# GrouseFlocks: Steerable Exploration of Graph Hierarchy Space

Daniel Archambault, Tamara Munzner, David Auber

# GrouseFlocks: Steerable Exploration of Graph Hierarchy Space

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

***Abstract*— Several previous systems allow users to interactively explore a large input graph through cuts of a superimposed hierarchy. This hierarchy is often created using clustering algorithms or topological features present in the graph. However, many graphs have domain-specific attributes associated with the nodes and edges which could be used to create many possible hierarchies providing unique views of the input graph. GrouseFlocks is a system for the exploration of this graph hierarchy space. By allowing users to see several different possible hierarchies on the same graph, the system helps users investigate graph hierarchy space instead of a single, fixed hierarchy. GrouseFlocks provides a simple set of operations so that users can create and modify their graph hierarchies based on selections. These selections can be made manually or based on patterns in the attribute data provided with the graph. It provides feedback to the user within seconds, allowing interactive exploration of this space.**

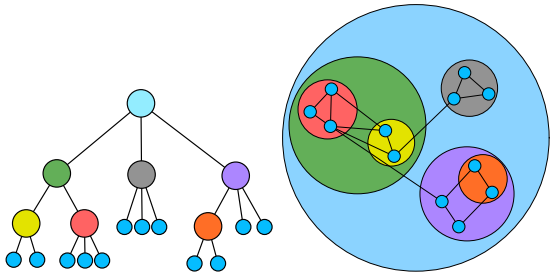
***Index Terms*— Graph Theory, Graph Drawing System, Graph Hierarchy**

## I. Introduction

GRAPH hierarchies have been exploited in order to simplify huge graphs to reveal elements of their structure. A **graph hierarchy** or **hierarchy**, shown in Figure 1 is defined as a recursive grouping placed on the nodes in this graph. For example, in a software engineering setting, where nodes are methods and an edge represents one method calling another, a hierarchy may group methods into classes on the small scale and packages on the large scale. **Metanodes** are the interior nodes of this hierarchy that contain a subset of nodes and a subset of the edges between those nodes. In our software engineering example, metanodes are nodes representing classes and packages. This subset of nodes and edges is known as a **subgraph**. A subgraph of interest to the user is a **feature**. The **leaves** in the hierarchy are the nodes of the input graph. In our software engineering example, these are the methods. As a metanode strictly contains or does not contain a leaf or metanode in the graph hierarchy, the topology of a hierarchy defined in this way is always a tree. Thus, we refer to the topology of the graph hierarchy as the **hierarchy tree**. In GrouseFlocks, we restrict ourselves to the visualization of graphs and their hierarchies. In many cases, representations other than a graph may be more effective for investigating a particular information space.

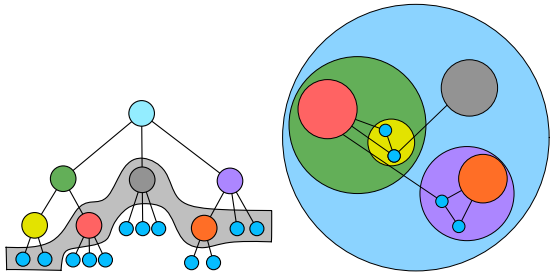
For large graphs, displaying a full layout of the entire graph may not provide a useful level of abstraction for users and can be visually overwhelming. Additionally, a layout of the entire graph is costly to compute, requiring minutes and sometimes hours [3]. Previous interactive systems used to explore this data have solved this problem by allowing users open or close metanodes to adjust the complexity of the view. This interaction specifies a **cut** of the graph hierarchy, namely the boundary between the visible and hidden parts of the graph, as shown in Figure 2. A cut defines which metanodes and leaves will be shown in the drawing and which will be hidden or abstracted away from the user. In the figure, the pink, orange and grey metanodes along with four leaves are on the cut and thus are opaque in the drawing. By allowing the user to interactively explore the graph hierarchy, we do not need to lay out the entire graph as only parts of the graph on or above the cut are visible. Leaves and metanodes below the cut are hidden from the user inside metanodes on the cut. A system which draws parts of the graph on demand as requested by the user is known as a **steerable** system.

