, both showing the selected node in red. On the left, all metanodes and leaves in the graph hierarchy in the tree view are seen. On the right, the graph view represents closed metanodes on the hierarchy cut with hexagons, and the open metanodes above them in the hierarchy as circles. The hierarchy structure is shown in the graph view by containment within nested circles, while graph structure is shown with connecting edges and metaedges. Metanodes are coloured according to the feature they contain with leaves shown as grey boxes.

metanode in the hierarchy. Thus, the best results come from generalizing algorithms to be area-aware. After this initial layout step, the algorithm executes overlap elimination and crossing reduction passes.

In TopoLayout, the final crossing reduction pass reduces, but does not completely eliminate, edge-edge and edge-metanode crossings by rotating metanodes according to a computed **torque** value. The generalization of this approach to steerable exploration is straightforward: the algorithm determines the forces using the cut metanodes, rather than the leaves of the hierarchy. The complexity of the generalized crossing reduction is $O(|N|E_v)$, where E_v is the number of visible edges in the current hierarchy cut.

Grouse also introduces a small improvement in the torque computation to better avoid the local minimum where equal and opposite torques average out to zero torsional force. The algorithm also computes the average absolute value of the torque. If this average is larger than 90°, the algorithm checks to see if rotating the node by 180° improves the drawing. If the average torque decreases, the algorithm keeps the node flipped; otherwise the algorithm leaves it unchanged.

5.2.2 Changing the Hierarchy Cut

In Grouse, when the user opens a metanode, the hierarchy cut changes and the system needs to update the graph view. The selected node changes type, moving above the hierarchy cut to become an open metanode. All formerly hidden nodes in the subgraph of the metanode are added to the hierarchy cut. Duplicated nodes that may have been added to a metanode during a biconnected component decomposition are not added to the cut. However, they are considered when the subgraph contained by a metanode is drawn. Figure 5.2 shows how metanodes are updated with a change in the hierarchy cut. Pseudocode for the algorithm is provided in Figure 5.3.

In this incremental layout approach, the size of all the metanodes in the graph hierarchy starts with a diameter of $\sqrt{N_m}$, where N_m is the number of leaves in the subtree beneath the metanode m. This default value causes the area of the metanode be roughly proportional to its number of leaves. When the user opens a metanode for the first time, the subgraph contained within it is laid out, leading to a change in the size of its open metanode. The algorithm has more information about the space requirements at each level because this size estimate is further refined as metanodes inside are opened. At a leaf node, the algorithm has perfect information about the size required.

After a layout event, the size of a metanode is changed, and the layout of

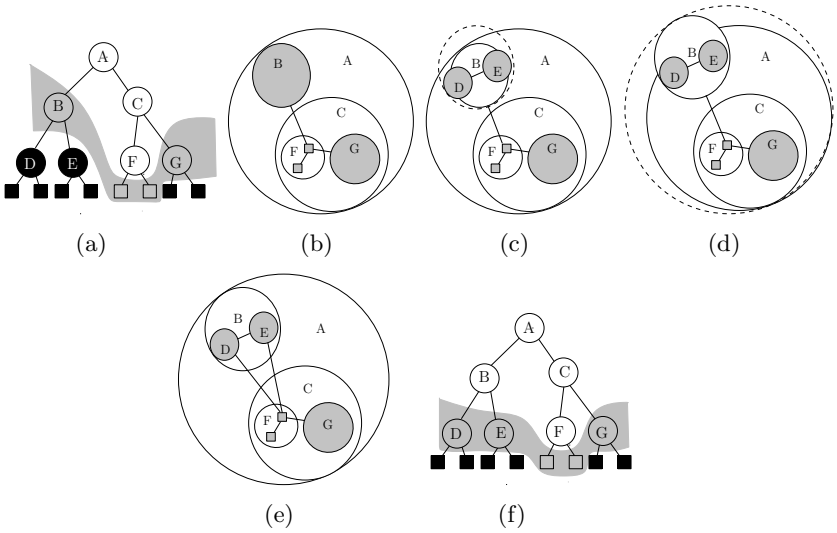


Figure 5.2: Example of the computations made when changing the hierarchy cut by opening a metanode. Open nodes are white, cut nodes are grey, and hidden nodes are black. **(a)** The initial hierarchy cut shown in the context of the whole hierarchy. It has open metanodes A, C, and F. Metanodes B and G are in the hierarchy cut, as are the leaves below F. The metanodes D and E are hidden, as are their leaves. **(b)** The initial hierarchy cut as shown in the graph view. **(c)** After the selection, metanode B changes from cut to open and the two formerly hidden metanodes D and E become cut metanodes. The subgraph containing D and E is laid out and the size of B is updated. **(d)** The subgraph inside A is laid out, and the size of A is updated. **(e)** The metaedge to B is deleted and two edges to D and E are added. **(f)** The final hierarchy cut, as a tree. The animated transition seen by the user is a linear interpolation between views (b) and (e). The intermediate stages of computation are not visible to the user.

```
proc ChangeCut (openedMetaNode)
  metanode = openedMetaNode
  while metanode != NULL
    InitialLayout(metanode)
    RemoveOverlaps(metanode)
    MinimizeCrossings(metanode)
    RecalcBoundingCircle(metanode)
    metanode ← metanode.parent
  end while
  RecomputeMetaEdges()
```

Figure 5.3: Pseudocode for the ChangeCut algorithm.

each subgraph along the path between the metanode and the root is updated. This cascading re-layout does not require or recompute the layouts for any of the other nodes in the cut. The worst-case number of re-layouts is thus $O(d)$, where d is the maximum depth of the cut. When the hierarchy is close to balanced, this depth is logarithmic in the number of nodes. Moreover, the cut depth is small when exploration begins. The complexity of the entire re-layout cascade depends on the features present in the metanodes on the path, since the layout complexity for each metanode ranges from linear to cubic.

After all nodes on the path up to the root are re-laid out, the algorithm must delete the metaedges incident to the parent metanode that is removed from the hierarchy cut. The algorithm must then add new metaedges for the newly laid out nodes in the child subgraph that have been added to the hierarchy cut. A metaedge exists between cut metanodes if there is an edge between leaves that the cut metanodes contain. The algorithm uses an interactive refinement approach [11] to compute these metaedges on the fly in linear time.

5.2.3 Minimizing Change in the Layout

The re-layout phase discussed above is one example of the dynamic or incremental layout problem, where the algorithm would like a new layout of a graph to be as close as possible to the old layout of a similar graph, with node size and connectivity changes. For most of the algorithms, the re-layout phase can eliminate unnecessary change by preserving the order that the nodes and edges are processed when a metanode is re-laid out.

However, simply preserving order does not solve this problem with GEM

force-directed placement. The algorithm uses the old layout as an initial guess for the new one. This approach works well when the size change of the opened metanode is small. The new layout might be quite different from the old when the sizes of the nodes in the subgraph change drastically. A node that has a large change in size requires more space in the layout, pushing nearby structure further away. Situations where node size changes substantially is currently a limitation of the system presented here, but it could potentially be overcome using other information visualization techniques.

5.2.4 Animating Transitions

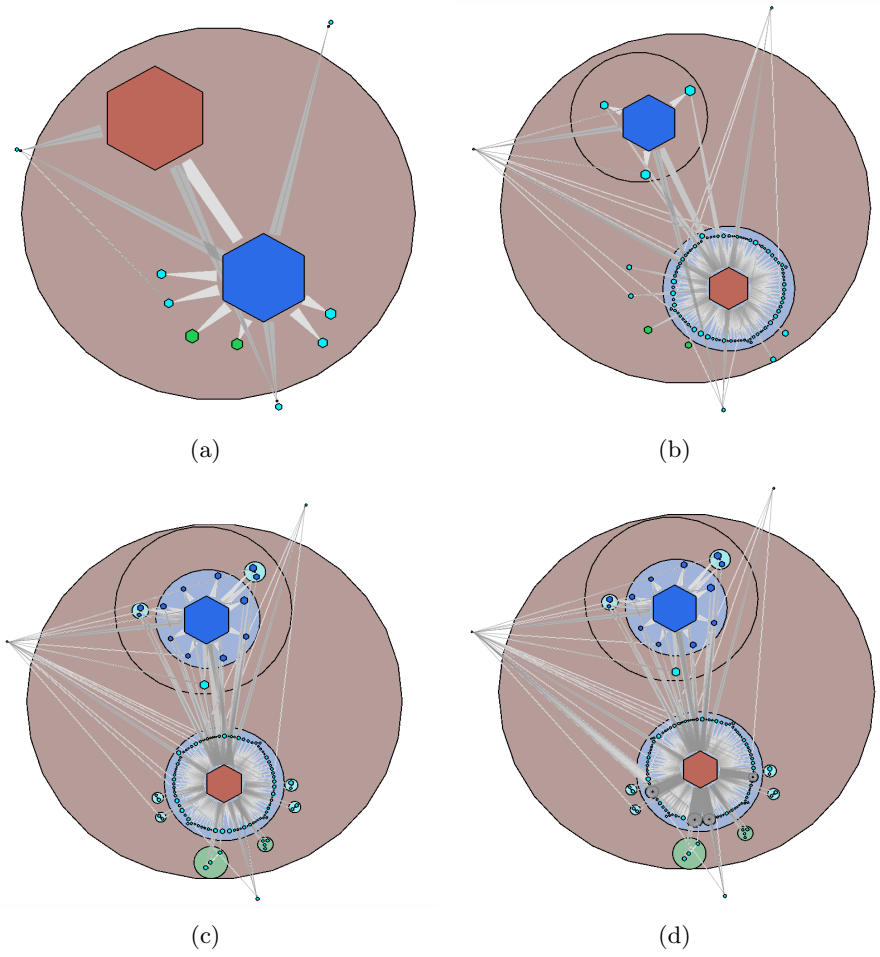
Opening a metanode can lead to size and position changes in any metanode along the path from the node being opened to the root of the hierarchy. It would be difficult to understand those changes if they were made abruptly. Grouse linearly interpolates between the old and new coordinates of the nodes while preserving containment relationships. The change in position is presented as an animation over time that moves the node along its linearly interpolated path from its old position to its new position. During this transition, the sizes of the open metanodes are morphed as well. In Figure 5.2, the old positions of the nodes are the positions presented in Figure 5.2(b) while the new positions for the nodes are the positions presented in Figure 5.2(e). The change in position between the old and the new positions of the nodes in the hierarchy cut is not shown in the figure.

5.3 Results

This section presents several layouts showing stages in the steerable exploration process. The datasets are subsets of the Internet Movie Database (IMDB) and are presented in Figures 5.4, 5.5, 5.6, and 5.7. This section discusses the benefits of preserving the input hierarchy. The Grouse algorithm is supported qualitatively by comparing the information density of the resulting layouts, comparing the scaling method to the re-layout method, and force-directed layout to feature-based layout.

5.3.1 Steerable Exploration

The input graph for both Figures 5.4 and 5.5 is the connected component containing Sharon Stone movies made in 1999. The nodes in this graph are actors and two nodes share an edge if the actors have acted in a movie



(a) (b) (c) (d)

Figure 5.4: Exploring an IMDB dataset of Sharon Stone movies from 1999, where the full input graph is 7,640 nodes and 277,029 edges. These four snapshots of the graph view show user exploration of the hierarchy.

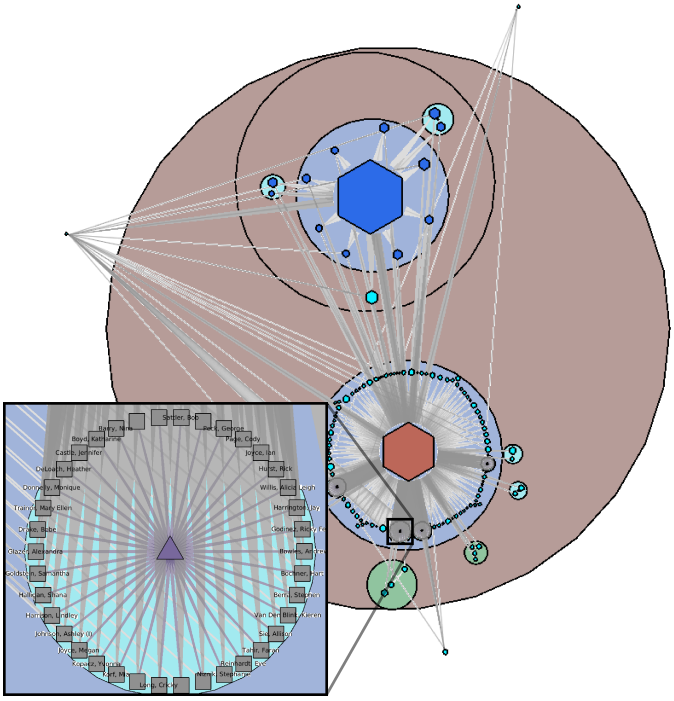


Figure 5.5: Exploring an IMDB dataset of Sharon Stone movies from 1999, where the full input graph is 7,640 nodes and 277,029 edges. The figure shows the final step of exploration. The cyan clique of actors in the movie *Anywhere but Here* is shown in the inset. Labels have been turned on in the inset view, showing actor names.

together. The graph has 7,640 nodes and 277,029 edges. Figures 5.4(a) through 5.4(d) show a sequence of snapshots of the graph view as metanodes are opened. In the final image on the right, the hierarchy cut includes a large high-level tree, containing a clique representing a movie (in cyan) near the bottom. The inset shows a close-up view of the clique with node labels.

5.3.2 Preserving Hierarchy Features

In the IMDB datasets shown in Figures 5.5 and 5.6, cliques represent movies. A movie is a clique because there exists an edge between every pair of actors in a cast. Actors that have acted in multiple movies appear between the cliques, as a biconnected component decomposition or clustering algorithm would segment them out. Algorithms that use hierarchy modifications in

the style of ASK-GraphView [2] would split up these cliques into multiple metanodes, thus making them difficult to recognize. In Grouse, these features are preserved and drawn using a glyph. The algorithm can do this efficiently because appropriate layout algorithms are used depending on the connectivity feature present in the subgraph. The system shows the hierarchical structure above the hierarchy cut explicitly through containment. The bounding circles drawn below open metanodes depict in a single view the hierarchical structure above the hierarchy cut.

5.3.3 Scaling vs. the Re-Layout Algorithm Along a Path

In Grouse, the algorithm recomputes part of the layout during exploration, at most d metanodes where d is the depth of the graph hierarchy. As the user steers the layout, the metanode size estimates improve and the cascading re-layout propagates this information up toward the hierarchy root. This approach represents a quality-for-time trade off compared to the ASK-GraphView method, where layout of any subgraph only occurs once and is scaled so that it fits into the open metanode on the previous level. Although the scaling approach is fast, major features can be difficult to perceive. Figure 5.6 compares these two approaches on one example. In the scaling example of Figure 5.6(a), the cyan clique is much smaller than in the re-layout example of Figure 5.6(b). Additionally, more levels are visible at a single scale in the drawing using the re-layout technique.

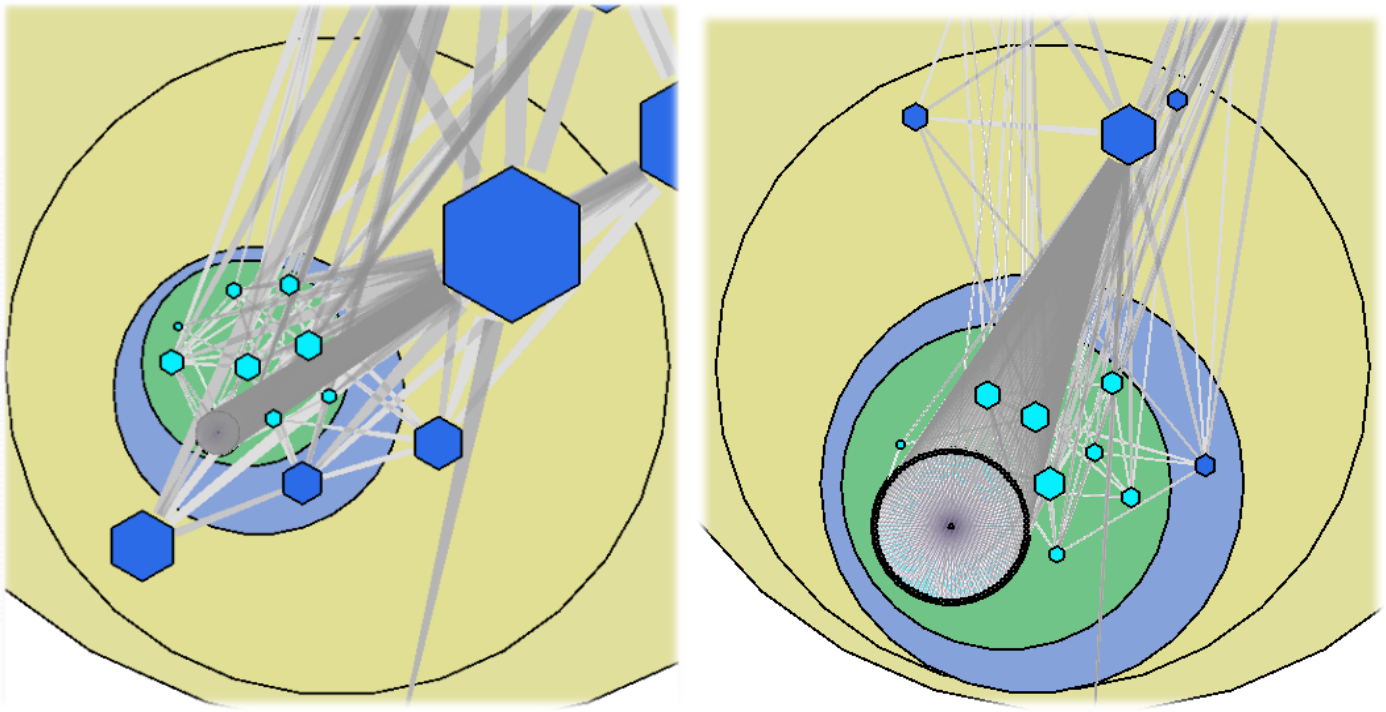
5.3.4 Force-Directed vs. Feature-Based Layout

Figure 5.7(a) shows a layout using only force-directed placement, as in previous steerable systems. This layout is compared qualitatively to Figure 5.7(b) that shows the feature-based layout of Grouse where appropriate algorithms are chosen for the type of connectivity feature present in the subgraph. With Grouse, it is easier to see the tree at the center of the drawing because of its shape. The circular drawings and glyphs inform the user that the cliques are actually complete. As movies are cliques, this indicates where movies are in the graph. In the force-directed drawing, the spacial layout does not provide such explicit cues. Thus, it is more difficult to really know if the metanodes contain movies.

5.4 Robustness

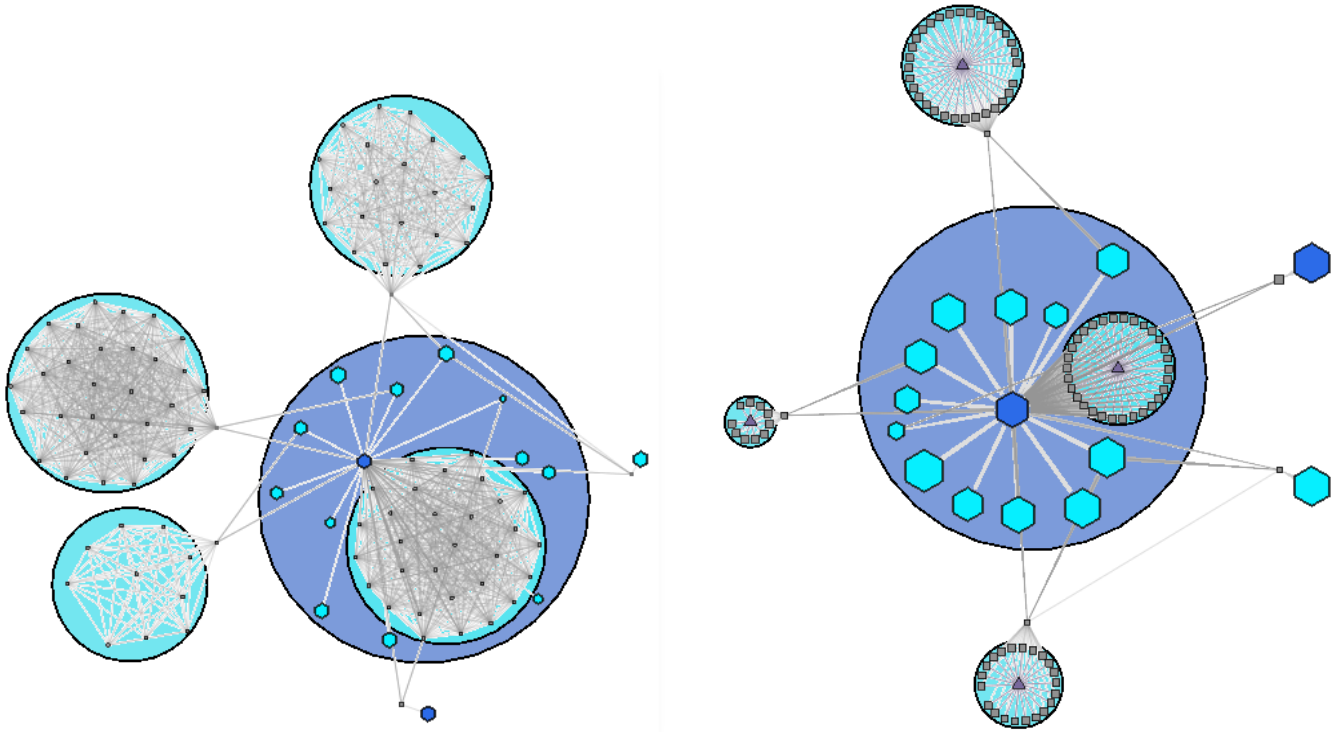
As Grouse uses TopoLayout hierarchies as input, the robustness constraints for the graph hierarchy are identical to those present in TopoLayout. The reader can refer to to Chapter 3 for further information.

The force-directed graph drawing algorithm used in our steerable graph drawing approach does not guarantee the same drawing on the same input. This limitation is problematic during a re-layout step. We are able to reduce, but not completely eliminate, changes to the drawing of the feature through a force-directed approach. This reduction is accomplished by initializing the node positions to the previous layout before executing a re-layout. More advanced techniques to ensure the stability of force-directed layouts exist [32] but have not been integrated into Grouse.



(a) scale (b) re-layout

Figure 5.6: Number of levels of the hierarchy that can be seen simultaneously in a comparison between scaling and re-layout techniques. With scaling **(a)**, it is difficult to see deep into the hierarchy. Using the re-layout algorithm **(b)**, more levels are distinguishable. Having more levels of the hierarchy visible at once facilitates analysis of the hierarchy across many levels of structure. The dataset presented is a subset of IMDB with 1,181 nodes and 31,527 edges centered around Jake Gyllenhaal.