In previous systems, most graph hierarchies have been generated automatically using graph topology [1], [2], [4], [30]. However, little attention has been paid to the investigation of



(a) Graph Hierarchy (b) Graph

Fig. 1. Illustration of a graph hierarchy. **(a)** Graph hierarchy and **(b)** 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 which contain leaves and other metanodes. A subgraph is the subset of leaves and metanodes contained inside a metanode.



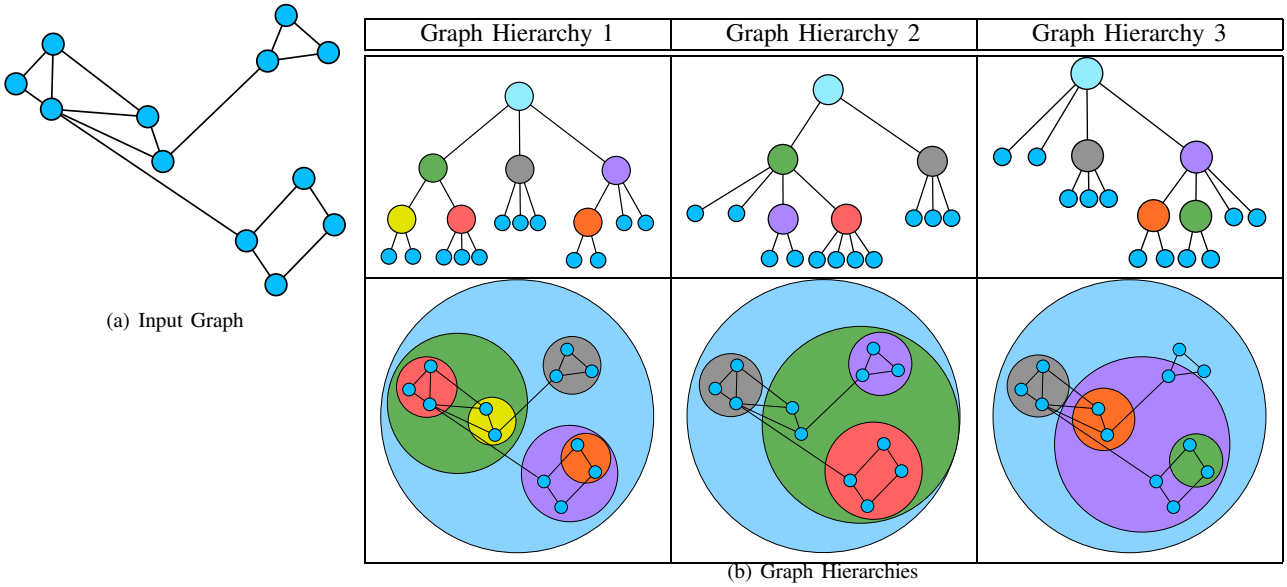
(a) Graph Hierarchy (b) Graph

Fig. 2. Illustration of a graph hierarchy cut. **(a)** Graph hierarchy with cut shown as grey curve and **(b)** graph hierarchy cut superimposed on top of a graph. Metanodes which appear on the cut are drawn opaque, hiding the subgraphs below them. The hierarchies are constructed by recursively grouping sets of leaves and metanodes into subgraphs which respect edge conservation and connectivity conservation constraints.

D. Archambault and T. Munzner are with the Department of Computer Science, University of British Columbia. Email {archam, tmm}@cs.ubc.ca

D. Auber is with LaBRI, the University of Bordeaux I. Email auber@labri.fr

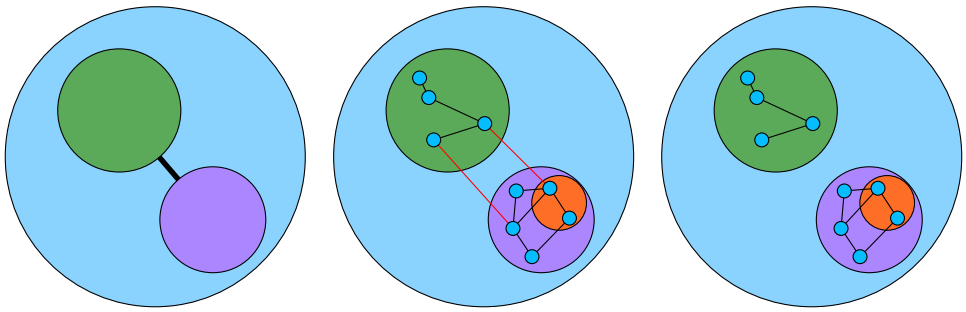
Manuscript received October 10, 2007, revised January 16, 2008, accepted January 18 2008.