(a) (b)

Figure 5.7: A comparison of information density using force-directed layout with feature-based layout. **(a)** In the force-directed case, the leaf nodes in the cyan clique are so spread apart that they are tiny. **(b)** With feature-based layout, the clique leaves are far larger, as are the blue cut metanodes representing trees in the open metanode at the center.

5.5 Discussion

Grouse uses graph hierarchy cuts to present simpler views of TopoLayout hierarchies. It does so in a steerable way where users do not need to wait for periods of time ranging from minutes to hours before exploration can begin. If the purpose of the investigation is to understand the types of connectivity features present in the data, Grouse provides a good solution. In the case of the IMDB examples presented in this work, Grouse better depicts areas of the graph that contain the cast of a movie.

One of the limitations of Grouse is that the force-directed algorithm used during the re-layout step may produce large changes in the positions of nodes. Instability in the positions of the nodes makes it difficult to understand how the hierarchy cut is changing during exploration of the data. To minimize this layout instability, the system could use the constraint-based graph drawing techniques of Dwyer *et al.* [22]. Also, the system could use an adaptation of dynamic graph drawing approaches such as Frishman and Tal [32].

However, more importantly, Grouse is limited to a single hierarchy of connectivity features during the exploration phase. Using Grouse, users cannot tailor the graph hierarchy. As such, if the high or mid-level structure they wish to investigate is not based on connectivity features, then Grouse cannot depict it. In the next chapter, GrouseFlocks begins to address this limitation by proposing a steerable technique for exploring the many hierarchies that can be created using attribute data associated with the nodes of the graph.

Chapter 6

GrouseFlocks: Steerable Exploration of Graph Hierarchy Space

In Grouse, a single hierarchy of connectivity features is used to explore the input graph. However, in most circumstances, attributes are associated with the nodes of the graph. These attributes can be used to construct a variety of hierarchies on top of the graph, providing different views of the underlying graph. For example, consider computer networking where nodes are servers and edges are connections between those servers. Attributes associated with the nodes of the graph group these servers into subnetworks. By grouping subnetworks together, we can reveal high-level structure associated with the graph. This structure would not generally be revealed by the single hierarchy of connectivity features supplied as input to Grouse. GrouseFlocks extends Grouse to allow for the steerable exploration of the space of graph hierarchies that can be constructed on top of a graph using attribute data.

In GrouseFlocks, we realize a steerable system for the space of graph hierarchies that can be created using a graph with attribute data. Both the decomposition and drawing phases of this algorithm are steerable. The majority of previous steerable systems do not support hierarchy creation [2, 20]. The few systems that do support hierarchy creation only do so through manual selection [11, 27] and the remainder do not support steerable exploration of the resultant hierarchy [61]. GrouseFlocks supports the steerable creation of graph hierarchies through attribute data present on the nodes of the graph. Different hierarchies provide different views of the graph. These different views allow the user to reason about its structure. This exploration of graph hierarchy space is constrained to be path-preserving. The constraint ensures that if a path appears in a cut, it also appears in the underlying graph. GrouseFlocks also introduces a coarsening algorithm that preserves large metanodes in the created hierarchy.

The contribution of GrouseFlocks is a path-preserving technique for the

steerable creation and exploration of graph hierarchy space based on attribute data associated with the nodes of the graph. The approach also introduces a coarsening algorithm that preserves large metanodes in the created hierarchy. We implemented GrouseFlocks and compared it empirically to previous systems. GrouseFlocks will appear as a *Transactions on Visualization and Computer Graphics* [8] article. An implementation of GrouseFlocks is available online[13].

This chapter is structured as follows. Section 6.1 introduces the concept of a path-preserving hierarchy. This concept is critical for depicting paths in a cut if at least one path exists in the underlying graph. In Section 6.2, the GrouseFlocks user interface is presented along with the operations the user performs to explore graph hierarchy space. In Section 6.3, the algorithmic details behind steerable hierarchy modification and coarsening are presented. Several graph hierarchies are constructed and contrasted in Section 6.4. A discussion of algorithm robustness is presented in Section 6.5. Finally, Section 6.6 discusses the contributions of GrouseFlocks and possible future directions.

6.1 Path-Preserving Hierarchies

A **path-preserving** graph hierarchy is one where a path in the hierarchy cut, consisting of a specific sequence of metanodes, exists if and only if there exists a path in the input graph, consisting of a specific sequence of leaves, such that the leaves are descendents of the metanodes. Path-preserving hierarchies are needed to ensure that all paths viewed in a cut actually exist in the underlying graph. They have been used previously in graph drawing systems [29], but we are unaware of their use in steerable hierarchy creation and exploration. Path-preserving hierarchies must respect two properties:

Definition 1. Edge Conservation: *An edge exists between two metanodes m_1 and m_2 if and only if there exists an edge between two leaves in the input graph l_1 and l_2 such that l_1 is a descendant of m_1 and l_2 is a descendant of m_2.*

This property is illustrated in Figure 6.2. In Figure 6.2(a), an edge exists in some upper level of the hierarchy between two metanodes. Figure 6.2(b) is valid since edges, drawn in red, exist between nodes in the input graph. Figure 6.2(c) is not valid because no such edge exists in the input graph.

[13] www.cs.ubc.ca/labs/imager/tr/2008/Archambault_GrouseFlocks_TVCG/

Intuitively, any edge that exists in a cut is an abstraction of one or more edges that existed in the input graph.

Definition 2. Connectivity Conservation: *Any subgraph contained inside a metanode must be connected.*

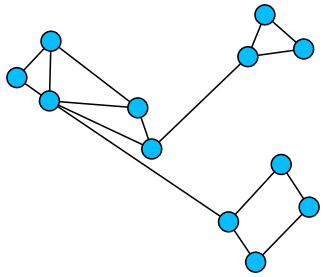
This second property ensures that a path through a subgraph always exists. If the user were allowed to place disconnected subgraphs inside a metanode, as shown in Figure 6.3(c), the metanode would imply that a path exists between the subgraphs. This placement implies the existence of a cycle that does not exist in the input graph. If we respect this restriction, as in Figure 6.3(b), these problems do not occur, as there is only one connected component in the subgraph.

By respecting these two properties, a path-preserving graph hierarchy is guaranteed. Edge conservation ensures that any hierarchy edge is witnessed by some edge in the input graph, while connectivity conservation ensures that any path in the cut can pass through any metanode. Therefore, any path that exists in a cut can be mapped to a sequence of leaves that exist in the input graph.

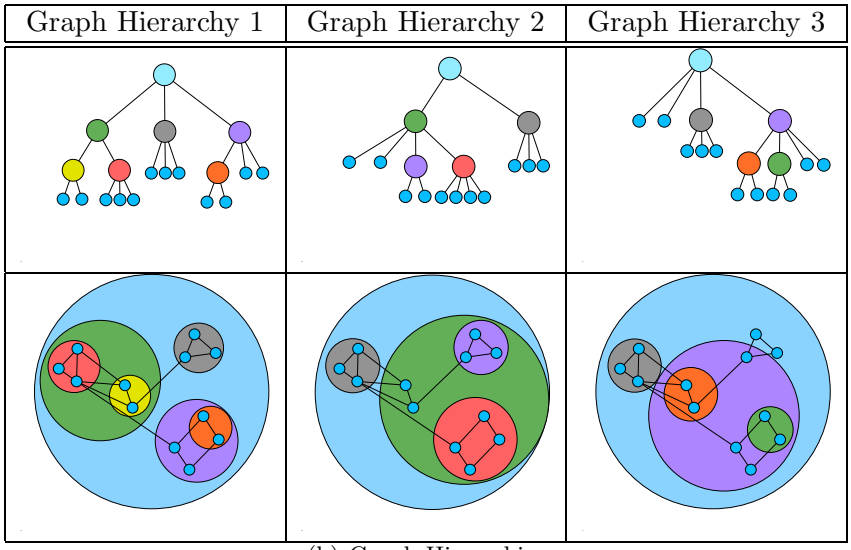
Even with the path-preserving restriction, there can still exist an exponential number of hierarchies for a single input graph. Figure 6.1(a) presents a possible input graph, and Figure 6.1(b) gives three examples of possible hierarchies that can be superimposed on this input graph. In the first row of the table are depictions of hierarchy trees. In the second row of the figure, hierarchy trees are superimposed on top of the input graph using containment, where a circular metanode contains all its children. This simple example demonstrates that the metanode structure imposed on the same graph can differ wildly from hierarchy to hierarchy. Since the hierarchy defines the cuts that can be visualized, many interesting cuts may be missed if the user is not given a reasonable way to adjust the hierarchy structure during the investigation of their data. We define **hierarchy space** as the space of all possible path-preserving hierarchies given an underlying graph with attributes.

6.2 Interface

The interface for GrouseFlocks is shown in Figure 6.4, with operations to facilitate hierarchy manipulation and exploration. Section 6.2.1 briefly reviews the steerable graph interface previously described in Grouse, Chapter 5. Then, Section 6.2.2 introduces the hierarchy creation and modification

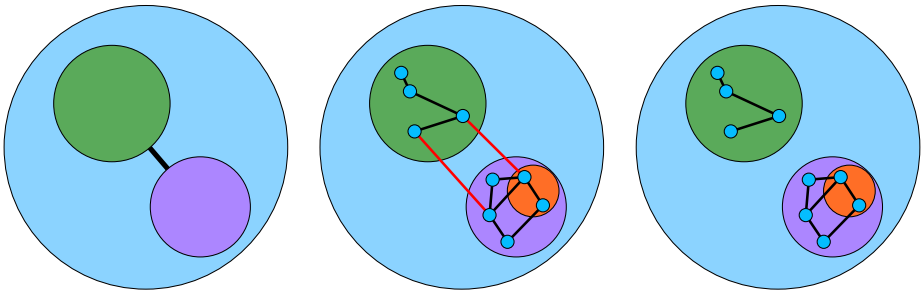


(a) Original Graph



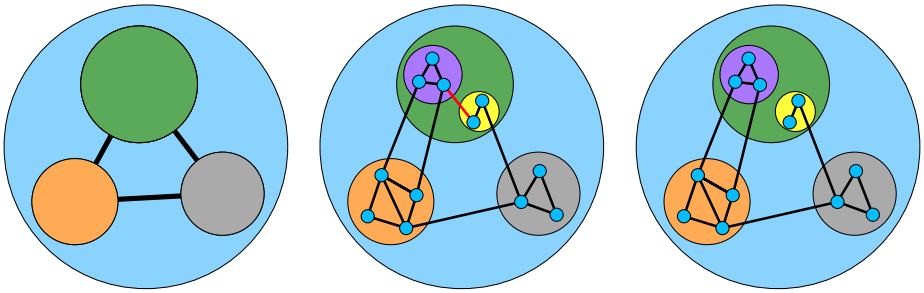
(b) Graph Hierarchies

Figure 6.1: Multiple graph hierarchies superimposed on the same graph. **(a)** The original graph without any hierarchies superimposed on top of it. **(b)** A table of three of the many possible hierarchies that can be superimposed on the graph in **(a)**. The first row of the table shows the three graph hierarchies. The second row of the table shows these graph hierarchies superimposed on the same base graph. As a graph hierarchy defines the types of abstractions that can be visualized by hierarchy cuts, a single graph hierarchy is not suitable for all interesting views of the graph data.



(a) Hierarchy Cut (b) Edge Exists (c) Edge Does Not Exist

Figure 6.2: Edge conservation. In **(a)** a metaedge exists between two metanodes at some level of the hierarchy. A valid input graph is shown in **(b)** where there exist edges connecting descendants of both metanodes. An invalid input graph is shown in **(c)** where edges do not connect descendants of the two metanodes.



(a) Hierarchy Cut (b) Metanode Connected (c) Metanode Not Connected

Figure 6.3: Connectivity conservation. In **(a)**, there is a cycle between three metanodes at some level of the hierarchy. A valid input graph for this hierarchy is shown in **(b)** as there exists a cycle in the underlying graph. An invalid input graph is shown in **(c)** where there is not a cycle in the underlying graph. Thus, subgraphs must be connected for our hierarchies to be path-preserving.

methods. The section describes how the user interacts with the GrouseFlocks interface. Algorithmic details of these operations are presented in Section 6.3.

6.2.1 Hierarchy Navigation

The navigation capabilities of GrouseFlocks, which were previously described in Chapter 5, consist of a tree view and a graph view. These views allow users to modify the hierarchy cut that is used to define the hierarchy editing operations. The **tree view (3)** appears directly below the progress bar on the left hand side of Figure 6.4. It shows all metanodes and leaves in the current graph hierarchy and supports standard interaction of expanding or collapsing items and vertical scrolling. The **graph view (1)** shows the current hierarchy cut with a node-link diagram. The graph hierarchy is superimposed on the hierarchy cut using containment: a metanode contains its subgraph or all its children. The graph view supports pan and zoom through its two-dimensional view.

GrouseFlocks supports linked highlighting between the tree and graph views. Nodes in the graph and tree views are selectable. Metanodes can be opened and closed using either the graph view or the tree view representation of the hierarchy. The labels of the leaves can be set to any combination of attributes by using the checkboxes in the attribute table **(4)**.

6.2.2 Hierarchy Creation and Modification

Users modify hierarchies by carrying out high-level operations that act on the selected set of nodes and metanodes. This section first describes what kinds of selection are supported, and then it describes the two hierarchy modification operators.

Selection

GrouseFlocks, like previous hierarchy editing systems, supports basic manual selection. Users can add or remove metanodes and nodes from the selection set by clicking on them in either the graph view or the tree view. In addition, GrouseFlocks supports selection operations based on regular-expression searching for strings on a chosen node attribute. When users enter a regular expression in the search box **(5)** GrouseFlocks searches all leaves below the cut metanodes in the graph view with respect to the attribute selected in the list box **(6)**. With **pattern match** selection, the nodes are divided into two sets, matched and non-matched. With **category**

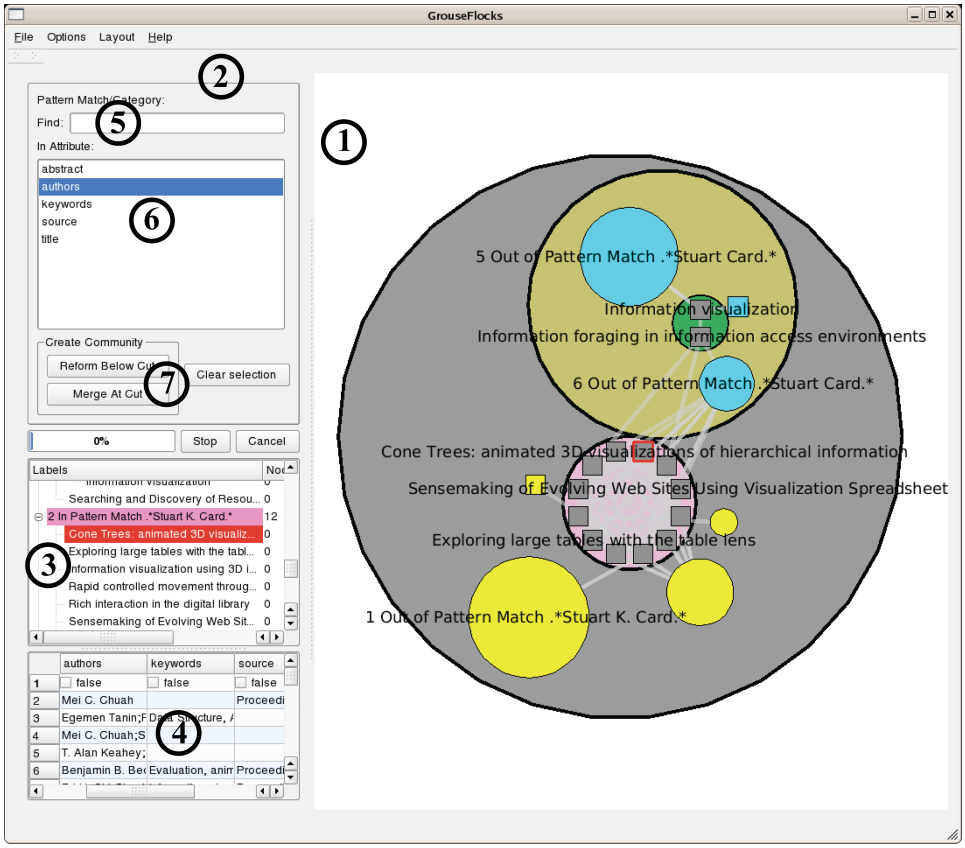


Figure 6.4: The GrouseFlocks user interface. The graph view **(1)** illustrates cuts to the hierarchy. On the left **(2)** are the searching capabilities of GrouseFlocks to select portions of the underlying graph. The tree view **(3)** shows the current graph hierarchy being navigated. The information that appears in the labels can be selected in table **(4)** where checkboxes indicate which attributes are present in the node labels. Regular expression can be entered in the textbox **(5)** to compute selections on the highlighted attribute in **(6)**. The buttons **(7)** provide operations to modify the current hierarchy and clear computed selections.

selection, the nodes are divided into many sets, one for each unique string that occurs in the attribute data.

For instance, a movie dataset might have many attributes, including *genre* and *director*. A pattern match search for the string *action* against the *genre* attribute would separate the nodes into the matched set of action movies, and the non-matched set all other movies. A category search against *genre* would return several sets: *action*, *scifi*, *documentary*, and others. Users can use the standard syntax of parentheses in the regular expression to denote a substring of interest, rather than categorizing against the entire attribute string.

With both forms of selection, matched leaves are highlighted with red label backgrounds in the tree view and red circles in the graph view. If any leaf below a cut metanode matches the search string, that metanode is also highlighted.

Hierarchy Modification

Both of the high-level hierarchy modification operators in GrouseFlocks carry out actions based on the current selection sets and the current hierarchy cut. They are invoked using the buttons **(7)**. The **Reform-Below-Cut** operation destroys the structure of the old hierarchy that is hidden below the hierarchy cut, and creates new metanodes based on the selection sets. The **Merge-At-Cut** operation preserves the hidden structure of the old hierarchy and merges the selected sets within each open metanode. After either of these two operations, GrouseFlocks always produces a new path-preserving hierarchy.

6.3 Algorithms

Section 6.2 describes all the algorithms used to implement selection and metanode creation in the system. The navigation system of GrouseFlocks is the same as that of Grouse discussed in Chapter 5.

6.3.1 Selection

A selection is formed based on the leaves of the graph. Once the user has entered either a pattern match or category expression as described in Section 6.2.2, the algorithm recursively searches the current graph hierarchy from the root downwards until it reaches a leaf. Then, the given leaf for the selected attribute is compared with the regular expression for classification.

Leaf selections are propagated up to the hierarchy cut, so the user is able to determine which metanodes contain selected leaves.

Pattern Match

In the case of a pattern match search, the leaf is either matched or unmatched. The leaf is marked true or false accordingly, and the search continues down to lower levels of the hierarchy. Matched leaves are selected and those selections are propagated to the hierarchy cut.

Category

A category selection divides subgraphs into multiple metanodes, depending on the number of unique strings found. As in pattern match, the hierarchy is recursively searched, and the substring of interest for each node is recorded based on the regular expression. If more than one category is seen below a particular cut metanode, a selection is propagated to the hierarchy cut.

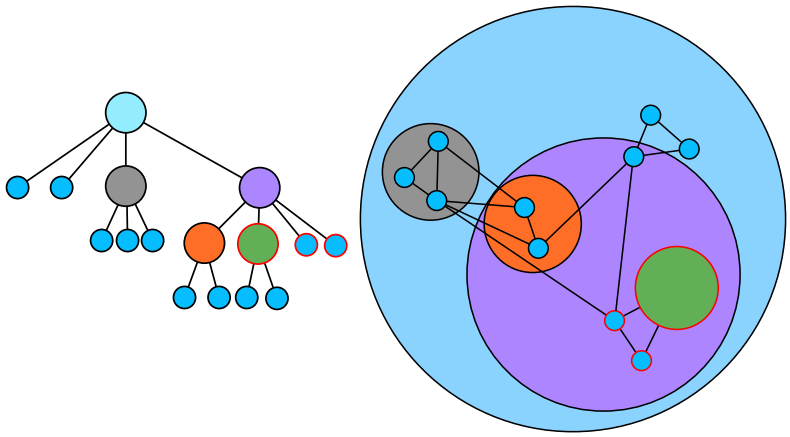
6.3.2 Hierarchy Modification

The hierarchy modification operators described in Section 6.2.2 use two low-level metanode operations as building blocks: merge and delete. These two low-level operations were supported in the previous DA-TU [27] system and in Auber and Jourdan [11].

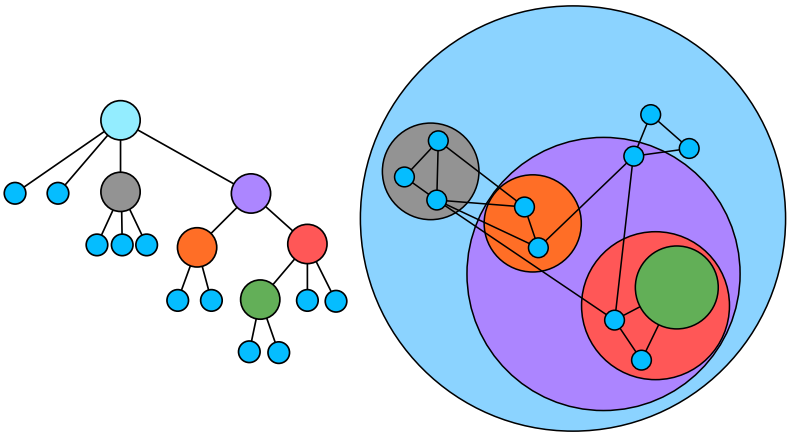
The **Merge** operation, illustrated in Figure 6.5, takes a connected subgraph contained within a single open metanode A and replaces the subgraph with a single metanode at A. Metaedges connect the new metanode to adjacent nodes to the subgraph. For each connected component in the selection, a single metanode is created.

The **Delete** operation, illustrated in Figure 6.6, takes a single metanode and destroys it, so that its children are placed inside its parent. In GrouseFlocks, the delete operation is often called recursively, destroying the entire hierarchy that exists below a metanode.

The next section describes the new high-level operators introduced by GrouseFlocks that use these low-level metanode operations to carry out hierarchy modification using the selection sets, while traversing the hierarchy at or below the current hierarchy cut.

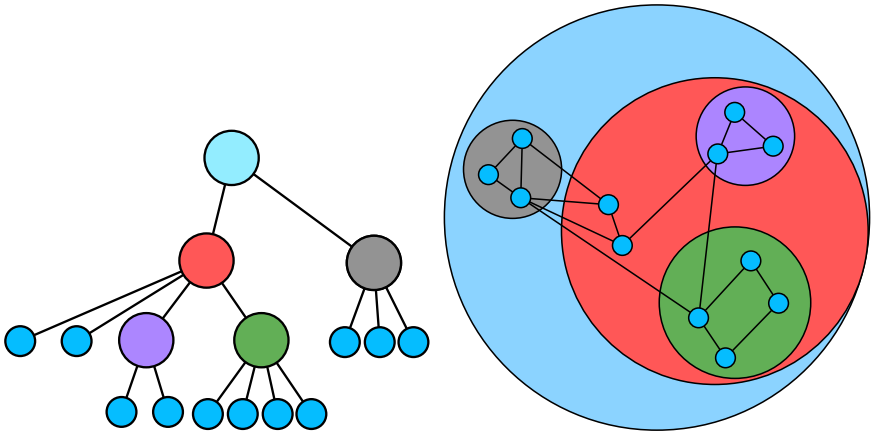


(a) Before Merge

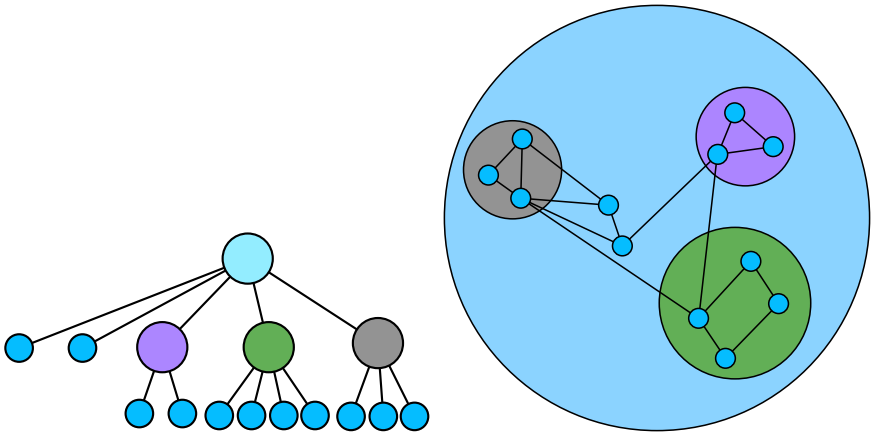


(b) After Merge

Figure 6.5: A merge operation example. Selected nodes in **(a)** are grouped into a single metanode at the current level of the hierarchy as shown in **(b)**. For a merge operation, the selection cannot cross open metanode boundaries.



(a) Before Delete



(b) After Delete

Figure 6.6: A delete operation example. The hierarchy structure below the red metanode in **(a)** is deleted, bringing all metanodes below the red metanode up one level as shown in **(b)**. In GrouseFlocks, the delete operation is only applied recursively to repartition nodes in a graph hierarchy.

Reform-Below-Cut

The Reform-Below-Cut operation consists of a recursive delete from each metanode in the cut all the way down to the leaves of the hierarchy, followed by a merge. Figure 6.7 shows an example.

The Reform-Below-Cut operation considers the selected set and the cut metanodes in the graph hierarchy. For each cut metanode, the operation begins by recursively deleting all hidden metanodes using the metanode Delete operation, as shown in Figure 6.7(b). Once complete, only leaves are present in the subgraphs below cut metanodes. The resultant leaf set is partitioned into metanodes based on the unmatched and matched sets or categories found by the selection. These components are merged with the metanode Merge operation respecting both edge and connectivity conservation. The cut metanodes are opened automatically to reveal the new hierarchy level directly underneath them. It is important to note that leaves may appear on the cut. As leaves do not contain subgraphs, they are not considered in a Reform-Below-Cut operation and are not modified.

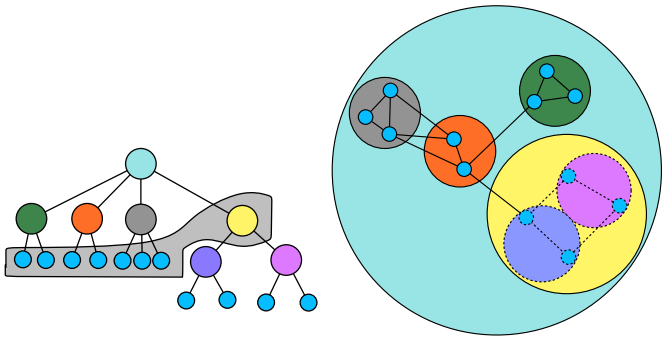
Since Reform-Below-Cut operates on the leaf set below cut metanodes it works well semantically with both pattern match and category operations. In pattern match, it divides the leaf set into matched and unmatched below a given cut metanode. In category, it divides the leaf set into multiple categories dependent on the substring of interest.

Merge-At-Cut

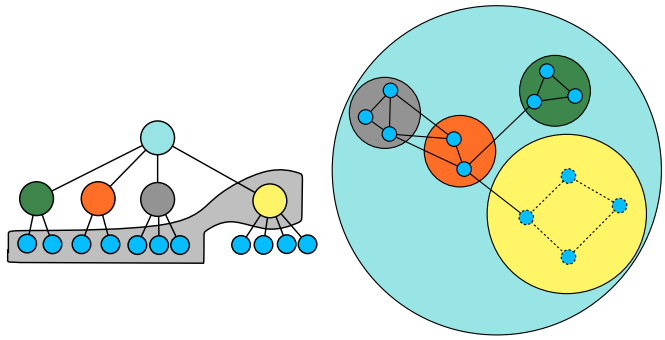
The Merge-At-Cut operation modifies the current hierarchy by merging sets of selected nodes on the hierarchy cut, respecting both metanode boundaries and connectivity conservation. The operation is simply a metanode Merge operation applied to the contents of each open metanode separately.

6.3.3 Coarsening

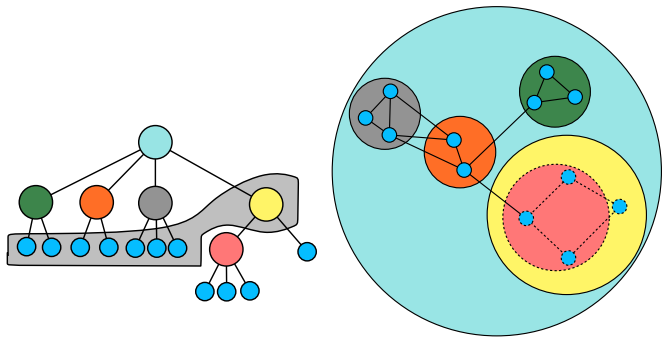
As the user navigates the current hierarchy, the user may open a metanode that is too large to lay out interactively. One way to ensure that exploration remains interactive in these cases is to coarsen the graph below the metanode to a size specified by the user as acceptable for visualization purposes. The algorithm should do this in such a way that large subgraphs contained in the metanode being opened, or major features in the hierarchy, are kept intact, and smaller features are merged together automatically using graph connectivity. The algorithm performs some decomposition into connectivity features, namely tree detection, but our coarsening algorithm is simply not



(a) Reform-Below-Cut Initial State



(b) After Recursive Delete



(c) Reform-Below-Cut Complete

Figure 6.7: Example of the Reform-Below-Cut hierarchy modification operator. Actions taken on a single metanode, the yellow node, shown in **(a)**. All leaves and metanodes below the cut have dashed boundaries. The metanode Delete operator is recursively called until only leaf nodes are present in the yellow metanode, as shown in **(b)**. The algorithm performs a single merge on the leaf set according to the previous selection operation, either match/non-match sets or multiple categories, as shown in **(c)**.

designed to recursively detect connectivity features. Rather, the approach coarsens a subgraph based on the sizes of the subgraphs below metanodes contained within the metanode being opened.

Our coarsening technique, outlined in Figure 6.8, preserves metanodes containing large features and merges smaller features together into metanodes at the next level of the hierarchy cut. The procedure is based on a modification of edge contraction, a method commonly used in graph drawing to produce hierarchies of coarse graphs. In edge contraction, pairs of nodes in the graph are recursively merged based on edge weights or criteria of the directly adjacent nodes. During each recursive pass of edge contraction, any given node can only be involved in a single pass of edge contraction.

Let g be the graph contained inside the metanode that is being opened. Assume the graph g is too large to lay out because the number of nodes exceed some user-specified threshold. The algorithm is divided into two stages: a first stage that performs tree detection and the second stage that performs edge contraction.

The first stage of the algorithm detects trees in g and merges them into single metanodes. The reason for this pass is to abstract away large trees that coarsen very slowly using edge contraction.

The second stage performs a type of edge contraction on g that tends not to coarsen large subgraphs, leaving them at the level directly below the metanode being opened. First, all metanodes and leaves contained in the opened graph are sorted according to the size of the subgraph they contain directly below their metanode. For leaves, this size is set to zero. Next, the list is processed from smallest node to greatest node. For a current unmarked node in the list, the algorithm scans all adjacent nodes that are also unmarked. The algorithm selects the edge with the smallest adjacent unmarked node and flags that edge to be contracted. This pass halts either when the entire node list has been processed, or when enough edges have been flagged for contraction to bring the number of nodes below the threshold. The edges are contracted and the process is repeated until the size of g is below the specified threshold.

If g has $|N|$ nodes and $|E|$ edges, tree detection requires $O(|N| + |E|)$ time. In the edge contraction procedure, the time required to sort the list of nodes is $O(|N| \log |N|)$ and the time to determine the edges to contract is $O(|N| + |E|)$. Thus, if each pass of edge contraction removes half of the nodes of g, the algorithm has a complexity of $O(|N| \log^2 |N| + |E| \log |N|)$.

```
coarsen (Graph g, unsigned int thresh)
  while (g.|N| > thresh)
    detectTrees (g, thresh) {This halts if below threshold}
    g.N ← sortNodesBySubgraphSize (g.N)
    marked ← ∅
    contracted ← ∅
    for all (n ∈ g.N)
      if (g.|N| − |contracted| > thresh)
        break;
      end if
      if (n ∈ marked)
        continue
      end if
      minEdge ← ∅
      minWeight ← ∞
      for all (e ∈ adjacentTo (n))
        opp ← opposite (e, n)
        if (opp ∈ marked))
          continue
        end if
        if (opp.|N| < minWeight)
          minEdge ← e
          minWeight ← opp.|N|
        end if
      end for
      if (e != ∅)
        contract ⋃{e}
        marked ⋃{n}
        marked ⋃{opposite (e, n)}
      end if
    end for
    contractEdges (g, contract)
  end while
```

Figure 6.8: Pseudocode for the coarsening algorithm. The set N represents node sets of metanodes or graphs. As the number of passes is usually sublogarithmic, the complexity of this algorithm is $O(|N| \log^2 |N| + |E| \log |N|)$.

6.4 Comparison of Hierarchies

GrouseFlocks was implemented using the Tulip graph drawing libraries [9] and Grouse [6]. This section compares the interactively created hierarchies of GrouseFlocks to the static connectivity feature hierarchies of Grouse. ASK-GraphView [2] produces similar hierarchies, as it detects the same set of connectivity features as Grouse in the same order. These tools were selected from the previous work because they present the only tools that allow steerable exploration of a static input graph and an associated hierarchy. A 3.0GHz Pentium IV with 3.0GB of memory running SuSE Linux with a 2.6.5-7.151 kernel was used in the empirical evaluation.

The InfoVis 2004 dataset [28] consists of papers and author names along with additional information such as keywords, abstract, title, and location of publication. A subset of this dataset was generated consisting of 103 nodes and 588 edges centered around three key researchers in the field: Jock D. Mackinlay, Stuart K. Card, and Ben Shneiderman. Nodes in this graph represent papers, and there exists an edge between two papers if they share at least one author in common. This graph is called `Coauthor` in the analysis.

The InfoVis 2007 contest dataset [52] consists of movies and associated information about them including: cinematographer, director, female actors, male actors, genre, and Oscars awarded. From this data, two movie graphs were generated. In a movie graph, each node is a movie and there exists an edge between two nodes if the movies shared at least one actor in common. The first dataset, `Movie Small`, consisted of all movies in the contest dataset that had reviewer ratings. The graph contained 9,475 nodes and 140,721 edges. The second dataset consisted of the entire movie database, `Movie Large`. The graph contained 17,192 nodes and 220,321 edges.

All hierarchy modification operations finished in less than one minute. All metanode opening and closing operations on a specific hierarchy took only a few seconds.

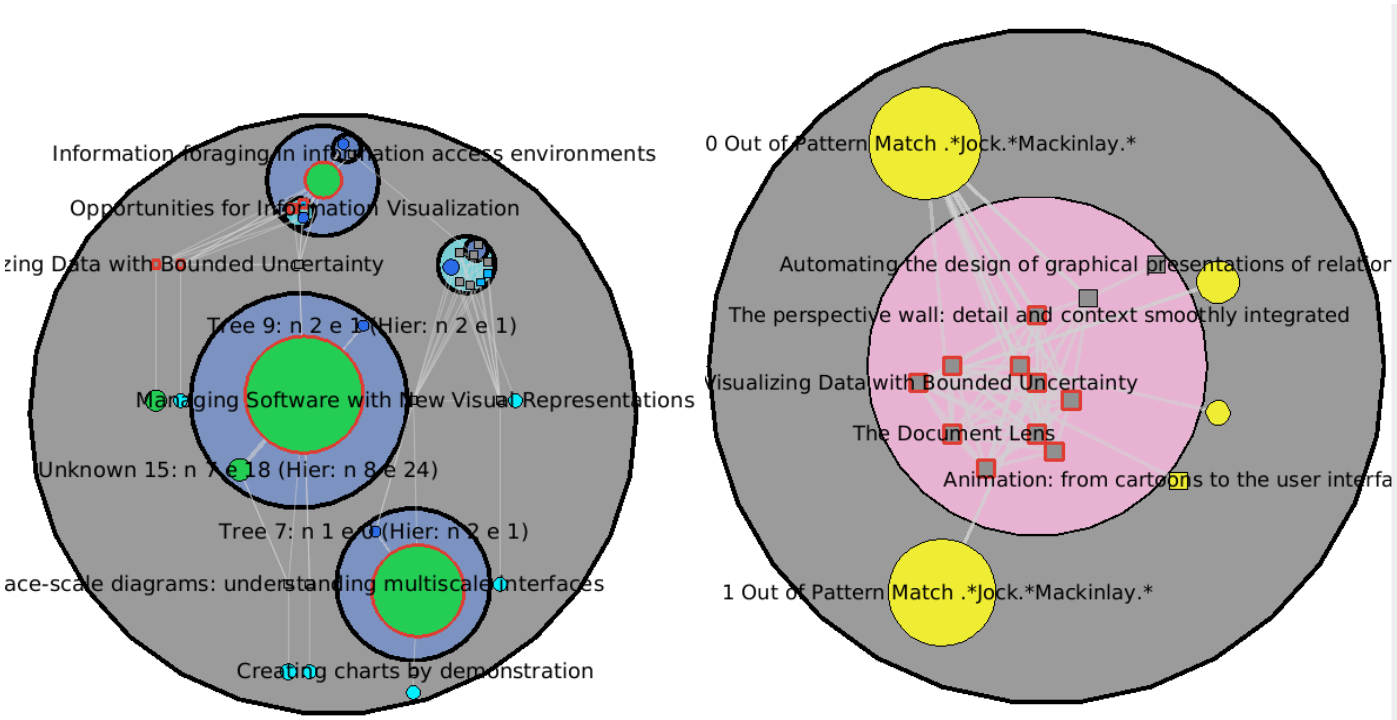
6.4.1 Coauthor

Figures 6.9, 6.10 and 6.11 show the `Coauthor` dataset. Figure 6.9(a) shows a decomposition into connectivity features from Grouse; the results from ASK-GraphView would be similar. After opening a few metanodes, a search for *Jock Mackinlay* is performed on the dataset, and metanodes containing nodes with this attribute are highlighted in red. These nodes are scattered across many metanodes. This result is unsurprising because the author

attribute data was not taken into account during the decomposition into connectivity features. The other attribute searches described below were similarly scattered across many metanodes.

Figures 6.10(a), and 6.10(b) show hierarchies created by GrouseFlocks after searching on the author attribute for *Jock Mackinlay* and *Ben Shneiderman* respectively. In the `Coauthor` dataset, the set of all papers written by an author should be a complete subgraph, with edges between all the nodes representing the fact that there is an author shared across all papers. GrouseFlocks shows complete subgraphs distinctively, using a circular layout. When the dataset was originally explored, a complete graph did not exist for Mackinlay: his name included a middle initial on some papers, and did not include a middle initial on others. Figure 6.9(b) shows a selection for *Jock.*Mackinlay* that captures all Jock Mackinlay's papers. A second search is performed for *Jock D. Mackinlay* and those papers are highlighted in red in the figure. A Merge-At-Cut operation is executed on the selection to group all of these papers into a single metanode. The result is shown in Figure 6.10(a). Figure 6.10(b) shows that the complete subgraph of Ben Shneiderman was found after only one search.

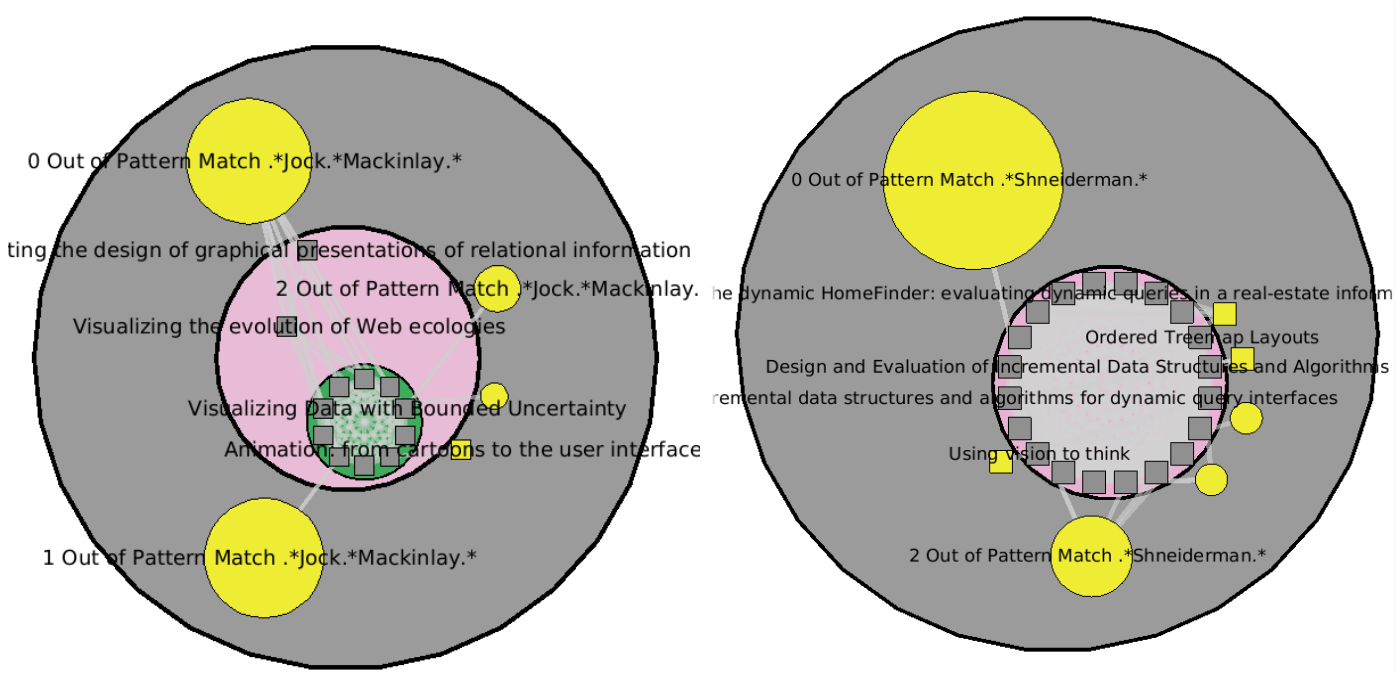
Figure 6.11 shows a different hierarchy generated on top of the same `Coauthor` graph. A Reform-Below-Cut operation is performed after searching for *interface* in the keywords attribute. Matches are pink, and non-matches are yellow. This operation is followed by a second Reform-Below-Cut operation after searching for *Mackinlay* in the authors attribute, with matches in green and non-matches in cyan. The majority of Jock Mackinlay's papers do not contain interface as a keyword, appearing in the open green metanode inside the large yellow metanode in the center. The pink metanode next to it contains the two of his papers that do have that keyword. One of them is *The Document Lens*.



(a) Connectivity Featues

(b) Jock.*Mackinlay

Figure 6.9: Comparison of two hierarchies with respect to the underlying `Coauthor` graph. The hierarchy of connectivity features in **(a)** was created by TopoLayout, and is similar to those produced by ASK-GraphView. The red highlight shows how this hierarchy spreads nodes with the author attribute *Jock D. Mackinlay* over many metanodes. Hierarchy **(b)** shows the result of a Reform-Below-Cut operation based on the selection *Jock.*Mackinlay*. The graph is not complete, which is strange since every paper Jock Mackinlay has coauthored shares Jock Mackinlay as an author. The decomposition is followed by a search for *Jock D. Mackinlay*, showing that some papers were missing the initial in the author name, which are the unselected ones in the figure.



(a) Jock D. Mackinlay

(b) Ben Shneiderman

Figure 6.10: Comparison of four hierarchies with respect to the underlying `Coauthor` graph. In hierarchy **(a)**, a Merge-At-Cut is performed on the selection of *Jock D. Mackinlay* papers shown in Figure 6.9(b). The green metanode contains all *Jock D. Mackinlay* papers while the two leaves in the pink metanode are *Jock Mackinlay* papers. Hierarchy **(b)** shows Ben Shneiderman papers, in pink, separated from the rest of the graph in yellow. Refer to Figure 6.4 for a decomposition of the same dataset for papers coauthored by Stuart K. Card.

6.4.2 Movie Small

Figure 6.13 shows the `Movie Small` dataset. Figure 6.4.2 shows a hierarchy of connectivity features. This hierarchy illustrates the connectivity features of the dataset, but it does not give information about the movies in which an actor appears. This figure shows the results of performing a search for all female actors containing the name Stone. Any cut metanode that contains such an actor is highlighted red. As a decomposition into connectivity features does not consider this attribute information, the data is scattered all over the hierarchy. Exploring a single static hierarchy, such as supported by ASK-GraphView and Grouse, does not provide a useful partition for this task.

By using the GrouseFlocks and the attribute data of this dataset, it is possible to divide the datasets into sets of movies in which an actor has appeared. Figure 6.13(a) shows the dataset divided into sets of female actors with the name *Stone*. Subsequently, second and third Reform-Below-Cut operations are performed to reveal movies containing *Sharon Stone* in the bottom inset of Figure 6.13(b) and *Dee Wallace Stone* in the top inset. By giving users the ability to modify graph hierarchy space, they can refine their investigation from general to more specific structures in the large graph.