(a) Input Graph

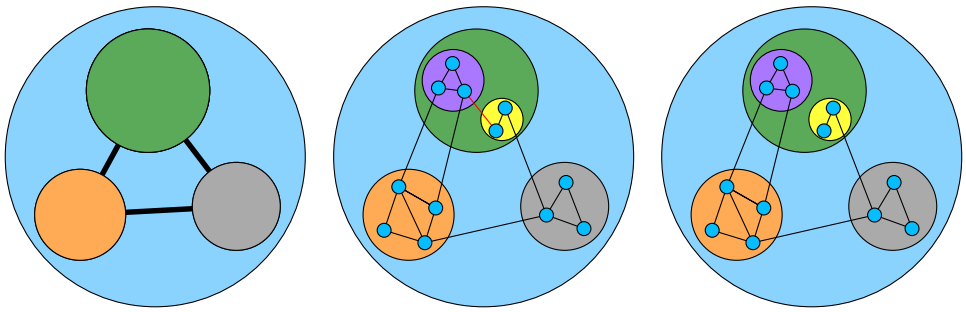
(b) Graph Hierarchies

Fig. 3. Multiple graph hierarchies superimposed on the same graph. In **(a)**, we see the input graph without any hierarchies superimposed on top of it. In **(b)**, we have a table of three of the many possible hierarchies which can be superimposed on **(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 which can be visualized by cuts, a single graph hierarchy is not suitable for all interesting views of the graph data.



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

Fig. 4. 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 which connect leaf nodes which are descendants of both metanodes. An invalid input graph is shown in **(c)** where edges do not connect descendants of the two metanodes.



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

Fig. 5. 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 topologically preserving.

hierarchy space which would allow users to see abstractions of their graph data based on attributes. In our software engineering example, it may prove useful to restructure the hierarchy to view methods which are or are not involved with some cross-cutting concern. A hierarchy based on this information would be better than the one of packages and classes to investigate the concern as significant parts of the graph can be abstracted away. Only a few systems allow hierarchy editing and these systems are limited to manual selection of nodes in the graph [7], [14] or provide limited tools for exploring the created hierarchy [25].

In GrouseFlocks, our principal contribution is providing a system to interactively explore and modify graph hierarchies with respect to a fixed large, input graph with attribute data. The system provides new ways of creating and modifying this hierarchy structure and allows the user to explore the newly created hierarchy in order to provide different abstractions of the graph to the user. We build on the Grouse system [4], which supports navigation and exploration of a graph and an associated hierarchy. In Grouse, both the hierarchy and underlying graph are required as input. GrouseFlocks does not require a hierarchy as input, but can accept one as input if needed, and allows the user to create and modify the hierarchy to suit their task. GrouseFlocks allows users to create metanodes using the attribute data present on the nodes in the graph. As in Grouse, the resultant subgraphs are drawn using appropriate algorithms based on their topological structure using the TopoLayout framework [5].

## II. Graph Hierarchy Space

A graph hierarchy provides a meaningful abstraction of the input graph by grouping subgraphs into metanodes. GrouseFlocks restricts the user to produce a **topologically preserving** graph hierarchy where there exists a path between two nodes in a cut if and only if there exists a path between two nodes in the input graph. If the user is still interested in understanding the structure of the input graph, then topologically preserving hierarchies are needed to ensure that all paths viewed in a cut actually exist. Topologically preserving hierarchies must respect two properties:

**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 4. In Figure 4(a), an edge exists in some upper level of the hierarchy between two metanodes. Figure 4(b) is valid since an edge in the input graph exists between two nodes in the input graph with one node contained by one metanode and the other contained by another. Figure 4(c) is not valid because no such edge exists in the input graph. Intuitively, any edge which exists in a cut is an abstraction of one or more edges which existed in the input graph.

**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 5(c), the metanode would imply that a path exists between the subgraphs. This placement implies the existence of a cycle which does not exist in the input graph. If we respect this restriction, as in Figure 5(b), these problems do not occur as there is only one connected component in the subgraph. If the graph is disconnected, we would have to break connectivity conservation at the root of the graph hierarchy with each connected component contained in their own metanode. However, as each component below the root is a connected component, subsequent levels of the hierarchy would be subjected to this property.

By respecting these two properties, we are guaranteed a topologically preserving graph hierarchy. Edge conservation ensures that any hierarchy edge is witnessed by some edge in the input graph, while connectivity conservation ensures that any path can continue through any metanode. Therefore, any path which exists in a cut represents at least one path between nodes in the input graph.

Even with the topologically preserving restriction, there can still exist an exponential number of hierarchies for a single input graph because there are as many hierarchies as there are subsets of nodes. Figure 3(a) presents a possible input graph, and Figure 3(b) gives three examples of possible hierarchies which can be superimposed on this input graph. In the first row of the table, we illustrate the hierarchy trees. In the second row of the figure, we draw the hierarchy trees on top of the input graph using containment, where a circular metanode contains all its children. As we can see from this simple example, the metanode structure imposed on the same graph can differ wildly from hierarchy to hierarchy. Since the hierarchy defines which cuts 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. In this paper, we call the space of topologically preserving hierarchies **hierarchy space**. GrouseFlocks allows users to investigate this space to find a graph hierarchy suitable for their tasks.

## III. Previous and Related Work

Graph drawing and visualization has been a very active area of research [10], [21], [22]. As this work investigates ways of exploring the space of graph hierarchies, we focus on work in graph drawing which involves the visualization of a graph and an associated hierarchy.

We divide the related work into three categories. Multilevel algorithms assign final positions to the nodes in the graph using graph hierarchies constructed using topology. Graph hierarchy exploration allows users to explore a fixed graph hierarchy on top of a fixed graph. Graph hierarchy editing allows users some limited operations to modify a given graph hierarchy.

### A. Multilevel Algorithms

Multilevel graph drawing algorithms automatically compute graph hierarchies that are used to accelerate or improve the layout process. The hierarchy construction algorithms of these approaches have been based on an estimates of maximal matching [31], graph filtration based on shortest path distance in GRIP [17], local graph connectivity in FM$^3$ [20], and topological features in TopoLayout [5].

None of these systems address interactive hierarchy exploration. They use the hierarchy to accelerate layout of the graph and present a final drawing. Although these techniques are able to quickly draw large graphs, for very large graphs, the cost of computing the entire layout of the graph is a barrier to exploration. Additionally, simply presenting the final drawing of a large graph is prone to large amounts of edge and node clutter and occlusion. However, multilevel algorithms do provide a base for interactively drawing graph hierarchies.

### B. Attribute-Based Clustering and Drawing

Several works have used attribute data as the driving characteristic of the graph in order to produce a drawing or clustering of a graph. However, much of this work does not take the topological structure of the graph into account.

Pretorius and van Wijk [26] describe a system to map multi-dimensional attribute data associated with the nodes and edges of a graph to spatial 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, accentuating areas of maximal variance in the data.

The PivotGraph system of Wattenberg [32] determines the spatial position of nodes using using high dimensional attribute data. The system divides nodes into equivalence classes, placing two nodes in the same class if they have the same value for the attribute being examined. These equivalence classes are used to influence spatial position or simplify the graph structure, producing a final drawing.

Pretorius and van Wijk [27] authored a second work which allows for the exploration of a state transition diagram using an attribute-based hierarchy. In their work, they recursively divide the nodes of the graph recursively by attribute forming a graph hierarchy. This hierarchy is drawn as a rooted tree with the nodes as leaves of this hierarchy. The edges of the state transition diagram are drawn as an arc diagram between the leaves.

Semantic substrates for graph layout, introduced by Shneiderman and Aris [29], allows nodes to be placed based purely on attribute data, with graph topology having no effect. Intelligent edge filtration techniques display underlying adjacency and topology information to encourage understanding of the graph with the attribute as the driving feature of the visualization.