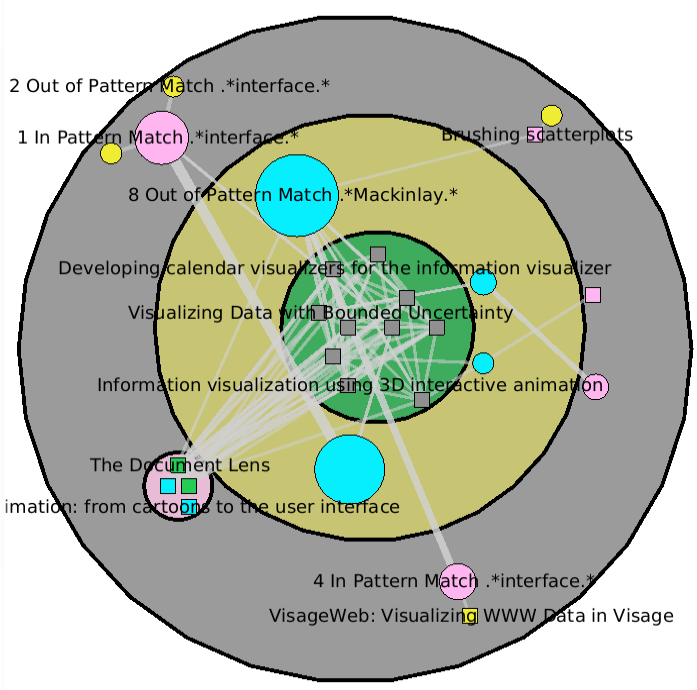


Figure 6.11: A fifth hierarchy on the `Coauthor` graph, decomposing it into two levels. First, papers that contain *interface* in their keyword attribute, in pink, are separated from and those that do not, in yellow. The second level separates the eight such papers with *Mackinlay* as an author, in green, from the two that do not, in cyan.

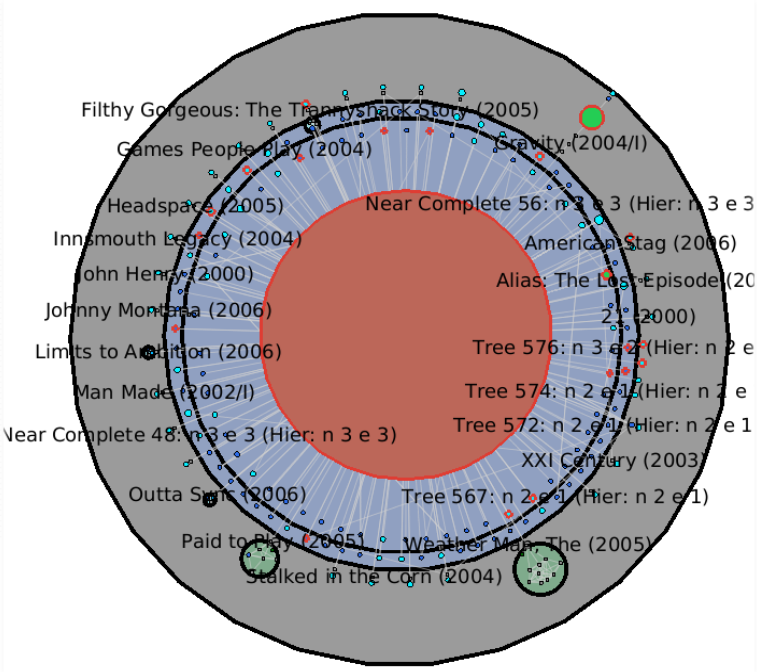
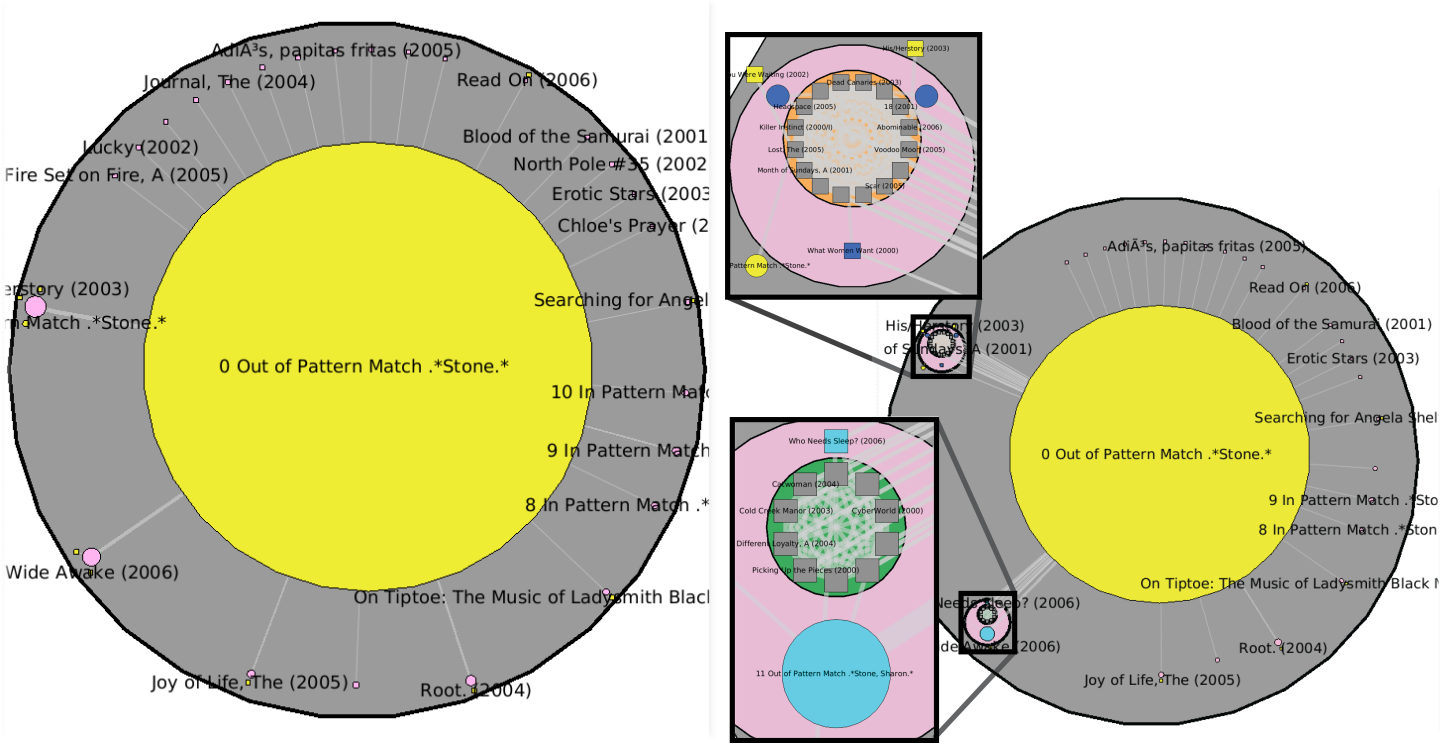


Figure 6.12: A hierarchy of connectivity features for the `Movie Small` dataset. The red selection shows the location of female actors with *Stone* in their name distributed all over the hierarchy.



(a) Stone

(b) Stone + Sharon + Dee Wallace

Figure 6.13: Comparison of multiple hierarchies produced by GrouseFlocks. Hierarchy **(a)** shows movies that contain at least one female actor with the name *Stone* in pink metanodes and movies that do not in yellow. In Hierarchy **(b)**, the movies in which *Sharon Stone* acts are contained in a green open metanode while the movies in which *Dee Wallace Stone* acts are shown in the orange open metanode.

6.4.3 Movie Large

In Figure 6.14 and 6.15, the 17,192 movies and 220,321 edges between them have been decomposed by using connectivity features and by using attribute data. In Figure 6.14(a), a hierarchy of connectivity features like those used in ASK-GraphView and Grouse is explored using the system. The remaining figures depict the exploration of movie genres, directors and actors.

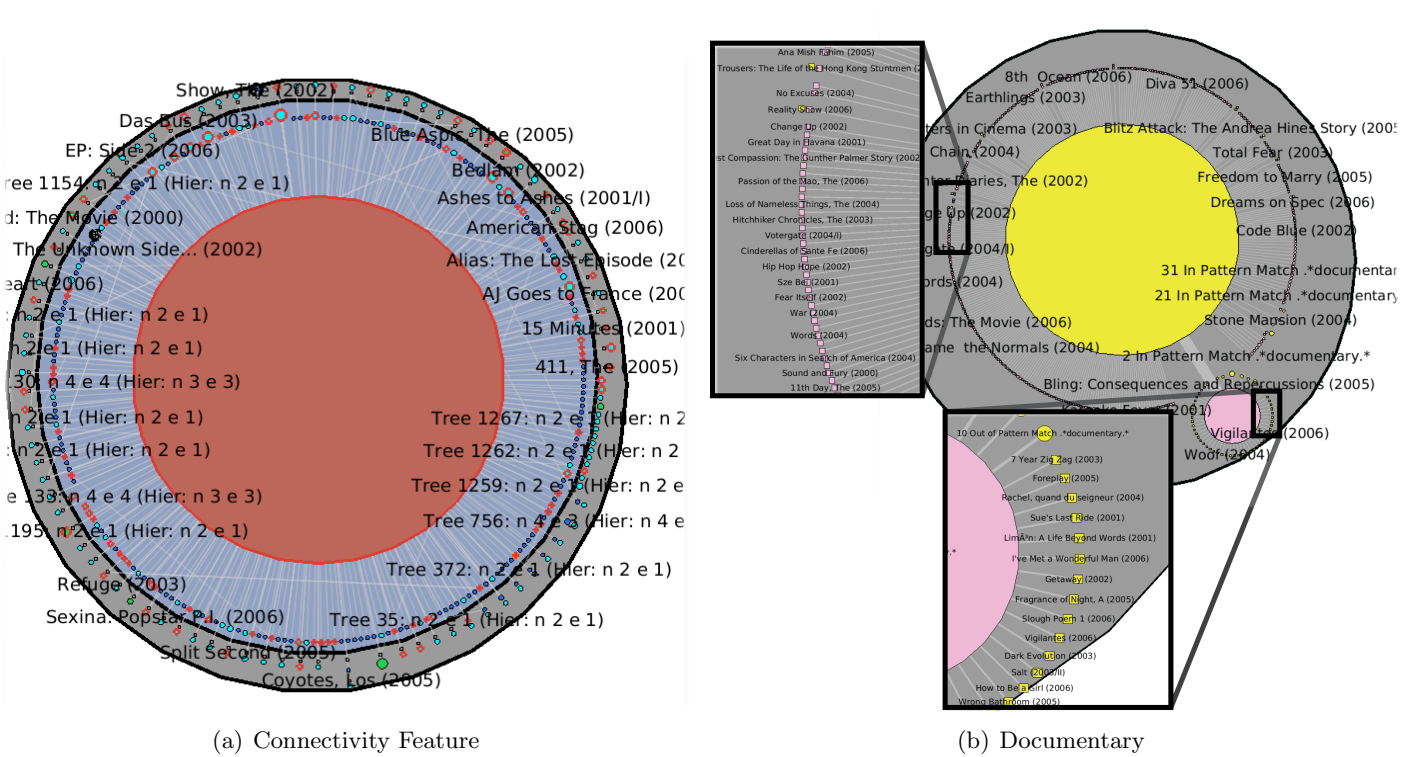
The decomposition of the graph into connectivity features, as presented in Figure 6.14(a), demonstrates much of the clique structure in the dataset. Cliques represent the set of movies for a single actor or a group of actors. Beyond answering questions about the types of movies an actor or a group of actors participates in, it is difficult to get any more information from these drawings. The documentaries in this hierarchy are spread through the hierarchy of connectivity features. Thus, this hierarchy of connectivity features is not suitable for investigating movie genre.

In the remaining diagrams of Figure 6.14 and 6.15, the data is divided into metanodes by genre through a category search on the genre attribute, followed by a Reform-Below-Cut operation. The large components inside the root metanode in these diagrams show an interesting trend: an actor who appears in a single movie of a particular genre is likely to act in multiple movies of that type. In Figure 6.14(b), the pink components are documentary metanodes while the yellow components are metanodes that are not documentaries. As connected components are respected in this graph, a metaedge will only connect a pink node to a yellow node: otherwise the two nodes would be in the same subgraph. In this decomposition, two large subgraphs are illustrated with few nodes being outside these metanodes. This trend persists for action movies as seen in Figure 6.15(a), and science fiction movies as seen in Figure 6.15(b). In the science fiction genre, subgraph structure is more fragmented, and therefore, coarsening was required. The coarsened metanode has been opened and is represented by the brown open metanode.

Figure 6.16 shows a further refinement of the documentary hierarchy of Figure 6.14(b) by performing two more Reform-Below-Cut operations. First, the movies directed by Michael Moore are found by selecting from the *director* attribute, with matches in green and non-matches in cyan. Then, the hierarchy is further refined by separating out movies in which he acted by selecting from the *male actor* attribute. Matches are shown in orange and non-matches in blue. After creating this new hierarchy, the diagram illustrates that Moore has appeared as an actor in several movies, most of which are documentaries, but has only acted in one documentary that he

directed, the orange leaf node *Bowling for Columbine*. The top inset shows documentaries in which Michael Moore acted but did not direct, contained in an orange metanode. In the inset on the left, the three movies in this dataset in which he acted and that are not documentaries are illustrated: *The Private Public*, *Lucky Numbers*, and *The Fever*.

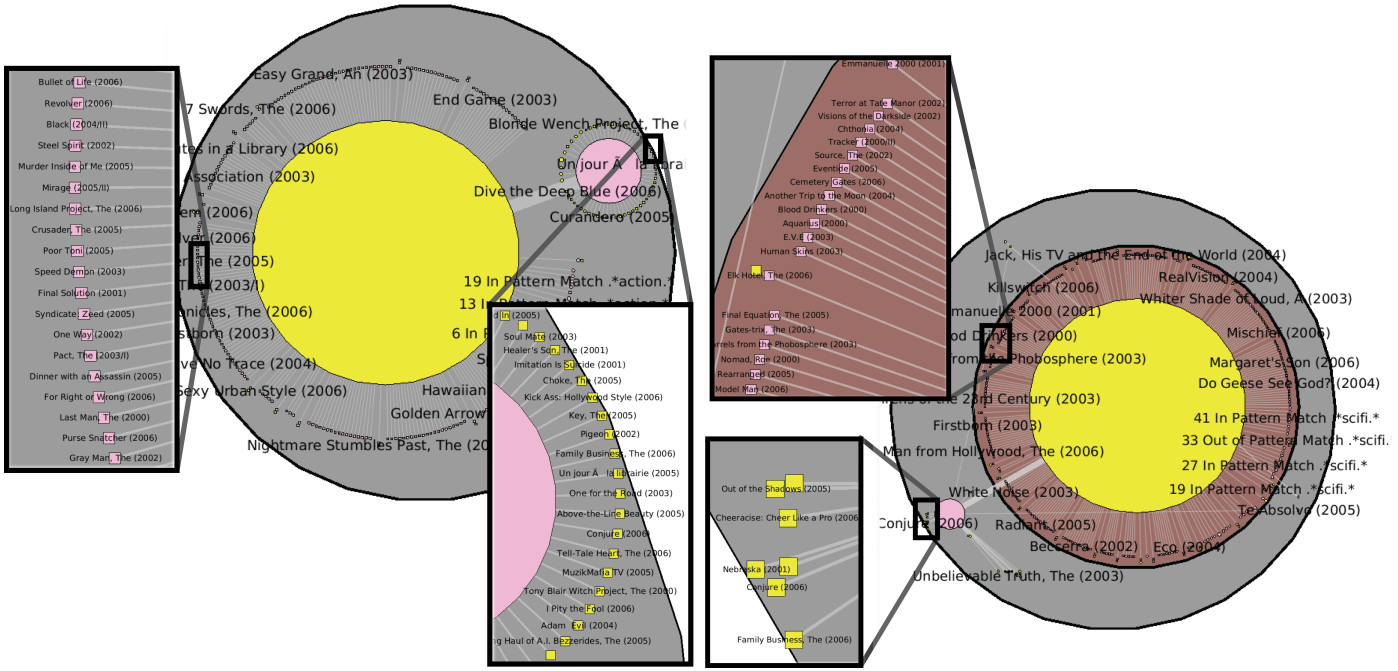
GrouseFlocks was able to create a graph hierarchy that illustrated information about Michael Moore and the movies in which he acts and directs. This information is not visible in the hierarchy of connectivity features as the relevant nodes are scattered across many metanodes, making it difficult to see these relationships in the graph. Steerable creation and modification of multiple hierarchies on top of the same base graph allowed us to iteratively refine hierarchies until a precise one is obtained. Users can use this new hierarchy to develop questions about their data.



(a) Connectivity Feature

(b) Documentary

Figure 6.14: Large version of the InfoVis 2007 contest dataset, where the base graph has a node for each movie and edges between all movies that share actors. In **(a)**, a decomposition into connectivity features is used. The decomposition illustrates some cliques of actors, but does not help the user understand the data with respect to its attributes. In **(b)**, the graph is split into metanodes by the movie genre documentary. By examining Figure 6.15, the data shows a trend across genres: anybody who acts in one movie of a particular genre is likely to act in other movies of the same genre.



(a) Action (b) Science Fiction

Figure 6.15: Large version of the InfoVis 2007 contest dataset, where the base graph has a node for each movie and edges between all movies that share actors. In **(a)** and **(b)**, the graph is split into metanodes by movie genre: action and science fiction. By examining Figure 6.14, the data shows a trend across these genres: anybody who acts in one movie of a particular genre is likely to act in other movies of the same genre.

6.5 Robustness

GrouseFlocks has the same robustness constraints as Grouse for its steerable graph drawing phase because Grouse is used for this phase. We refer the reader to Chapter 5 for a discussion of robustness with respect to drawing.

With respect to hierarchy creation, the addition of a few edges can change the appearance of the high-level structure, as the connectivity feature can also change. As we are exploring the space of hierarchies based on attribute data, a subtle, but important change in high-level structure should perhaps be emphasized via a sharp change in the drawing. However, further user testing is required to validate such a claim.

Subtle changes with respect to the attribute data can also affect the high-level structure of the graph by merging connected components or splitting connected components into two. We do not view this situation as a lack of robustness as these splits and merges provide an accurate reflection of high-level graph connectivity respecting paths in the underlying graph.

6.6 Discussion

GrouseFlocks heavily exploits spacial cues in order to convey the parts of the graph with similar attributes. As spacial proximity is the strongest visual cue, it clearly conveys elements of the graph with the same or similar values for an attribute. The primary disadvantage of this approach over other potential approaches is that the layout of the graph is constantly changing as the user explores cuts of the current hierarchy or the space of hierarchies associated with an input graph. Ways to preserve spacial proximity and minimize motion during operations on the graph and hierarchy would be of benefit to the system.

Initial users working with GrouseFlocks in an informal setting said they had trouble understanding how containment conveys information about the dataset, especially when the nesting of containment is three or more deep. The approach has the advantage of an arbitrary number of composition relationships, but more system support for users is required to help them remember those relationships.

The current coarsening technique attempts to preserve metanodes containing large subgraphs at the current level in order to make them visually salient. This technique is an important first step in helping preserve salient features in the graph hierarchy while preserving some level of interactive performance. Further investigation into coarsening techniques that take into

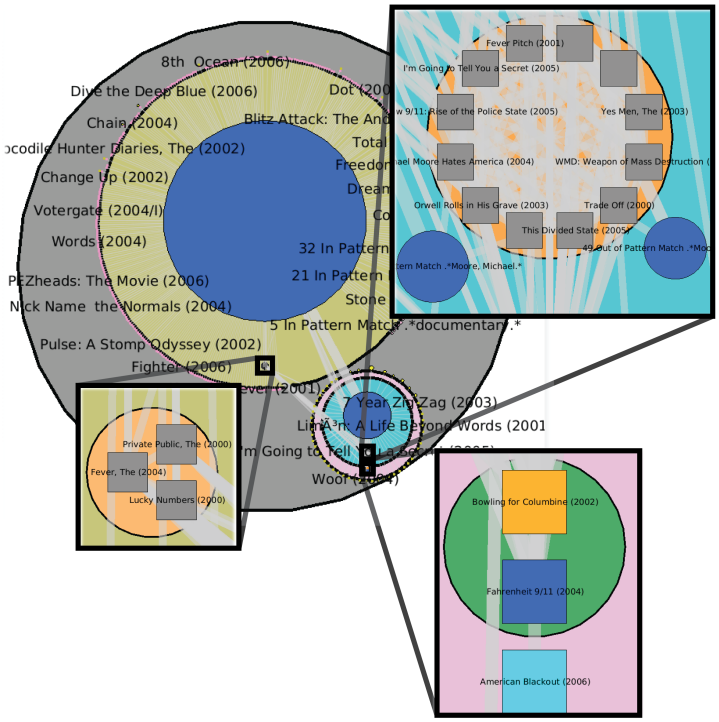


Figure 6.16: Further refinement of the documentary genre hierarchy shown in Figure 6.14(b).

account feature importance in a hierarchy should be investigated.

The primary advantage of GrouseFlocks is that many hierarchies can be investigated within minutes. Previous approaches required the construction of an entire graph hierarchy, an entire layout, or both before investigation of graph hierarchy space on an input graph could begin. Entire initial hierarchy computations may take minutes or hours before the investigation of the data can begin. As neither a graph hierarchy nor a layout is required by GrouseFlocks, the system is more scalable than previous approaches.

Chapter 7

TugGraph: Path-Preserving Hierarchies for Browsing Proximity and Paths in Graphs

During steerable exploration of graph hierarchy space with GrouseFlocks, metanodes can often contain subgraphs of hundreds of thousands of nodes. These metanodes are time-consuming to draw: even the fastest systems, such as SPF, would require thirty minutes to lay them out. The user, however, is frequently interested in the structure located near a particular feature in the graph. Local exploration near a feature in a graph appears to be a natural way to explore the graph. In fact, the technique was originally motivated by a problem posed by a computer networking expert who was interested in understanding the structure of a computer network a few hops away from a particular subnetwork in a larger networking dataset. By seeing a graphical representation of this structure, he would be able to reason about the connections between the subnetwork and the larger networking dataset.

TugGraph is a system that realizes steerable exploration of areas located near a feature. The majority of previous steerable systems [2, 11, 27] as well as GrouseFlocks support hierarchy creation through global or manual selection of features in the graph and must resort to coarsening in order to draw subgraphs consisting of hundreds of thousands of nodes. In contrast, TugGraph teases out areas of the graph adjacent to a feature, depicting how they connect to the larger graph.

TugGraph contributes a technique, and algorithms implementing the technique, for exploring a region of the graph located near a feature. We implemented TugGraph and compared it empirically to previous systems. The work is currently not published.

This chapter is structured as follows. Section 7.1 presents an overview of

TugGraph and its various components. Section 7.2 describes the colouring and node size scheme used in the system. Section 7.3 presents results of the work on a few datasets. A discussion of algorithm robustness is presented in Section 7.4. Finally, Section 7.5 presents a discussion of the TugGraph technique.