These systems produce drawings which encode the spatial positions of nodes in the graph using only attribute data. However, for many questions, users would like to investigate attribute data in the context of graph topology. For example, a user may want to understand how the servers of a network at the University of British Columbia connect to those at the University of Bordeaux through the Internet. In order to answer this questions, we require topological information and attribute data such as server location (UBC, Bordeaux, other). In this paper, we describe a system which provides a compromise between purely attribute-based systems and purely topological ones.

### C. Graph Hierarchy Exploration

In interactive approaches to hierarchy exploration, the entire graph is not shown at once. These systems present abstractions of the input graph where the user can choose cuts interactively to display metanodes and leaves. In this section, we present two types of graph hierarchy navigation systems: those which require a pre-existing layout and those which do not. The latter we call steerable systems.

*1) Existing Layout Required:* These systems are some of the oldest graph hierarchy exploration techniques. They quite frequently exploit properties of the layout into order to form a graph hierarchy on top of the pre-existing graph to illustrate graph and hierarchy structure in a single drawing. Various techniques exist including: visualizing the graph and associated hierarchy extruded into the third dimension [13], multi-focal fisheye approaches where metanodes are expanded and viewed in the context of the entire graph [28], topological fisheyes where abstract versions of the graph are presented far away from a focus centre [18], linking the graph hierarchy to a separate treemap view [1], interactively visualizing hierarchies of small world clusterings [30], and visualization of complex software in three dimensions using level of detail techniques [8].

In all of these techniques a static layout is computed once up front, and a static hierarchy is computed from it relating to the spatial position of the vertices in the drawing. GrouseFlocks supports the creation of multiple possible hierarchies that do not depend on a particular graph layout and requires a steerable system as a base.

*2) 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. As they do not require this layout, they can more readily be used for exploration of graph hierarchy space.

Di Giacomo *et al.* [11] present a system for visualizing a graph and a associated hierarchy both created by a search engine query. In their system, a search query produces a graph: a set of documents, which are the nodes, and an edge exists if documents are sufficiently semantically related. The strength of the relationship is encoded with an edge weight and a topological clustering algorithm is recursively applied to the graph, forming a graph hierarchy. Cuts to this hierarchy are visualized through orthogonal graph drawing algorithms as the user explores the hierarchy.

ASK-GraphView [2] is a powerful steerable system with multiple linked views. It computes hierarchies using a topological 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.

Grouse [4] focuses on interactive exploration of a graph and an associated hierarchy. As input, the system takes a graph and an associated hierarchy computed by recursive decomposition of the graph into topological features as presented in TopoLayout [5]. Cuts of this graph hierarchy are used to steer user exploration of the input graph. It chooses which layout algorithm to use for each metanode based on the topological structure of its subgraph.

None of these systems support creation of multiple possible hierarchies based on attribute data. In GrouseFlocks, we build on top of the Grouse system that supports interactive navigation of the created hierarchies. The GrouseFlocks coarsening operator has the goal of maintaining responsiveness, as does the ASK-GraphView thresholding mechanism, but our technique attempts to preserve the underlying features in the created hierarchy more faithfully.

### D. User-Specified Hierarchy Editing

Three systems in the literature allow users to manually modify graph hierarchies in a limited way. These systems rely heavily on user actions for hierarchy specification with little automated support.

The DA-TU system [14] 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 cut and manually select all nodes to change the hierarchy. DA-TU is also steerable, using a modified version of a force-directed approach which biases the 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 which have changed. This approach does not scale to large graph sizes.

The work of Auber and Jourdan [7] supports interactive hierarchy editing. The paper's primary focus is on fast algorithms for replacing subgraphs with metanodes which can be done 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.

The Clovis system [25] 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 effect any aspect of the appearance of a node or edge except position. However, the system must present all nodes of the input graph at all times with no tools beyond selection for graph hierarchy navigation. Additionally, the user must manually specify for each metanode when the subgraph it contains should be redrawn and with what algorithm. These two aspects of the system limit its scalability to hierarchies with a small number of metanodes and input graphs of small sizes.

These three systems rely heavily on user action in order to specify a graph hierarchy. They are good tools for graph hierarchy creation, but do lack support for hierarchy navigation and do not constrain graph hierarchies to be topologically preserving. In GrouseFlocks, we provide an integrated system for hierarchy exploration and creation based on the attribute data associated with the nodes of the input graph. The system provides higher level operations to assist the user in hierarchy creation and exploration.

## IV. Interface

The interface for GrouseFlocks is shown in Figure 6, with operations to facilitate hierarchy manipulation and exploration. In Section IV-A, we briefly review the steerable graph interface previously described in Grouse [4]. Then, in Section IV-B, we introduce our hierarchy creation and modification method. The section only describes how the user interacts with the GrouseFlocks interface. Algorithmic details of these operations are presented in Section V.

### A. Hierarchy Navigation

The navigation capabilities of GrouseFlocks, which were previously described in Grouse [4], consist of a tree view and a graph view. These views allow users to modify the cut to the graph hierarchy which is used to define our hierarchy editing operations. The **tree view**, **(3)**, appears directly below the progress bar on the left hand side of Figure 6. 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 cut with a node-link diagram. The graph hierarchy is superimposed on the cut using containment: a metanode contains its subgraph or all its children. The graph view supports pan and zoom through its two-dimensional view.

Metanodes in the graph hierarchy can be in one of three states: **open**, where they show the subgraph beneath the metanode in the hierarchy contained within its enclosing circle; on the **cut**, visible and drawn as an opaque circle without further detail; or **hidden**, not visible in the graph view because they are contained inside a cut metanode. In the graph view, containment illustrates the graph hierarchy. We show the graph relationships by drawing edges from leaf nodes in the usual way, and 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 boxes on the screen.

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)**.

### B. Hierarchy Creation and Modification

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

*1) 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** 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. If text is unsuccessfully captured, the category would be an empty 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.

*2) Hierarchy Modification:* Both of the high-level hierarchy modification operators in GrouseFlocks carry out actions based on the current selection sets and the current graph 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 cut, and creates new metanodes based on the selection sets. The **Merge-At-Cut** preserves the hidden structure of the old hierarchy, and merges the selected sets within each open metanode. In both cases, GrouseFlocks always produces a new topologically preserving hierarchy.