7.1 Algorithm

The input to TugGraph is a graph, a source node, and a hierarchy associated with it. The examples in this chapter use a graph hierarchy created by GrouseFlocks set to a specific hierarchy cut. The **source** node is a node of the input graph or a metanode of the hierarchy that is tugged to reveal adjacent input graph components. Once the source is selected, the algorithm operates in five stages:

1. Compute the set of nodes in the input graph, or leaf nodes in the hierarchy, that are descendants of the source. This set of nodes is the **source set** denoted S.

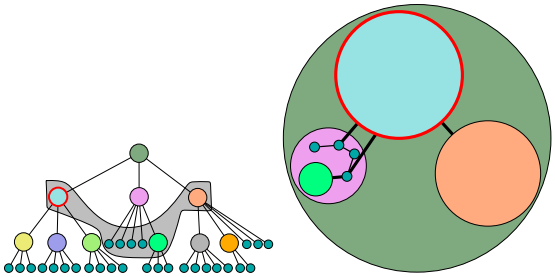
2. Discover the set of leaf nodes of graph-theoretic distance one from the source set that are not elements of the source set themselves. This set is the **proximal set** denoted P.

3. Determine the set of cut metanodes that contain elements of the proximal set. This set is the **proximal cut set** denoted C.

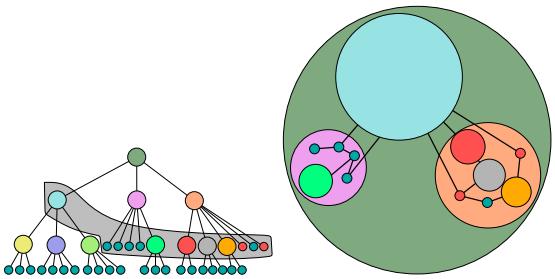
4. For each element n of the proximal cut set, place nodes of the proximal set inside their own metanodes respecting the constraints of a path-preserving hierarchy directly below n.

5. Reconstruct the hierarchy for all other leaf nodes that are descendents of metanodes of the proximal cut set but not elements of the proximal set.

Figure 7.1 shows the result of the five stages on a metanode selected on the graph hierarchy. When describing each step, the complexity of each step is presented after the description. The execution of TugGraph produces a modified graph hierarchy and cut. Elements of the proximal set, all of which were below the cut supplied as input, appear in their containing proximal cut metanodes that are moved above the hierarchy cut. One step produces the set of nodes one hop away from the source set.



(a) Input



(b) Output

Figure 7.1: Result of TugGraph operation. A cut node of the hierarchy is selected, in this case, the highlighted node in light blue as shown in **(a)**. The result of the operation is shown in **(b)**. All the nodes in red consist of leaves exactly one hop away from descendent leaves of the light blue node.

7.1.1 Computing the Source Set

The source set S is the set of nodes of the input graph that are descendants of the selected node on the hierarchy cut. If the selected node is a leaf, S contains one element: the selected node. If S is a metanode, as in Figure 7.2(a), the algorithm traverses the graph hierarchy top down from the selected metanode to discover all leaf descendants as shown in Figure 7.2(b). These leaves are the source set S, outlined in red in Figure 7.2(c).

To compute the source set, the algorithm traverses the hierarchy below the selected metanode and extracts the set of leaf descendants. Let M_S be the set of metanodes below the selected hierarchy. Then, this traversal takes $|M_S| + |S|$ time as each leaf and metanode is scanned exactly once.

7.1.2 Computing the Proximal Set

Once the source set has been computed, the algorithm computes the proximal set. The proximal set is defined as the set of leaf nodes of graph-theoretic distance one from the source set that are not elements of the source set themselves. It is computed on the input graph. More formally, for an edge of the input graph (u, v) and the edges adjacent to node u in the set E_u, this set is denoted P:

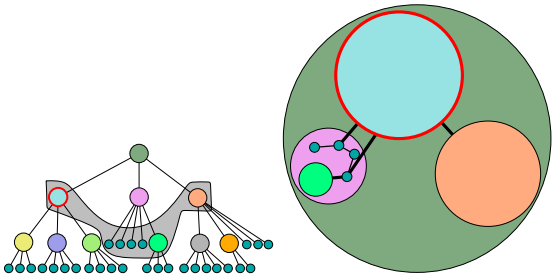
$$P = \{v | (u, v) \in E_u, u \in S, v \notin S\} \tag{7.1}$$

For each element of the source set, the algorithm scans the adjacent leaf nodes in the input graph and determines if it satisfies the criteria in Equation (7.1). The result of this part of the algorithm is shown in Figure 7.3(b).

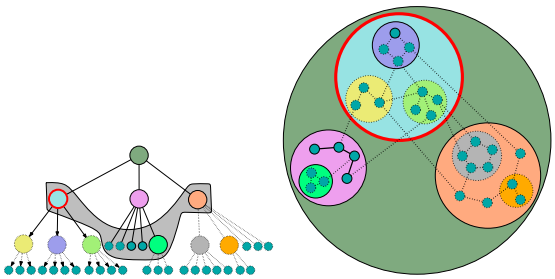
To compute the proximal set, the algorithm scans the set of nodes directly adjacent to all elements of the source set. Leaves that are not elements of the source set are, by definition, elements of the proximal set. Let D_S be the sum of the degrees of the source set. This stage is then $O(D_S)$.

7.1.3 Computing the Proximal Cut Set

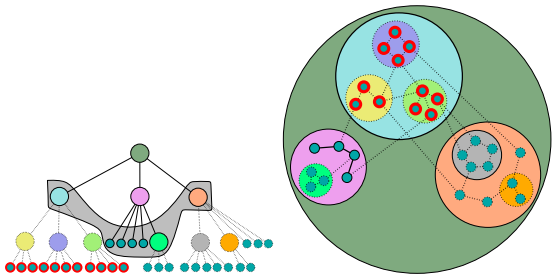
The algorithm derives the proximal cut set C from the proximal set. The proximal cut set is the set of metanodes currently present on the hierarchy cut that contain elements of the proximal set. This set is computed by traversing the graph hierarchy bottom up from each proximal set element up to the first cut metanode ancestor. Figure 7.4(b) shows how the proximal cut set is computed. The proximal cut set links each element of the proximal set to a cut metanode so that they can be placed into components one level



(a) Selected Cut Node

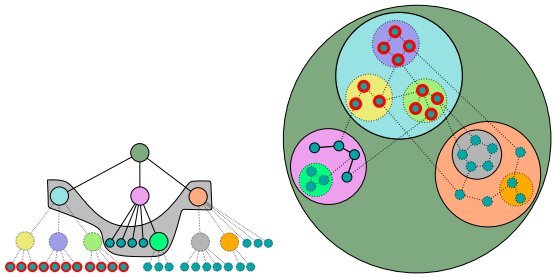


(b) Selected Cut Node Full Hierarchy

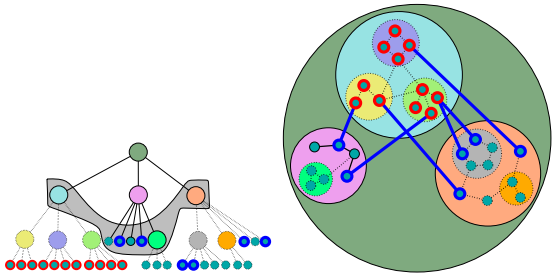


(c) Computed Source Set

Figure 7.2: Computing the source set. Computing the source set starts by considering the cut metanode that is selected as shown in **(a)** and in a full hierarchy view **(b)**. The set of leaf descendants of the cut metanode is computed by traversing the hierarchy top down from the cut metanode. In this case, the result is shown in **(c)**. Dotted parts of the hierarchy in both diagrams are below the hierarchy cut.

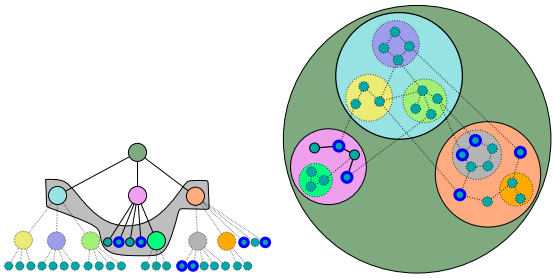


(a) Computed Source Set

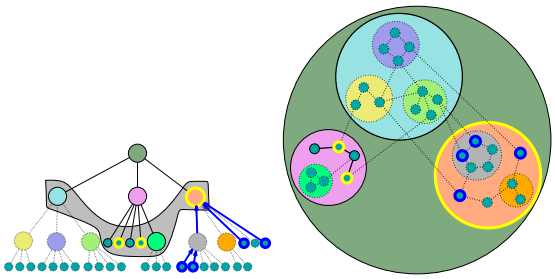


(b) Computed Proximal Set

Figure 7.3: Computing the proximal set. Given the source set S as input, the algorithm scans for adjacent nodes in the input graph that satisfy Equation (7.1). In **(a)**, the source set S is highlighted in red. This set is used to compute the proximal set P highlighted in blue. Dotted parts of the hierarchy in both diagrams are below the hierarchy cut.



(a) Computed Proximal Set



(b) Computed Proximal Cut Set

Figure 7.4: Computing the proximal cut set. Given the proximal set P as input, the algorithm traverses the hierarchy bottom until it reaches a cut metanode. In **(a)**, the proximal set is illustrated in blue. In **(b)**, the proximal set is propagated up to the hierarchy cut and the proximal cut set is shown in green. Dotted parts of the hierarchy in both diagrams are below the hierarchy cut.

below their containing cut metanode. This is why a traversal up to the hierarchy cut is required for each proximal component.

To compute the proximal cut set, the algorithm performs a bottom up traversal of the hierarchy above the proximal set. Whenever the algorithm discovers the cut metanode that is the ancestor of the element of the proximal set, it is stored in a hash table. Therefore each metanode in the hierarchy above elements of the proximal set is visited twice. Let M_P be the metanodes above the proximal set P. Then, this stage is $O(|P| + |M_P|)$.

7.1.4 Computing the Proximal Components

Once the algorithm has determined the proximal cut set and the proximal set, it will proceed to reconstruct the hierarchies below the proximal cut set such that the elements of the proximal set are in metanodes that respect

the rules of a path-preserving hierarchy. These subgraphs are proximal components as every element is an element of the proximal set, meaning they are directly connected by an edge to an element of the source set.

Figure 7.4(b) shows the input to this stage of TugGraph. The proximal set, P, is outlined in blue, while the proximal cut set, C, is outlined in yellow. Before proceeding, a copy of the graph hierarchy is created so that it can be reconstructed below the proximal cut nodes in the last phase of this step.

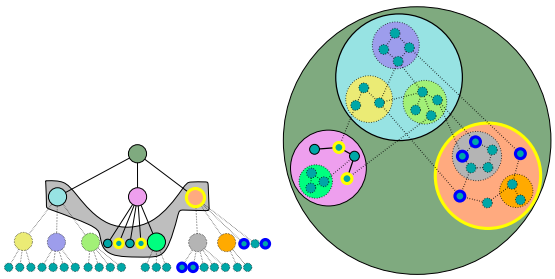
In Figure 7.5(b), the hierarchy below every element of the proximal cut set is destroyed. The resulting hierarchy below any proximal cut node is always a set of leaves. If the cut proximal element is a leaf of the graph hierarchy, it remains unaffected by this step as there is no hierarchy below it to destroy. This step is identical to a recursive delete operation as described in Chapter 6.

The algorithm then computes the proximal components as shown in Figure 7.5(c). These components are the set of induced subgraphs by nodes of the proximal set and each induced subgraph is placed inside its own metanode. An induced subgraph is defined by a set of nodes, in this case the nodes of P, and any edge that links a pair of nodes in P. The result is a set of connected subgraphs. If each connected subgraph is placed in its own metanode, it respects connectivity conservation. If every edge that connects a node in the proximal component to a node not in that proximal component n is replaced by an edge between the metanode and n, it respects edge conservation. Thus, the result is a path-preserving hierarchy as it respects both connectivity and edge conservation.

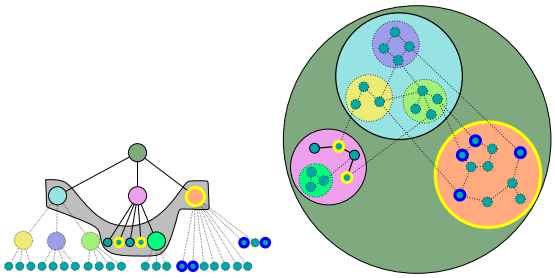
As any fixed fraction of nodes in the proximal set can create a component, at most $|P|$ proximal components are created.

7.1.5 Reconstructing the Hierarchy

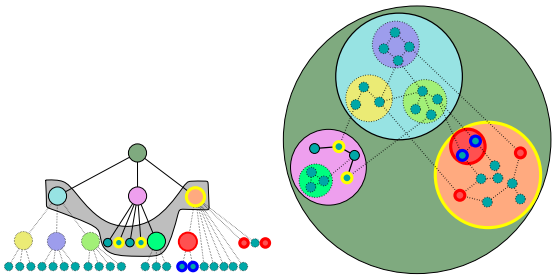
Finally, the algorithm reconstructs the hierarchy that existed previously below the elements of the proximal cut sets, using the backup it had created previously. The hierarchy is constructed bottom up in a way that ensures a path-preserving hierarchy. The removal of a proximal node may disconnect a metanode of the hierarchy by having its edges be the only link between two disjoint subgraphs. As a result, the two newly disconnected components must be placed in separate components in order to respect connectivity conservation. This operation is essentially a recursive application of the Reform-Below-Cut operation of GrouseFlocks, as described in Chapter 6, where the components are divided into sets defined by the previous hierarchy.



(a) Proximal and Cut Proximal Sets

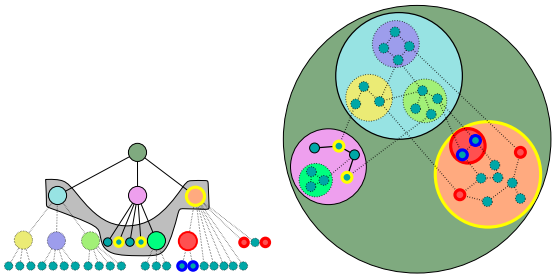


(b) Destroy Below Cut Proximal

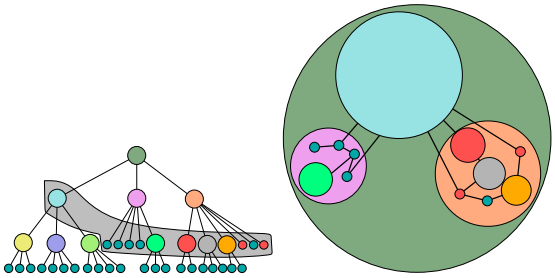


(c) Create Proximal Components

Figure 7.5: Creating proximal components around the proximal set. Three stages are required to rebuild the existing hierarchy around the proximal set. The input to the reconstruction is shown in **(a)** with the cut proximal set, highlighted in green, and the proximal set, highlighted in blue. The hierarchy is backed up and then destroyed in **(b)**. The proximal set is divided into proximal components respecting the cut proximal set and the rules of path-preserving hierarchies in **(c)**.



(a) Input to Hierarchy Reconstruction



(b) Reconstruct Hierarchy

Figure 7.6: The graph hierarchy is rebuilt around the proximal components. The proximal components, existing hierarchy in **(a)**, and backed up hierarchy are provided as input. Finally, the hierarchy is reconstructed around the proximal components in **(b)**.

Once complete, TugGraph has modified the hierarchy so that proximal sets can be investigated below the proximal cut nodes of the hierarchy. The metanodes in the proximal cut set are opened, displaying the results to the user.

The final stage involves reconstructing the hierarchy above the proximal set. Let this hierarchy be the set of M_P metanodes. A proximal node can split a number of metanodes proportional to its degree. If D_P is the sum of the degrees of the proximal nodes, the complexity is $O(|M_P| + |P| + D_P)$.

7.1.6 Worst Case Complexity

Let the graph $G = (N, E)$ consist of two sets: the node set N and the edge set E. Assume the depth of the hierarchy is at most $O(|N|)$ or that we cannot have a metanode contain a single metanode with no edges. In worst case, a tug can take $O(|E|)$ time. This worst case is realized when the sum

Stage	Complexity
Computing Source Set	$O(\|M_S\| + \|S\|)$
Computing Proximal Set	$O(D_S)$
Computing Proximal Cut Set	$O(\|P\| + \|M_P\|)$
Computing the Proximal Components	$O(\|P\|)$
Reconstructing Hierarchy	$O(\|M_P\| + \|P\| + D_P)$

Figure 7.7: Summary of asymptotic complexity of TugGraph stages. The sets S is and P are the source and proximal sets. The sets D_S and D_P are the sum of the degrees of all nodes in the S and P sets respectively. The sets M_S and M_P consist of the sets of metanodes that exist above S and P to the elements of C.

of the degrees, D_S or D_P, is $O(|E|)$, causing proximal set computation or hierarchy construction to be expensive. For deep hierarchies, computing the source set and reconstructing the hierarchy is expensive but $O(|N|)$. For large proximal sets or source sets, computing the respective set dominates, but it is also $O(|N|)$.

7.2 Colouring and Node Sizes

Many TugGraph operations can be executed on an input graph one after the other and in conjunction with Reform Below Cut operations of Grouse-Flocks. In order to distinguish operations from each other, the system rotates through a series of colours: purple, tan, blue, green, and light blue. Proximal components are a more saturated version of the colour while all other components are less saturated. Likewise, matches in a Reform-Below-Cut operation are more saturated compared to nodes that do not match. After a TugGraph operation have been executed, the resultant metanode is opened and appears in the background as a disk, bounding the pull out components.

TugGraph tends to produce many small components and a few large components when operating on a large input graph. The small components are the few nodes adjacent to the node or feature in the hierarchy. The large components are the remaining elements of the graph not adjacent to the node or feature. Due to this disparity in sizes of components the $\sqrt{|N|}$ size estimate used in Grouse and GrouseFlocks prevents a compact drawing. To help solve this problem, TugGraph can present nodes at a logscale node size. When logscale is used, it is explicitly indicated in the text.

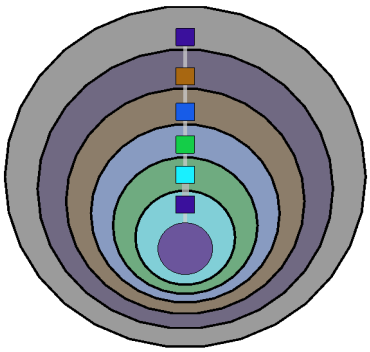
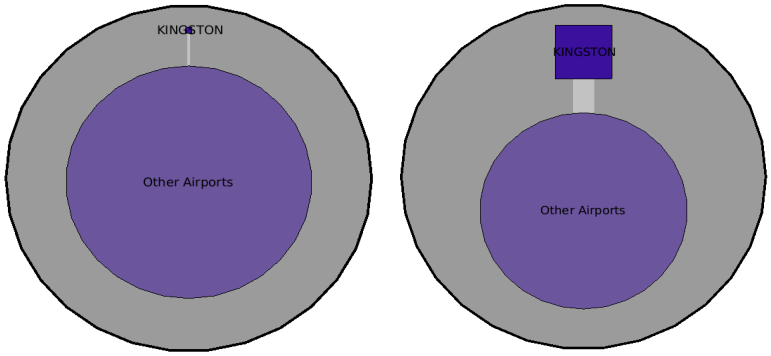


Figure 7.8: Chain illustrating rotating colour scheme one decomposition and seven iterations of TugGraph. The start of the chain, the top purple square, was selected first by Reform-Below-Cut decomposition. All subsequent nodes proximal components were pulled out by executing TugGraph on the preceeding component. Colours cycle through: purple tan, blue, green, and light blue.



(a) $\sqrt{|N|}$) Estimate (b) Log Size

Figure 7.9: Comparing normal sized decompositions to log sized decompositions. The square node is a single leaf node while the circular metanode contains 1,539 nodes. Using the $\sqrt{|N|}$ size estimate used in Grouse and GrouseFlocks, as in **(a)**, makes it difficult to see small components and large components on the same scale. By taking the log of this estimate, as in **(b)**, the two components are visible on the same scale. As TugGraph typically generates many small components and few large ones, log scale node size helps produce a readable drawing.

7.3 Results

TugGraph is implemented using the Tulip graph drawing libraries [9], Grouse, and GrouseFlocks. This section compares TugGraph to other techniques in terms of visual quality and timing numbers on three datasets.

7.3.1 Datasets

The `Airport` dataset is a graph of worldwide airline flights where nodes are airports and there exists an edge between two nodes if there exists a non-stop flight between the two airports. The dataset only has airport name as a node attribute, making attribute-based systems less effective. No physical location information is available. The dataset has 1,540 nodes and 16,523 edges.

The `Net05` dataset [5] shows the structure of the Internet backbone routers as generated in 2005 by Cheswick's Internet Mapping Project[14]. Nodes in this graph are servers and there exists an edge if those servers exchanged packets. The dataset has server name and IP address as attributes for each node. It has 190,384 nodes and 228,354 edges.

The final `Actors` dataset is an IMDB subset centered around Sharon Stone only considering movies in the years 1998 through 2001. In this graph, nodes are actors and there exists an edge between two nodes if those actors acted in a movie together in those years. Actor name is the only attribute on the nodes. The dataset has 38,997 nodes and 1,948,712 edges.

7.3.2 Algorithms Included in Comparison

TugGraph is compared to a variety of graph hierarchy visualization systems, and to highlighting a portion of a layout.

When a hierarchy of connectivity features can be computed recursively in a reasonable amount of time, TugGraph is compared to Grouse. Grouse is a system that recursively detects connectivity features such as trees, biconnected components, creating a graph hierarchy that is used for visualization. ASK-GraphView [2] produces similar hierarchies as its detection algorithms are the same as Grouse and are applied in the same order.

TugGraph is compared to GrouseFlocks on the first two datasets. GrouseFlocks is able to globally detect features in a large graph using attribute data associated with the nodes of the graph and explores the graph using techniques presented in Grouse. It uses coarsening based on connectivity

[14] `www.cheswick.com/ches/map`

feature detection and edge contraction when a metanode is too large to draw in a reasonable amount of time. The TugGraph colour scheme has differences from GrouseFlocks: matches are encoded in saturated colours and non-matches in desaturated versions of the same colour. We made this change based on user feedback. We have also darkened the brown colour used for coarsened metanodes for better contrast.

As LGL [3] and SPF [5] have been shown to work well on datasets like `Net05`, these approaches are included in the comparison with highlighting techniques to show nodes in proximity of the network of interest.

7.3.3 Presentation of Results

For each dataset, a result is presented using TugGraph and the result or part of the result is highlighted in the remaining systems systems. Specifics are explained in Sections 7.3.4, 7.3.5, and 7.3.6 respectively. As TugGraph supports label editing, we manually rename proximal component metanodes created during exploration to have meaningful names. When a TugGraph operation was executed, the metanode that was tugged to generate the image is outlined in red. Proximal components are always presented in saturated colours. We used a 3.0GHz Pentium IV with 3.0GB of memory running SuSE Linux with a 2.6.5-7.151 kernel. This section tests TugGraph on four graphs of varying sizes.

7.3.4 Airport

With `Airport`, we endeavour to discover how the flight paths between Columbus and Vancouver are interconnected. This answer involves some attribute information for the Vancouver, Columbus, and other airports, as well as some adjacency information, mainly the paths between Columbus and Vancouver.

Figures 7.10, 7.11, and 7.12 shows the results for `Airport` under Grouse, GrouseFlocks, and TugGraph. In Grouse, the decomposition into connectivity features neither takes advantage of the attribute information nor the proximity information. Figure 7.10 shows that even finding the airports one hop away is buried very deep in a hierarchy of connectivity features. Figure 7.11(a) demonstrates that GrouseFlocks is better able to solve this problem. The system decomposes the graph into three components initially: Vancouver, Columbus, and other airports. Since there is no attribute information for node proximity, coarsening is used to explore the airports adjacent to both Vancouver and Columbus as shown in Figure 7.11(b). The

solution is better than the one produced by Grouse, but the airports one hop away from Vancouver are still scattered all over the graph hierarchy.

Figure 7.12(a) and 7.12(b) present the results using TugGraph. The process starts with same initial decomposition shown in Figure 7.11(a). First, Vancouver is tugged, extracting the airports one hop away from it. In Figure 7.12(a), the tugged `Vancouver` node is outlined in red, and the dark tan node labeled `Van One Hop` contains all airports one hop from Vancouver. Many small light tan nodes surround it on the periphery: they are things connected to things one hop from Vancouver, thus there are many components two hops from Vancouver. `Other Airports` contains most of the airports two or more hops from Vancouver. Notice that `Vancouver` is only connected to `Van One Hop` and `Columbus` is only connected to `Van One Hop` and `Other Airports`. These connections signify that all paths between Vancouver and Columbus must pass through at least one of `Van One Hop` or `Other Airports`. As metanodes must be connected in path-preserving hierarchies, this indicates the paths between Vancouver and Columbus are interconnected. Figure 7.12(b) shows the results of subsequently tugging on `Van One Hop`. The new blue metanode, `Van Two Hops`, contains airports two hops away from Vancouver because they are adjacent to the set of airports one hop away. The paths are still highly connected as few connections exist to Columbus at the bottom of the figure.

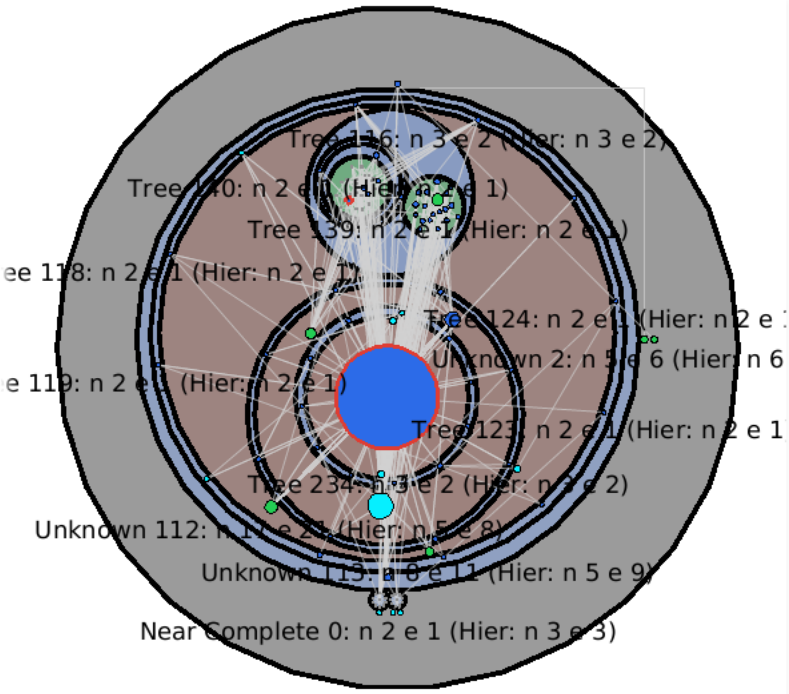
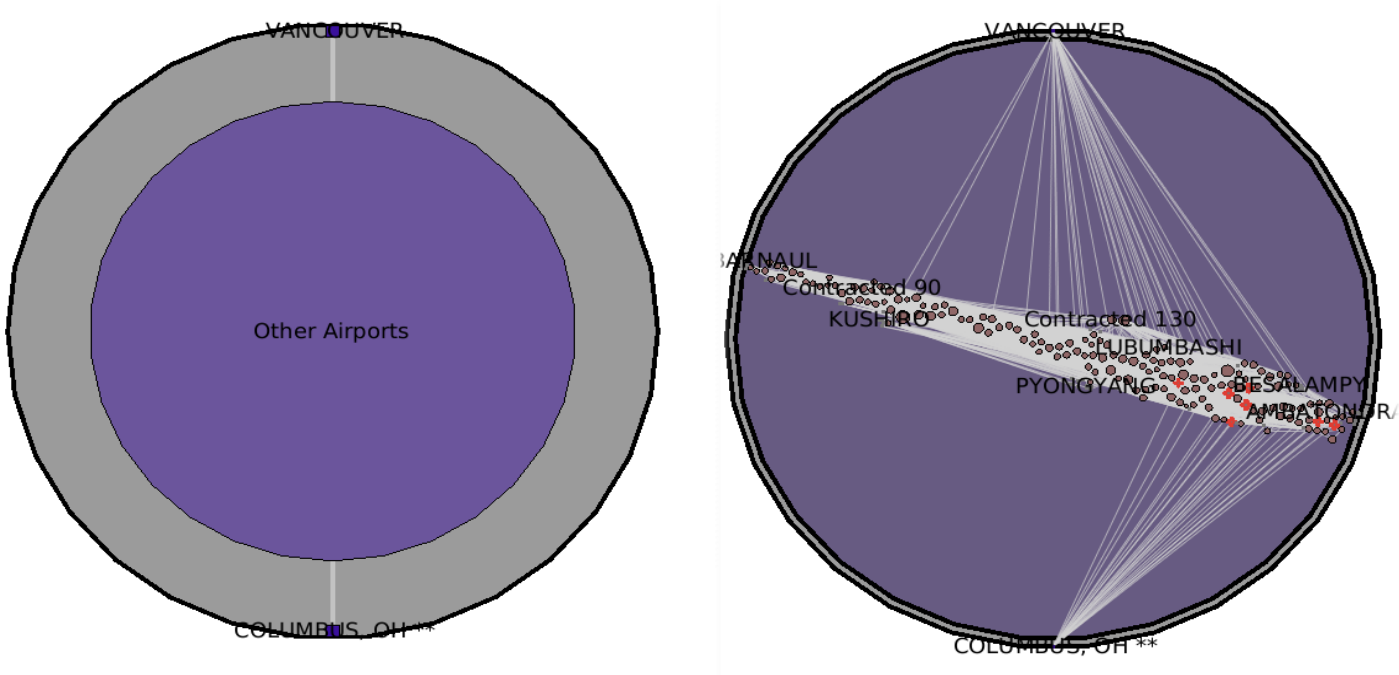


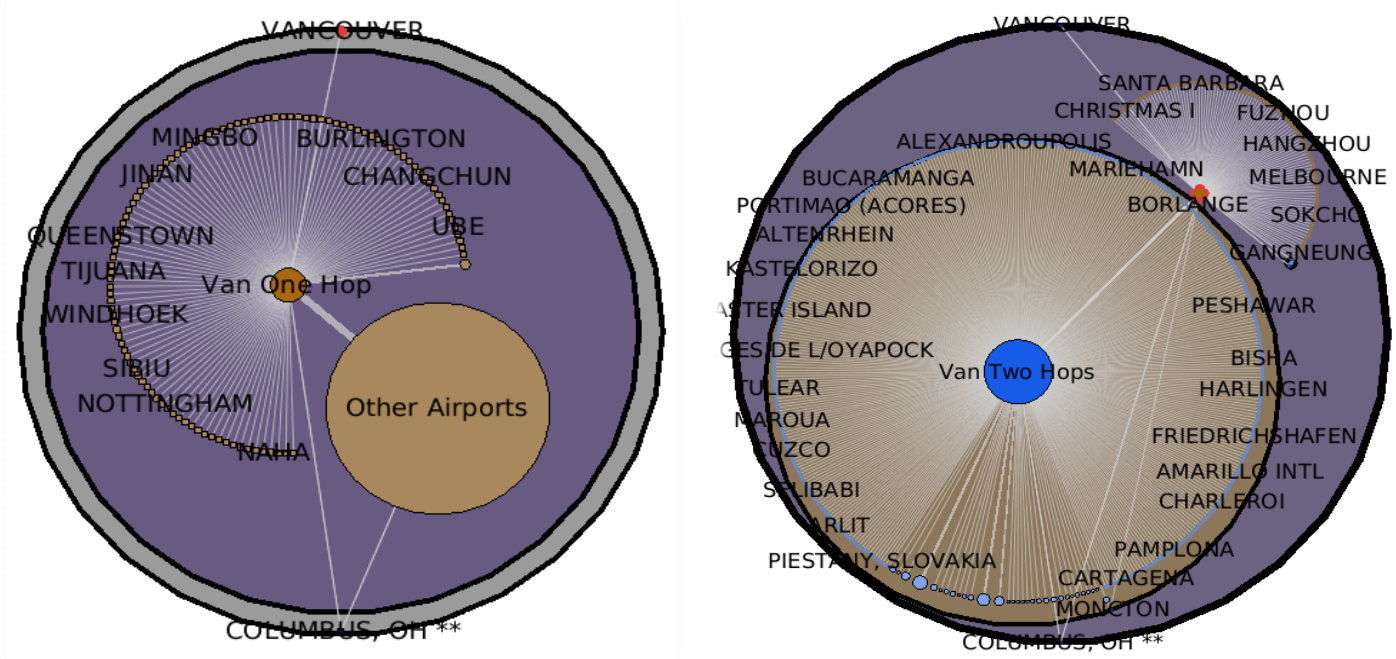
Figure 7.10: Browsing paths between Vancouver and Columbus with results of a Grouse hierarchy of connectivity features.



(a) GrouseFlocks Decomposition

(b) GrouseFlocks Coarsen

Figure 7.11: Browsing paths between Vancouver and Columbus with initial GrouseFlocks decomposition shown in **(a)** and GrouseFlocks coarsening shown in **(b)**.



(a) Tug Vancouver

(b) Tug Van One Hop

Figure 7.12: Browsing paths between Vancouver and Columbus with results of TugGraph shown in **(a)** and **(b)**.

7.3.5 Net05

With `Net05`, we endeavour to discover the structure around the `*.net.ubc.ca` portions of the network. After twelve hours of computation, Grouse had not finished computing a hierarchy of connectivity features, so we do not show it here. Instead, drawings produced by LGL [3] and SPF [5] are compared as these algorithms work well on this type of data. For this dataset, logscale node sizes are used.

The results are shown in Figures 7.13, 7.14, 7.15, 7.16, and 7.17. Once again, the problem is solved using TugGraph and used highlighting to show the solution in other approaches. Figures 7.13 and 7.14 show where the UBC servers are in the dataset and highlight the portions that are four hops away. In these figures, all nodes and edges of the dataset four hops from the `*.net.ubc.ca` servers are highlighted. As the graph has not been simplified, it is difficult to see the path in the context of the entire graph. The paths between the UBC servers that are far away from each other cannot be emphasized without redrawing the data. With GrouseFlocks, shown in Figure 7.15(a), the initial decomposition segments out the UBC servers into two disconnected components with the rest of the Internet in between them. As with the previous dataset, when the huge `Internet` metanode is expanded it is too complex to draw in full and must be coarsened, as shown in Figure 7.15(b). The servers four hops away are all inside the single large metanode outlined in red. The GrouseFlocks solution is more suitable for this task than the previous two drawings, but it is still difficult to browse the connections between UBC and the rest of the Internet.

Figures 7.16(a) through 7.17(c) show how TugGraph can help browse the connections of UBC into the Internet to see if there is a single server that connects UBC to the Internet. Again, TugGraph starts from the initial GrouseFlocks decomposition shown in Figure 7.15(a). First, the `UBC` metanode is tugged, revealing that the two parts of the UBC network are still disconnected and adjacent to the two saturated tan leaves `142.103.204.2` and `ubci9-tx-vantx.bc.net`. However, there is no single connection to `Internet`. `ubci9-tx-vantx.bc.net` is tugged and the result is shown in Figure 7.16(b). `rx0wh-bcnet.vc.bigpipeinc.com` is the greatest common ancestor, joining the two disconnected components of UBC to `Internet` as no other edges connect Internet to the rest of the graph. We continue browsing this connection in Figures 7.16(c), 7.17(a), 7.17(b), and 7.17(c) by tugging on the nodes named in the captions and outlined in red in each of these figures.

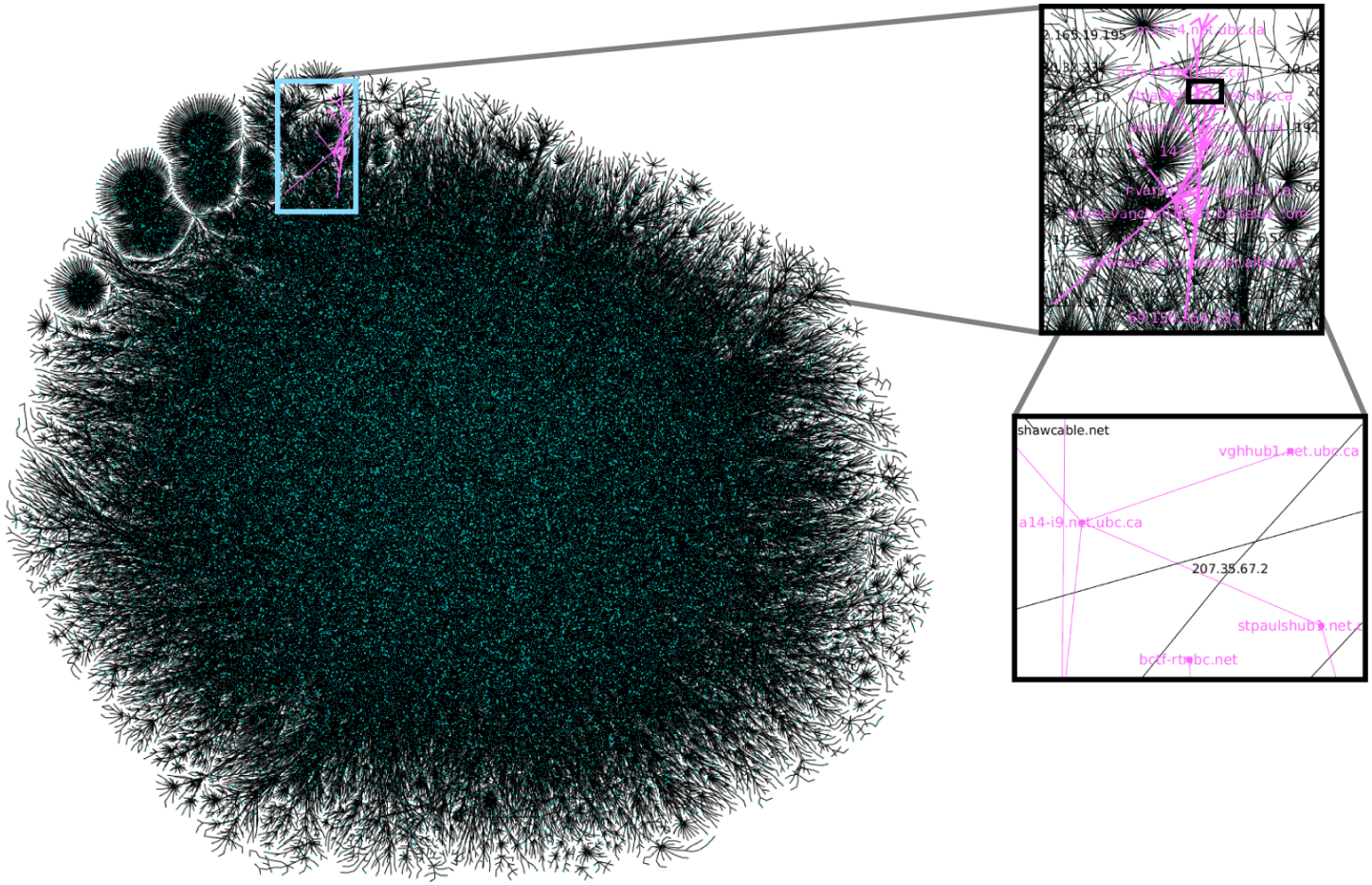


Figure 7.13: Exploration of the `Net05` dataset using LGL. In this drawing, UBC servers and those four hops away are highlighted pink.

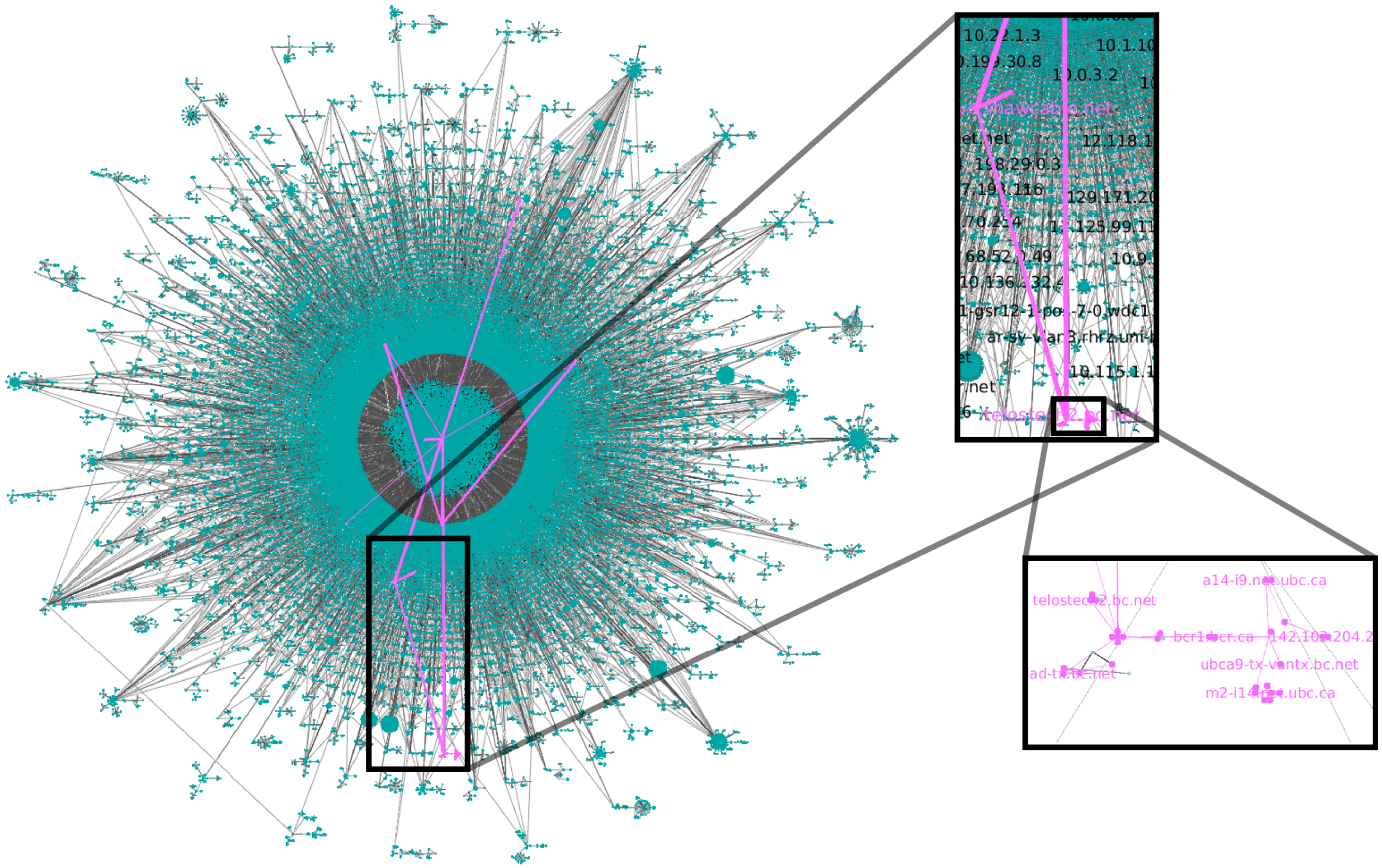
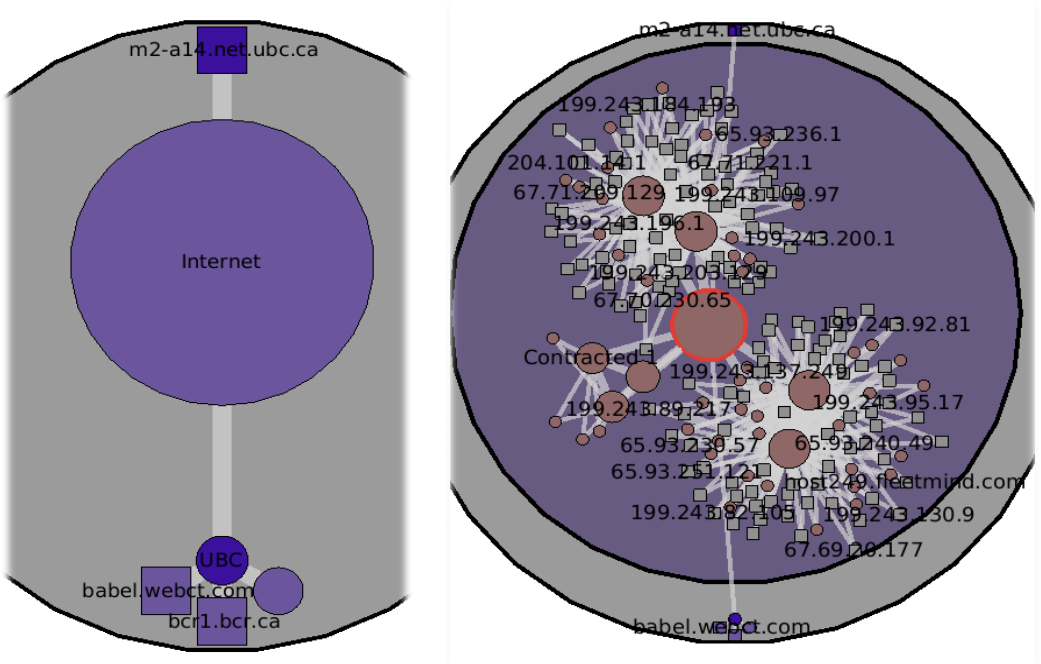


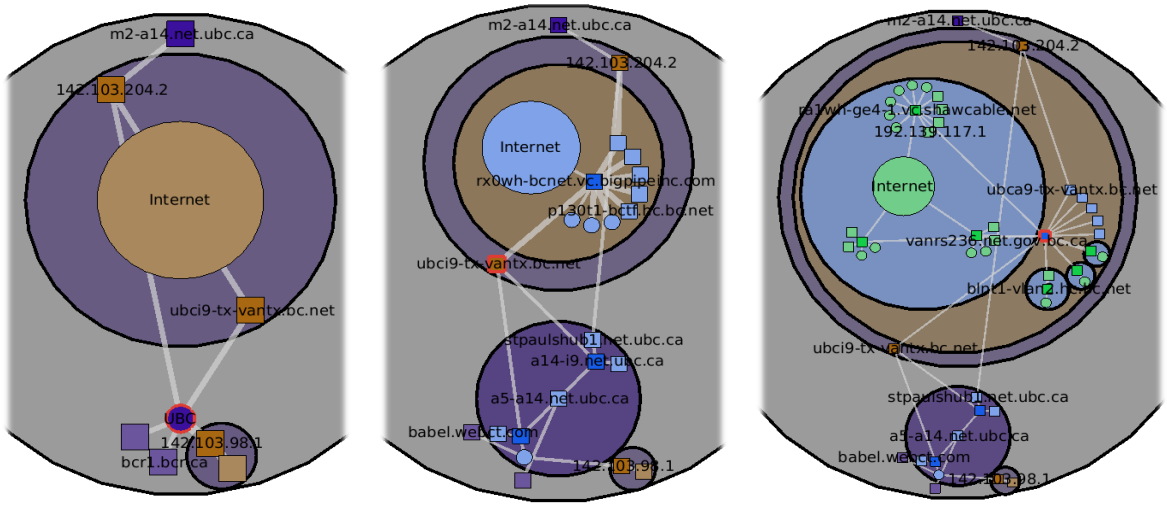
Figure 7.14: Exploration of the `Net05` dataset using SPF. In this drawing, UBC servers and those four hops away are highlighted pink.