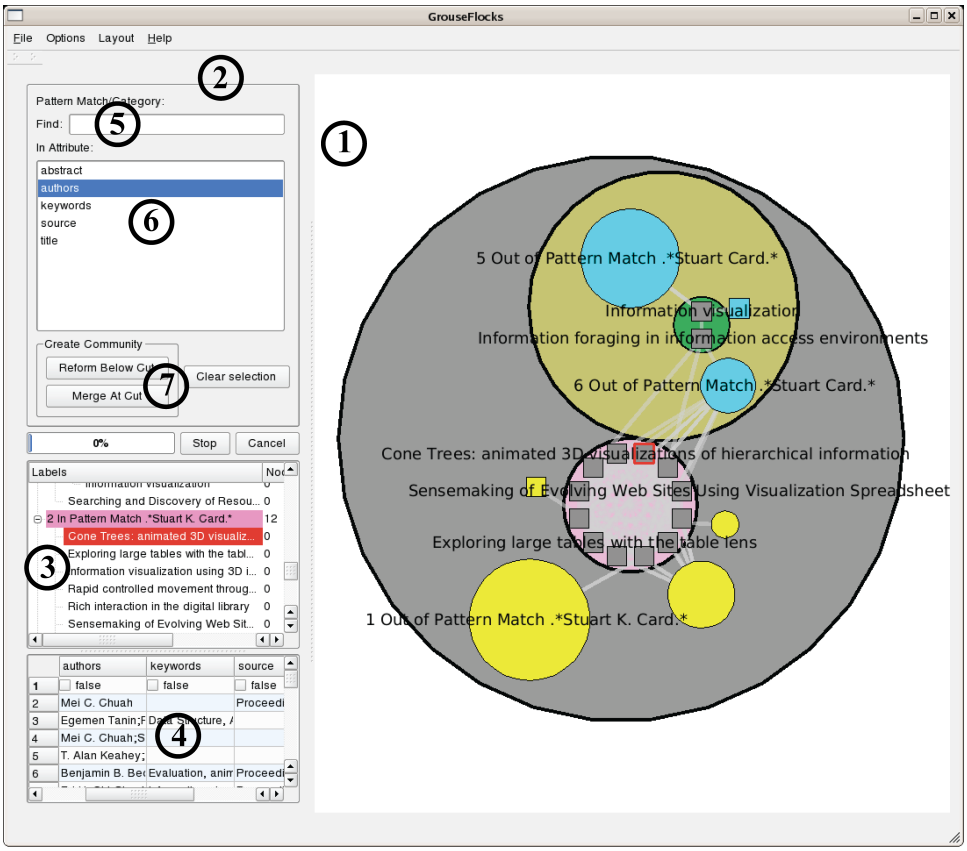



Fig. 6. The GrouseFlocks user interface. The graph view, **(1)**, illustrates cuts to the hierarchy. On the left, **(2)**, are the searching capabilities of GrouseFlocks to select portions of the underlying graph. The tree view, **(3)**, shows the current graph hierarchy being navigated. The information which appears in the labels can be selected in table **(4)** where checkboxes indicate which attributes are present in the nodes. 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.

## V. Algorithms

In this section, we describe all the algorithms used to implement the interface described in Section IV. We describe the algorithms used for hierarchy navigation, selection, and metanode creation in the system.

### A. Hierarchy Navigation

GrouseFlocks uses a progressive version of the TopoLayout approach [5], drawing the subgraph below a metanode with an algorithm appropriate for its topology. Specific algorithms are used to draw trees [9], [19], mesh-like components [23], circular layout for complete graphs, and force-directed [16] when the topology is unknown. Additionally, all overlaps between metanodes and leaves are eliminated with an overlap elimination pass [12]. A crossing reduction pass minimizes edge-edge and edge-metanode crossings by rotating metanodes according to a computed torque value.

GrouseFlocks uses the interactive hierarchy browsing capabilities of Grouse [4]. When a metanode on the cut is drawn, its size is an estimate based on the number of its descendants. After it is opened by the user, a more accurate estimate of the space required to draw without overlap can be made. This space requirement is propagated up the hierarchy to the root and affected subgraphs are relaid out, while minimizing changes to the positions of affected nodes. Edges which connect the subgraph inside the opened metanode are introduced to the cut in linear time, using the interactive refinement approach [7]. Finally, linear interpolation of node positions smoothly morphs the previous into the new hierarchy cut.

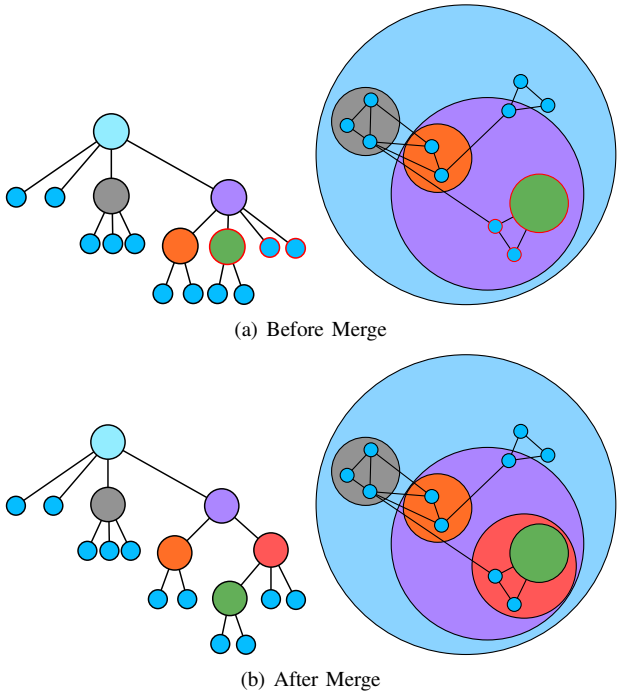
This steerable navigation system allows users to explore their graph and decide if they would like to modify the current graph hierarchy. Once they navigate to a specific cut, selection and metanode creation algorithms are used to modify the actual hierarchy.

### B. 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 IV-B.1, 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 cut, so the user is able to determine which metanodes contain selected leaves.