(a) GrouseFlocks Decomposition

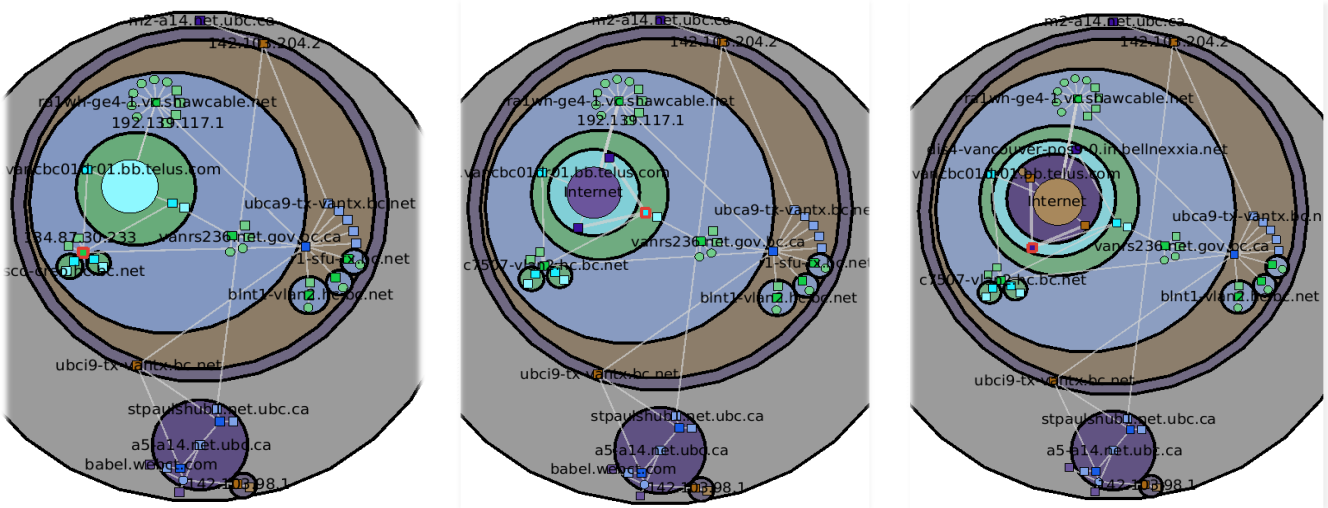
(b) GrouseFlocks Coarsen

Figure 7.15: Exploration of the `Net05` dataset using GrouseFlocks. An initial GrouseFlocks decomposition in **(a)** based on server name. As there is no attribute encoding proximity information, GrouseFlocks resorts to coarsening in **(b)**. GrouseFlocks shows a good initial decomposition but is unable to go further since the attribute information on this dataset is minimal. Coarsened components shown in brown in this figure.



(a) Tug UBC (b) Tug ubci9 (c) Tug rx0wh

Figure 7.16: Exploration of the `Net05` dataset using TugGraph. TugGraph shows how UBC connects to the Internet in **(a)** through **(c)**. Nodes tugged to produce each presented subfigure are outlined in red.



(a) Tug c7507 (b) Tug 69.156.254.254 (c) Tug bellnexia

Figure 7.17: Exploration of the `Net05` dataset using TugGraph. TugGraph shows how UBC extends further into the Internet in **(a)** through **(c)**. Nodes tugged to produce each presented subfigure are outlined in red.

7.3.6 Actors

On `Actors`, we demonstrate that we can generate an overview of Bacon numbers for any movie released between 1998 and 2001. This example is primarily used to illustrate the capabilities of TugGraph and does not necessarily correspond to a real user task. The Bacon number of an actor is zero if the actor is Kevin Bacon and $b + 1$ if the actor has acted in a movie with an actor of Bacon number b. In a graph where nodes are actors and an edge is a movie both actors have acted in, Bacon numbers are paths through this graph and the length of a path to get to an actor from Kevin Bacon determines the Bacon number. If we consider shortest paths, like we do with TugGraph, we are considering the minium Bacon number of the actor. Grouse was unable to generate a hierarchy of connectivity features in over twelve hours of execution time. GrouseFlocks could be used for this exploration, but would produce images very similar to the ones already shown. We thus show only a TugGraph result in Figure 7.18. For this dataset, logscale node sizes are used.

Figure 7.18 shows that two tugs creates an overview of how Bacon numbers are organized in this graph. The diagram shows that all actors with `Bacon Number 1` have acted in a movie with at least one other actor with the same Bacon number by connectivity conservation. Actors with `Bacon Number 2` also have this property as there are only two saturated blue components. However, the trend stops at Bacon number of three as there are many desaturated blue components connected to `Bacon Number 2`. Actors with Bacon numbers of one or two tend to act in movies together as there are few connected components.

7.3.7 Timings

This section presents timing numbers for each of the visualizations above.

On `Airport`, to compute the hierarchy of connectivity features required by Grouse, the decomposition algorithm took 189 seconds. In GrouseFlocks, selection and decomposition into Columbus and Vancouver took about 1.5 seconds. Coarsening in the next step took 0.64 seconds. As the first step of TugGraph is the same as GrouseFlocks, the decomposition and selection was about the same. Each tug took about two seconds to complete.

To draw the `Net05` dataset LGL and SPF took 12 hours and 30 minutes respectively to draw the entire graph. GrouseFlocks took 115 seconds for the initial decomposition and 15 seconds to produce the coarsened graph. After selection and decomposition into UBC and non-UBC servers, TugGraph

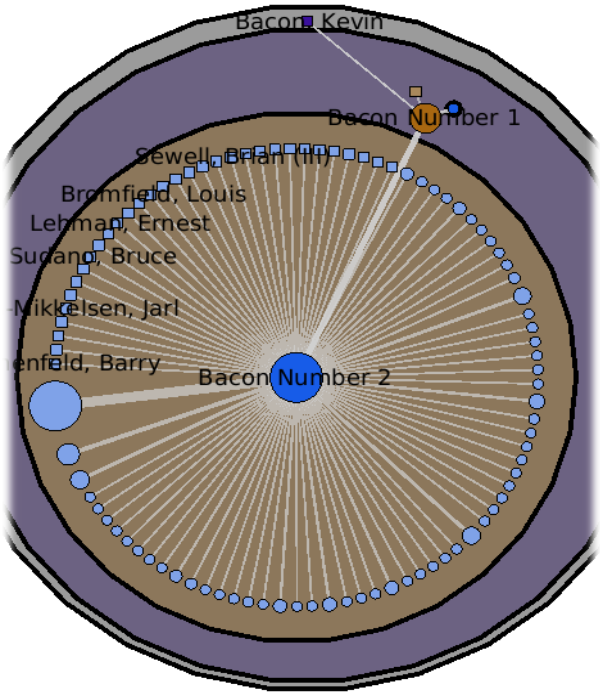


Figure 7.18: Investigating trends in Bacon numbers across `Actors`. `Bacon Number 1` contains all actors with a Bacon number of one. The saturated blue node to its right and `Bacon Number 2` and contain actors with a Bacon number of 2. Actors with Bacon numbers of one or two tend to act in a lot of movies as their components are connected.

took about 20 seconds to tug out each proximal component along the path.

TugGraph took about 110 seconds for the initial decomposition into Kevin Bacon and the rest of the graph in `Actors`. The algorithm took 101 seconds to tug out `Bacon Number 1`, and 409 seconds for `Bacon Number 2`.

7.4 Robustness

In its drawing phase, TugGraph exhibits the same robustness constraints for steerable drawing as Grouse because Grouse is used. We refer the reader to Chapter 5 for a discussion of robustness with respect to drawing.

With respect to hierarchy creation, subtle changes to the connectivity close to to a focus feature should be emphasized because we are explor-

ing the connectivity around a focus feature. Connectivity located far away from a focus feature is not emphasized as it is contained by large metanodes. Therefore, the technique is robust with respect to steerable hierarchy creation.

7.5 Discussion

In the four example datasets, the dependence of tug timings on the sizes of the source and proximal sets along with the sum of their degrees is apparent. There is a general progression in the increase in runtime as dataset size increases. Between `Net05` and `Actors`, the test demonstrates that a sharp increase in degree affects the running time because of the increase in connectivity from two hundred thousand to two million edges.

Another important observation is that the TugGraph result images do not require zoomed-in insets to show details of the graph structure that cannot be seen simultaneously with the overview. Typically, in large graph visualization a wide range of scales are needed to understand features in the data in a global context. Hierarchy-based visualization tools, including TugGraph as presented in this paper, are able to represent the sought structure succinctly at a single scale.

Looking at the overall running times for each of the datasets, we see that TugGraph is suitable for displaying the structure near a small set of nodes in a larger graph. Compared to the time required to compute a hierarchy of connectivity features or computing a full layout using LGL or SPF, the running time of executing TugGraph is significantly smaller in most cases. Also, the diagrams it produces are better suited for exploring connectivity near a feature because less relevant portions of the graph are abstracted away.

Chapter 8

Future Directions

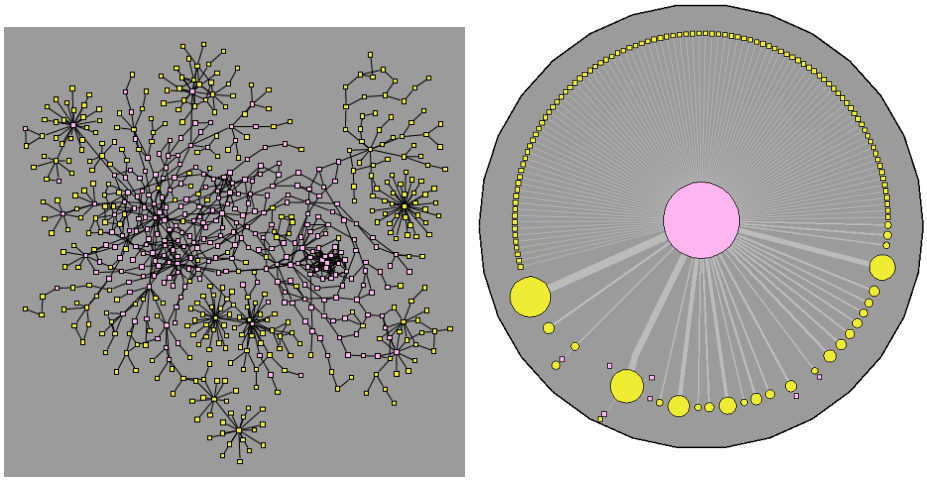
The thesis has extensively explored a feature-based approach to graph drawing. Although we have presented many novel techniques and algorithms, these techniques and algorithms are not complete solutions and many open problems remain. A few of these open problems are discussed in this chapter.

8.1 What application areas are there for feature-based graph drawing?

The feature-based graph drawing techniques and algorithms presented in this thesis are helpful in providing drawings with more clarified high-, mid-, and low-level structure. However, we do not have conclusive evidence that these techniques and algorithms help provide domain experts answer their questions.

Chapter 4 presented SPF. This algorithm made advances in displaying the biconnected component structure of datasets in these domains by breaking edge uniformity constraints. These improvements were measured in terms of speed of algorithm execution and through metrics that counted the number of component overlaps.

In this section, we now explore whether it is possible to further clarify these drawings by using GrouseFlocks as presented in Chapter 6. Figure 8.1 presents an LGL drawing of a small network in Figure 8.1(a). Nodes coloured yellow in this drawing are nodes whose names do not contain `.net` while pink nodes are nodes whose names contain `.net`. Relationships between individual servers are illustrated, but relationships between the components that contain `.net` or do not contain `.net` are more difficult to see. In Figure 8.1(b), a drawing using GrouseFlocks shows the same network using the same colouring scheme. Circular nodes in this diagram are metanodes containing components of the network that are either entirely `.net` or not entirely `.net`. This diagram is an overview of the entire dataset where users can drill down to see relationships. It may be more suitable for investigating how subnetworks are connected. However, further user experimentation is



(a) LGL (b) GrouseFlocks

Figure 8.1: Comparing LGL and GrouseFlocks on a decomposition of a small network dataset. Pink components are all routers whose names end with `.net` in both networks. Yellow components are all other servers on the network. GrouseFlocks is able to show how these .net components connect the other groups of servers in the dataset.

required to determine this conclusively.

This thesis does not present conclusive evidence that such a interactive system will be beneficial for the investigation of large networks. This evidence would be acquired through formal user studies that would help us understand how users investigate large graphs and define more precisely the kinds of features domain experts are interested in seeing in a large graph. These questions could be answered through design studies, formal user experimentation, and insight-based approaches [64].

8.2 How can feature-based graph drawing be applied to weighted graphs?

In many graph drawing problems, the input graph supplied to the graph drawing system has weights. These real values, associated with the nodes or edges of the graph, are often very important to application areas of graph drawing. For example, in computer networking where nodes are servers and edges are the connections between servers, weights on the edges between

nodes could indicate the level of congestion for the link. How could the feature-based approaches discussed in this thesis be applied to handle graphs with weights?

Currently, if a particular weight on the nodes of the graph is sought, GrouseFlocks could be used directly to decompose the graph into connected components of nodes that have a particular weight and nodes that do not. However, neither ranges of weights or weights on edges of the graph are handled. In order to handle ranges of weights, we could extend GrouseFlocks to perform decompositions on ranges of values instead of exact values. In this case, nodes of the graph would be grouped into connected components of nodes that have weights in a certain range and nodes that are not in this range. To handle edge weights via a decomposition approach similar to GrouseFlocks, a few methods for decomposing graphs based on edges exist [47]. However, we have not investigated the applicability of such approaches yet. Further study is required before we could describe such a system.

Another way to present weighted graphs would be to allow these weights to influence the spacial position of the nodes in the final drawing. It may be possible to apply some of the attribute data presented in Section 2.2 to emphasize the weights present on the nodes of the graph. Another approach would be an adaptation of Kamada and Kawai [46] where weights are used to compute the graph theoretic distance between nodes. These weighted distances would produce a drawing more reflective of weights present on the edges of the graph. However, this thesis does not deal directly with weighted graphs. Further study is required to determine appropriate techniques and algorithms for the drawing of weighted graphs.

8.3 How can feature-based graph drawing be applied to graphs with a natural embedding?

Sometimes, graphs come with a natural geometric embedding. A good example of this type of data is the `Airport` data present in Section 7.3.4. In this graph, nodes are airports and edges connect airports if there is a direct flight between them. For this data, it would seem natural to place the airports according to their geographic positions and have a visualization technique that is oriented around this specific embedding.

In this thesis, we do not provide any techniques or algorithms for visualizing this type of data. Therefore, a thorough literature search on techniques for visualizing this type of data was not conducted. One could picture a

system where a hierarchy is constructed based on the euclidian positions of nodes in the given layout. Many of the techniques presented in Section 2.2.3 could be used to construct a hierarchy based on the layout presented to the algorithm. A technique such as EdgeLens [81] could be used to clarify the connections between nodes in areas of the supplied layout where edges have small angles between them. However, it remains an open problem to see how these techniques can be combined into one that can visualize this sort of data.

8.4 Can feature-based graph drawing be applied to directed graphs?

The thesis has focused on undirected graphs. The question remains what sorts of features can be investigated on directed graphs. By convention, directed graphs have an ordering to their drawings. Sources, nodes containing edges primarily directed out from them, are placed at the top of the drawing. Sinks, nodes containing primarily edges directed towards them, are placed near the bottom of the drawing. The work of Sugiyama and Misue [70] provides an excellent starting point for this research, but further investigation into decomposition methods and appropriate drawing algorithms is necessary.

What would be formal definitions for features in directed graph drawing? Often in directed graph drawing, a user may want to understand edges that point up to nodes at previous levels of the drawing. Are there higher level features in directed graphs? Can these connectivity features be combined with attribute data? Currently, it is future work, to see how the techniques and algorithms in this thesis can be extended to directed graph drawing.

8.5 Is there a notion of almost-feature?

A feature-based approach to graph visualization consists of a decomposition phase and a layout phase. In the decomposition phase, a feature is detected or it is not. This hard constraint is similar to those in constraint-based graph drawing: a subgraph either matches a feature or it does not. An almost-feature is a feature that is missing an element of the strict definition of the feature, but shares some high percentage of similarity with the feature. Almost-trees were studied with the SPF system presented in chapter 4, but many open questions remain.

Is it even possible to describe when a subgraph is some percentage similar to an exemplar feature being sought in a large input graph? In what applications is this concept meaningful? Notions such as stretch factor may be adapted in order to detect almost-features, but further research is required to see if almost connectivity features can be defined. Additionally, it is possible that almost-features based on attribute data exist as well, but it remains to be seen how those features can be formalized. A further understanding of almost-features will help increase the robustness of the algorithms and techniques presented in this thesis.

8.6 Can systems be made to help users design features?

A feature-based approach caters to users by helping them find levels of structure in their large input graphs. Defining a feature is pivotal to an effective feature-based graph drawing system. Are there better ways than those presented here to help users define connectivity and attribute features?

One could picture a system where users draw the type of subgraph sought in the input graph. However, such a system would have to solve the subgraph isomorphism problem that is known to be NP-complete. Are there other ways for users to design their own connectivity features? Would such a basic system be effective because the features being sought are small, because the user has drawn them by hand? This system, where users draw the type of subgraph sought, does not consider attribute data associated with the nodes and edges of the graph. What would be a good system for designing features based on attribute data? Also, is there a way to make a system that can define features based on a combination of connectivity and attribute data?

8.7 Can we automatically recognize layouts of poor visual quality?

Some algorithms work very well on certain types of graph data, but not very well on others. A fundamental question of graph drawing has always been: can we automatically recognize a layout of poor visual quality?

Purchase [63] and Ware *et al.* [78] provide some quantitative metrics for assessing visual quality of a graph layout, but in order to compute these metrics, a full layout of the graph in its entirety is required as input. Is it possible to assess visual quality in less time than the time required to draw

the graph? Can we assess it before the layout starts or can we determine visual quality on-the-fly during algorithm execution? Currently, we don't know if it is even possible to assess visual quality during or before layout execution. Perhaps it is possible to create other detection algorithms, similar to the HDE detector in TopoLayout, that can determine if a layout algorithm is capable of producing a good result before drawing begins.

In order to answer this question, we require a better understanding of what visual quality means, and we need to develop further metrics that have efficient algorithms to assess the level of visual quality quantitatively.

8.8 Are there features that are a mix of connectivity and attribute data?

In this thesis, features have been either based on connectivity or attribute data. With the exception of requiring a feature based on attribute data to be connected, there are few examples where graph connectivity and attribute data combine to make a feature. Is it possible to create hybrid features where the definition of the feature uses both connectivity and attribute information?

An example could arise when visualizing large computer networks. Suppose an organization wants to understand how their networks are redundantly connected through the Internet. To answer this question, not all bridge nodes and edges are interesting: only those bridge nodes that are part of the Internet and could connect the networks of the company are of interest. What sorts of systems could be developed to help users investigate these hybrid features?

8.9 Can we relax the path-preserving hierarchy constraint?

Path-preserving hierarchies guarantee that if there exists a path through any hierarchy cut presented to the user, there exists a path in the underlying input graph. If a system were to ignore this constraint, paths present in a hierarchy cut may not actually exist in the underlying graph, misleading the user to think that there is a path in the graph when there is not. Are there types of tasks suitable for these types of hierarchies?

If the designer of a system were to discard connectivity conservation, paths through metanodes in the graph hierarchy are not guaranteed to ex-

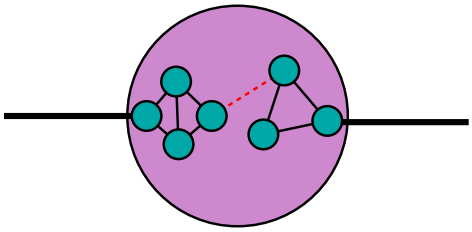


Figure 8.2: If connectivity conservation is broken, paths may no longer exist through metanodes of the graph hierarchy. If the red-dashed edge is removed, a path cannot continue from the left hand metaedge to the right hand metaedge through the purple metanode.

ist. As shown in Figure 8.2, if the red dashed edge is removed then there no longer exists a path through the purple metanode from the left hand metaedge to the right hand metaedge drawn as thick lines in the diagram. If the questions only required direct adjacency information to be answered, a graph hierarchy created in this way is more suitable for these tasks. As all connected components can be grouped into a single metanode, the overview would have less information, making it more likely to be understandable. Questions that would involve only direct adjacency information would include: *What are the one hop connections from my computer network to the Internet?* and *What countries are directly serviced by Germany?*

Relaxing the edge conservation constraint would mean that edges in a cut may or may not be present in the underlying graph. A system with this constraint relaxed could show high level relationships between metanodes that the domain expert knows exists, but are not present in the data. They could also be used to show intermittent connections in time varying data. For example, suppose the user would like to show the links on a large computer network that have gone down. These links would not be present in the acquired network as the servers that have gone down would not transmit packets. A domain expert could potentially add in these links if they are sure that they exist in the graph. Perhaps, in these situations, breaking edge conservation would help users with this task.

8.10 Can feature-based graph drawing be used to improve graph diff?

Graph diff is a problem where a user presents the system with two graphs and asks it to represent the nodes in common or present in only one of the graphs. If nodes have one-to-one correspondences in the two input graphs, the system does not need to solve subgraph isomorphism. Thus, it may be possible to define difference features: features that are present in one graph, in the other graph, or both graphs.

Would such a system be able to help users examine differences between graphs the two graphs? Visualization would be important for this problem in order to formulate questions about why or how a graph is changing over time. For example, in computer networking, such a system could be used in order to understand how a network grows over time. This suggested approach shows promise as the user would be able to abstract away graph differences into metanodes if they were only interested in graph similarities or vice versa.

Chapter 9

Conclusion

This thesis presents five feature-based graph drawing systems that contribute various novel techniques and algorithms to the area. Although the techniques and algorithms presented in this thesis do not provide final solutions to problems in graph drawing, they provide improvements in terms of running time and visual quality over previous approaches. Support for these improvements is provided through empirical evaluations.

This chapter presents the advantages and limitations for each of the techniques discussed in the thesis in Section 9.1 and presents a summary of each chapter and its contributions in Section 9.2.

9.1 Technique Advantages and Limitations

This section discusses the advantages and limitations of each of the approaches presented in this thesis and situations where they might be applied. A summary of all the information in this section is presented in Figure 9.1.

TopoLayout, presented in Chapter 3, is an algorithm for drawing graphs that contain connectivity features. One of its primary advantages is that the approach is unsupervised and the drawing algorithm does not require user interaction. However, the technique has difficulty illustrating mid-level structure. The TopoLayout approach should be applied when the input graph does contain connectivity features and a user would like to see the data in terms of these features. Smaller input graphs would be preferable to larger graphs since the entire graph is depicted in the final drawing.

SPF, presented in Chapter 4, is well suited for the drawing of quasi-trees. One of its primary advantages is that the approach is unsupervised and does not require user interaction. Additionally, it has running time advantages over the TopoLayout approach for graphs that have many biconnected components. However, the entire graph must be drawn and displayed at once. This approach can be applied when the user needs to see all of the data and the graph has tree-like structure.

Grouse, presented in Chapter 5, is well suited for exploring a fixed graph hierarchy on top of a fixed graph. The approach scales to larger graph

datasets because it is able to reduce node and edge occlusion over the previous cited approaches. The reduction in the amount of occlusion is supported through a qualitative assessment of visual quality of the images. The hierarchy is generated automatically. This automatic hierarchy generation is an advantage when the user is not interested in customizing the hierarchy according to attribute information, or no attribute information is available. However, the entire graph is not shown and only connectivity features are considered when creating the hierarchy. This approach is recommended when no attribute information associated with the nodes of the graph is available.

GrouseFlocks, presented in Chapter 6, allows users to customize hierarchies based on attributes associated with the nodes of an input graph. The primary advantage of this approach is that the system allows users to generate graph hierarchies based on attribute data. Additionally, automatic coarsening ensures interactive performance while preserving large features in the hierarchy. However, manual creation of a graph hierarchy can be difficult as there are exponential number of such hierarchies. If attribute information is impoverished or not available, this technique is probably not applicable because hierarchies cannot be generated easily. GrouseFlocks is suited for large graphs with many attributes associated with the nodes of the graph.

TugGraph, presented in Chapter 7, allows users to explore a pre-generated hierarchy using proximity information. Its primary advantage is that it provides adjacency-driven hierarchy creation capabilities that are useful when attribute information is scarce. However, the technique requires a graph hierarchy as input and does not consider attribute information. This technique is suggested when a good, shallow hierarchy is available and the data is attribute impoverished, or the structure near a feature needs to be explored.

Technique	D	L	Advantages	Limitations
TopoLayout, Chapter 3	O	O	- Fast, general graph drawing technique - Unsupervised hierarchy generation and drawing - Depict connectivity features well	- Entire graph must be drawn - Connectivity features must be present - Mid-level structure remains hidden
SPF, Chapter 4	O	O	- High- and low- level structure improvement - Fast on large quasi-trees	- Only works well on quasi-trees - Entire graph must be drawn
Grouse, Chapter 5	I	S	- Cuts reduce node and edge occlusion - Graph drawn incrementally, on demand	- Parts of graph hidden - Hierarchy required as input
GrouseFlocks, Chapter 6	S	S	- Nodes with same attribute value kept together - Interactive performance with coarsening - Immediate visualization of attribute data	- Manual creation of many possible hierarchies - Deep or coarsened hierarchies can be confusing - Rich attribute information required
TugGraph, Chapter 7	S	S	- Adjacency driven hierarchy creation - Works with attribute-poor data - Structure visible at a single scale	- Created hierarchy required as input - Only local proximity information considered

Figure 9.1: In this table, the advantages and limitations of TopoLayout, SPF, Grouse, GrouseFlocks, and TugGraph are presented. Discussion of these advantages and limitations is present in Section 9.1. In this table, the **D** column is the decomposition phase of the technique and the **L** column is the layout phase of the technique. A value of O in these columns indicates offline computation, a value of I indicates required as input, and a value of S indicates steerable computation.

9.2 Chapter Summaries and Contributions

This section summarizes each chapter and restates the contributions of the work presented as presented in the Thesis Overview of Section 1.5, providing a conclusion to the thesis.

Chapter 3 presents TopoLayout. TopoLayout is a multilevel, feature-based graph drawing algorithm that recursively detects the connectivity features described in the introduction, forming a graph hierarchy. Neither its decomposition nor its layout phase are steerable.

The contribution of TopoLayout is a multilevel graph drawing algorithm based on connectivity features. It also introduces a pass for reducing edge crossings in the final drawing.

Chapter 4 presents a second feature-based graph drawing algorithm, Smashing Peacocks Further or SPF. SPF is a feature-based graph drawing algorithm that is based on TopoLayout. The algorithm is tuned for graphs with tree-like structure. Neither its decomposition nor its layout phase are steerable.

The contribution of SPF is an algorithm that tunes the TopoLayout pipeline for drawing quasi-trees. Additionally, the approach extends the previous RINGS algorithm in order to make it area-aware. In the empirical evaluation of SPF, we extended some traditional graph drawing metrics to a multilevel graph drawing context in order to evaluate the high-level structure in the drawings.

Chapter 5 presents Grouse, a steerable graph hierarchy exploration system. Grouse realizes an algorithm to explore a previously constructed hierarchy of connectivity features. Grouse takes this previously constructed graph hierarchy as input, but its drawing phase is steerable.

The contribution of Grouse is an algorithm for the steerable exploration of a graph and an associated graph hierarchy of connectivity features. Additionally, a re-layout algorithm is presented that keeps nodes in the final drawing closer to their input size.

Chapter 6 presents GrouseFlocks, a system for the exploration of the space of graph hierarchies that can be constructed on top of a graph using attributes associated with the nodes of the graph. In this system, both the layout and decomposition phases are steerable.

The contribution of GrouseFlocks is a path-preserving technique for the steerable creation and exploration of graph hierarchy space based on attribute data associated with the nodes of the graph. The approach also introduces a coarsening algorithm that preserves large metanodes in the created hierarchy.

Chapter 7 presents TugGraph, a system that assists users in exploring areas of a graph adjacent to a feature. Using an initial decomposition, the systems allows users to interactively modify the graph hierarchy based on information directly adjacent to nodes and metanodes. In this system, both layout and decomposition phases are steerable.

TugGraph contributes a technique, and algorithms implementing the technique, for exploring a region of the graph located near a feature.

In this thesis, we extensively explore the area of feature-based graph visualization by developing and evaluating novel techniques and algorithms. We provide support for our claims of improved visual quality and running times of these systems through empirical evaluations against previous approaches.

Bibliography

[1] J. Abello, S. G. Kobourov, and R. Yusufov. Visualizing large graphs with compound-fisheye views and treemaps. In *Proc. Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 431–441. Springer-Verlag, 2004.

[2] J. Abello, F. van Ham, and N. Krishnan. ASK-GraphView: A large scale graph visualization system. *IEEE Trans. on Visualization and Computer Graphics (Proc. Vis/InfoVis '06)*, 12(5):669–676, 2006.

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

[4] 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.

[5] D. Archambault, T. Munzner, and D. Auber. Smashing peacocks further: Drawing quasi-trees from biconnected components. *IEEE Trans. on Visualization and Computer Graphics (Proc. Vis/InfoVis 2006)*, 12(5):813–820, Sept.-Oct. 2006.

[6] D. Archambault, T. Munzner, and D. Auber. Grouse: Feature-based, steerable graph hierarchy exploration. In *Proceedings of Eurographics / IEEE VGTC Symposium on Visualization (EuroVis 2007)*, pages 67–74, May 2007.

[7] D. Archambault, T. Munzner, and D. Auber. TopoLayout: Multilevel graph layout by topological features. *IEEE Trans. on Visualization and Computer Graphics*, 13(2):305–317, March/April 2007.

[8] D. Archambault, T. Munzner, and D. Auber. GrouseFlocks: Steerable exploration of graph hierarchy space. *IEEE Trans. on Visualization and Computer Graphics*, 14(4), 2008. Accepted Jan 18, 2008, to appear.

[9] 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.

[10] 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.

[11] David Auber and Fabien Jourdan. Interactive refinement of multi-scale network clusterings. In *Proc. 9th Int. Conf. on Information Visualisation (IV'05)*, pages 703–709, 2005.

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

[13] M. Balzer and O. Deussen. Level-of-detail visualization of clustered graph layouts. In *Proc. of the 6th International Asia-Pacific Symposium on Visualization (APVIS'07)*, pages 133–140, February 2007.

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

[15] 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-Verlag, 2002.

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

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

[18] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM TOG (Trans. Graphics)*, 15(4):301–331, 1996.

[19] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[20] E. Di Giacomo, W. Didimo, L. Grilli, and G. Liotta. Graph visualization techniques for web clustering engines. *IEEE Trans. on Visualization and Computer Graphics*, 13(2):294–304, March/April 2007.

[21] T. Dwyer and Y. Koren. DIG-CoLa: Directed graph layout through constrained energy minimization. In *Proc. of Symposium on Information Visualization (InfoVis '05)*, pages 65–72, 2005.

[22] T. Dwyer, Y. Koren, and K. Marriott. IPSep-CoLa: An incremental procedure for separation constraint layout of graphs. *IEEE Transactions on Visualization and Computer Graphics (Proc. Vis/InfoVis 2006)*, 12(5):821–828, 2006.

[23] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. In *Proc. Graph Drawing (GD'05)*, volume 3843 of *LNCS*, pages 153–164. Springer-Verlag, 2005.

[24] 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.

[25] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[26] P. Eades and Q. Feng. Multilevel visualization of clustered graphs. In *Proc. Graph Drawing (GD'96)*, volume 1190 of *LNCS*, pages 101–112. Springer-Verlag, 1996.

[27] P. Eades and M. L. Huang. Navigating clustered graphs using force-directed methods. *Journal of Graph Algorithms and Applications*, 4(3):157–181, 2000.

[28] J. D. Fekete, G. Grinstein, and C. Plaisant, editors. *IEEE InfoVis 2004 Contest: The History of InfoVis*, 2004. www.cs.umd.edu/hcil/iv04contest.

[29] Q. Feng, R. Cohen, and P. Eades. How to draw a planar clustered graph. In *Proceedings of the First Annual International Conference on Computing and Combinatorics*, pages 21–30, 1995.

[30] 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.

[31] Y. Frishman and A. Tal. Multi-level graph layout on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1317, Nov./Dec. 2007.

[32] Y. Frishman and A. Tal. Online dynamic graph drawing. In *Proc. Eurographics/IEEE VGTC Symp. on Visualization (EuroVis'07)*, pages 75–82, 2007.

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

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

[35] E. Gansner, Y. Koren, and S. North. Graph drawing by stress majorization. In *Proc. of the 12th Symposium on Graph Drawing*, volume 3383 of *LNCS*, pages 239–250. Springer-Verlag, 2004.

[36] E. Gansner, Y. Koren, and S. North. Topological fisheye views for visualizing large graphs. *IEEE Trans. on Visualization and Computer Graphics*, 11(4):457–468, 2005.

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

[38] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. Technical report, Georgia Institute of Technology, 1994.

[39] S. Hachul. *A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs.* PhD thesis, University of Köln, Department of Computer Science, 2005.

[40] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Proc. Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 285–295. Springer-Verlag, 2004.

[41] S. Hachul and M. Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *Proc. Graph Drawing (GD'05)*, volume 3843 of *LNCS*, pages 235–250. Springer-Verlag, 2005.

[42] K. M. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229, 1970.

[43] 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.

[44] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. *Journal of Graph Algorithms and Applications*, 6:179–202, 2002.

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

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

[47] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *LNCS Tutorial*. Springer-Verlag, 2001.

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

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

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

[51] C. Kosak, J. Marks, and S. Shieber. Automating the layout of network diagrams with specified visual organization. *IEEE Trans. on Systems, Man, and Cybernetics*, 24(3):440–454, 1994.

[52] R. Kosara, T. J. Jankun-Kelly, and E. Chlan, editors. *IEEE InfoVis 2007 Contest: InfoVis goes to the movies*, 2007. www.apl.jhu.edu/Misc/Visualization/index.html.

[53] D. C. Lay. *Linear Algebra and Its Applications*. Addison-Wesley, 3rd edition, 1997.

[54] 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.

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

[56] T. Munzner. *Interactive Visualization of Large Graphs and Networks*. PhD thesis, Stanford University, 2000.

[57] T. Munzner. Process and pitfalls in writing information visualization research papers. In *Information Visualization: Human-Centered Issues and Perspectives*, volume 4950 of *LNCS*, pages 134–153. Springer, 2008.

[58] 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.

[59] A. Noack. An energy model for visual graph clustering. In *Proc. Graph Drawing (GD'03)*, volume 2912 of *LNCS*, pages 425–436. Springer-Verlag, 2003.

[60] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *In Proc. Supercomputing*, 1995.

[61] T. Pattison, R. Vernik, and M. Phillips. Information visualization using composable layouts and visual sets. In *Proc. of the 2001 Asia-Pacific Symposium on Information Visualization*, pages 1–10, 2001.

[62] A. J. Pretorius and J.J. van Wijk. Multidimensional visualization of transition systems. In *Proc. of the Interational Conferance of Information Visualization*, pages 323–328, 2005.

[63] H. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing*, volume 1353, pages 248–261. Springer-Verlag, 1997.

[64] P. Saraiya, C. North, V. Lam, and K. Duca. An insight-based longitudinal study of visual analytics. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1511–1522, Nov./Dec. 2006.

[65] 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.

[66] B. Shneiderman and A. Aris. Network visualization by semantic substrates. *IEEE Transactions on Visualization an Computer Graphics (Proc. Vis/InfoVis 2006)*, 12(5):733–740, 2006.

[67] 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-Verlag, 1999.

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

[69] Maureen Stone. Color in information display. IEEE Visualization 2006 Course Notes. Available as `http://www.stonesc.com/Vis06`, Oct 2006.

[70] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Trans. on Systems, Man, and Cybernetics*, 21(4), 1991.

[71] 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.

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

[73] W. T. Tutte. *Selected Papers of W. T. Tutte*, volume 1, chapter 24. How to Draw a Graph, pages 360–388. St. Pierre, Manitoba: Charles Babbage Research Centre, 1979. Reprint with permission of "How to Draw a Graph." In *Proc. London Mathematical Society (Series 3)*, 13:743–767, 1963.

[74] S. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000.

[75] F. van Ham and J.J. van Wijk. Interactive visualization of small world graphs. In *Proc. IEEE Symposium on Information Visualization (InfoVis'04)*, pages 199–206, 2004.

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

[77] C. Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann, 2004.

[78] C. Ware, H.C. Purchase, L. Colpoys, and M. McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.

[79] M. Wattenberg. Visual exploration of multivariate graphs. In *Proc. of SIGCHI conference on Human Factors in Computing Systems*, pages 811–819, 2006.

[80] M. Williams and T. Munzner. Steerable, progressive multidimensional scaling. In *Proc. IEEE Symposium on Information Visualization (InfoVis'04)*, pages 57–64, 2004.

[81] N. Wong, S. Carpendale, and S. Greenberg. Edgelens: An interactive method for managing edge congestion in graphs. In *Proc. IEEE Symposium on Information Visualization (InfoVis'03)*, pages 51–58, 2003.

[82] 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.

Appendix A

Parameter Settings to Obtain Results

In the following appendix, we document the parameter settings for all of the techniques and algorithms presented in this thesis used to generate the result images presented in this thesis.

A.1 ACE Algorithm

- all node weights set to 1.0
- power iteration has converged when angle between the eigenvector of the current iteration and the previous iteration is less than $1e^{-8}$

A.2 HDE Layout Algorithm

- number of dimensions to the high-dimensional space d is 50
- power iteration has converged when angle between the eigenvector of the current iteration and the previous iteration is less than $1e^{-3}$

A.3 GRIP Algorithm

The parameters are cited here for completeness. These values are actually the default values of the GRIP code supplied by S. Kobourov.

- dimensionality of layout is set to 2
- ideal edge length 32
- number of initial vertices 4
- displayPar set to 1

- random set to false
- FR_full set to false
- plot_all_vert set to false
- rounds is 20
- finalRounds is 30
- heat_fraction is 0.17
- r is set to 0.15
- s is set to 3.0

A.4 FM^3 Algorithm

The parameters are cited here for completeness. These values are actually the default values of the FM^3 code supplied by S. Hachul.

A.4.1 General Parameters

- save_rep_forces set to 0
- print_information set to 1
- fix_selected_nodes set to 0
- edgelenth_from_label set to 1
- rand_seed set to 100
- unit_edgelength set to 100
- edgelength_measurement set to 1
- divide_et_impera set to 1
- multilevel set to 1
- max_int_pos_exponent set to 40

A.4.2 Divide et Impera Step Parameters

- steps_for_rotating_componenets set to 10
- allow_components_to_be_tipped_over set to 1
- rectangle_offset set to 1
- random_fits set to 5
- random_runs set to 5

A.4.3 Multilevel Step Parameters

- enforce_few_edge_crossings set to 0
- planarity_required set to 1
- min_graph_size set to 50
- galaxy_choice set to 1
- random_tries set to 20
- change_max_iter set to 1
- max_iter_factor set to 10
- init_placement_way set to 1

A.4.4 Calculation Step Parameters

- force_model set to 2
- spring_strength set to 1
- strength_of_rep_forces set to 1
- rep_calculation set to 5
- stop_criterion set to 2
- threshold set to 0.01
- max_iterations set to 30
- cool_temperature set to 0

- coolvalue set to 0.99
- force_scaling_factor set to 0.05
- initial_placement set to 2
- plan_subgraph_layout set to 1

A.4.5 Postprocessing Parameters

- resize_drawing set to 1
- resizing_scalar set to 1
- fine_tuning_iterations set to 20
- fine_tune_scalar set to 0.2
- adjust_post_rep_strength_dynamically set to 1
- post_spring_strength set to 2.0
- post_strength_of_rep_forces set to 0.01

A.4.6 Repulsive Force Parameters

- resize_drawing set to 1
- resizing_scalar set to 1
- fine_tuning_iterations set to 20
- fine_tune_scalar set to 0.2
- adjust_post_rep_strength_dynamically set to 1
- post_spring_strength set to 2.0
- post_strength_of_rep_forces set to 0.01

A.5 TopoLayout Parameters

- Bushy Tree set to Bubble Tree
- Connected set to Connected Component Packing
- Deep Tree set to Hierarchical Tree (R-T Extended)
- Cluster Buster is unchecked
- Intl. Fast Ov. is checked
- Near-Mesh is Area HDE
- Node/Edge is 1.0
- Stren High is Circular
- Stren Low is GEM (Frick)
- Torques is checked
- Unkwn High is Circular
- Unkwn Low is GEM (Frick)

A.5.1 HDE Detection Algorithm

- all drawing parameters present in Section A.2
- minimum allowable variance (largest eigenvalue) is 100

A.5.2 Tree Walker Layout Algorithm

- minimum spacing between layers is 1.0
- minimum spacing between nodes is 1.0

A.5.3 HDE Layout Algorithm

- same as those presented in Section A.2

A.5.4 Area-Aware GEM Algorithm

- μ_r is the average radius of a node in the graph
- ideal edge length l is $\max(4, \mu_r)$
- maximum number of iterations is the number of nodes in the graph

A.5.5 Crossing Reduction Algorithm

- maximum number of iterations is the number of nodes in the graph

A.5.6 Fast Overlap Removal Algorithm

- x direction node border $1e^{-6}$
- y direction node border $1e^{-6}$

A.6 SPF Parameters

A.6.1 LGL Algorithm

- node size is 1.0
- grid size is 400
- ideal edge length is 10

A.6.2 Optimized LGL Algorithm

- $R_{\max}$ is the maximum radius of a node
- N is the number of nodes in the input graph
- Grid size is $400R_{\max}\sqrt{N}$
- ideal edge length is $10R_{\max}$

A.7 Grouse Parameters

- Layout parameters same as Section A.5 because TopoLayout used unmodified
- Number of animation frames set to 150

A.8 GrouseFlocks Parameters

- Layout parameters same as Section A.5 because TopoLayout used unmodified
- Number of animation frames set to 50
- Coarsening threshold set to 200

A.9 TugGraph Parameters

- Layout parameters same as Section A.5 because TopoLayout used unmodified
- Number of animation frames set to 50
- Coarsening threshold set to 200

Appendix B

Figure Permissions

A few figures presented in Chapter 2 appeared in publications of other authors. Credits are presented below:

- Figures 2.3(a) and 2.3(b) are reproduced with permission of their respective authors. Source images supplied by their respective authors.
- Figure 2.4 reproduced with the permission of Dr. Tim Dwyer and the IEEE (©2006 IEEE) [22].
- Figure 2.5 reproduced with the permission of Dr. Michael Jünger. Source image supplied by the author.
- Figures 2.6 and 2.7 reproduced with the permission of Dr. Yehuda Koren. Source image supplied by the author.
- Figure 2.8 reproduced with the permission of Dr. Tamara Munzner. Source image supplied by the author.
- Figure 2.9 reproduced with permission of Aleks Aris. Source image supplied by the author. Used with permission of Human-Computer Interaction Lab, University of Maryland.
- Figure 2.10(a) reproduced with permission of Dr. Emden R. Gansner and the IEEE (©2004 IEEE) [36].
- Figures 2.10(b) and 2.11 reproduced with the permission of Dr. Frank van Ham. Source images supplied by the author.