*1) 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 cut.

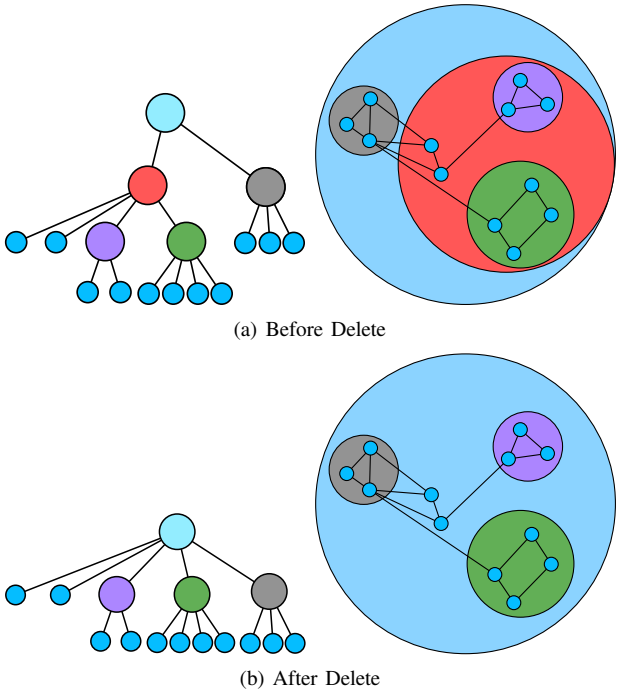
*2) Category:* A category selection divides subgraphs into multiple metanodes, depending on the number of unique strings found. As in pattern match, the hierarchy is recursively searched, and the substring of interest for each node is recorded based on



(a) Before Merge

(b) After Merge

Fig. 7. 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

Fig. 8. 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, we only apply a delete recursively to repartition nodes in a graph hierarchy.

the regular expression. If more than one category is seen below a particular cut metanode, a selection is propagated to the cut.

### C. Hierarchy Modification

The hierarchy modification operators described in Section IV-B.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 [14] system and in Auber and Jourdan [7].

The **Merge** operation, illustrated in Figure 7, 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 8, 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 which exists below a metanode.

We now describe in detail 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 cut.

*1) 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 9 shows an example.

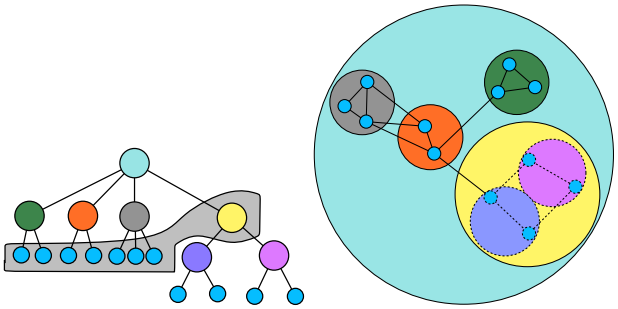
The Reform-Below-Cut operation considers the selected set and the cut metanodes in the graph hierarchy. For each cut metanode, the operation begins by recursively deleting all hidden metanodes using the metanode Delete operation, as shown in Figure 9(b). Once complete, only leaves are present in the 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.

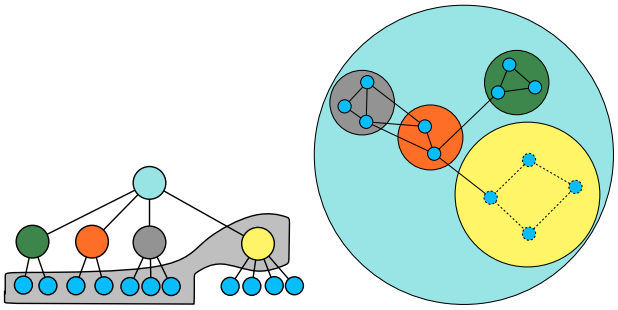
*2) Merge At Cut:* The Merge-At-Cut operation modifies the current hierarchy by merging sets of selected nodes on the cut, respecting both metanode boundaries and connectivity conservation. The operation is simply a metanode Merge operation applied to the contents of each open metanode separately.

### D. Coarsening

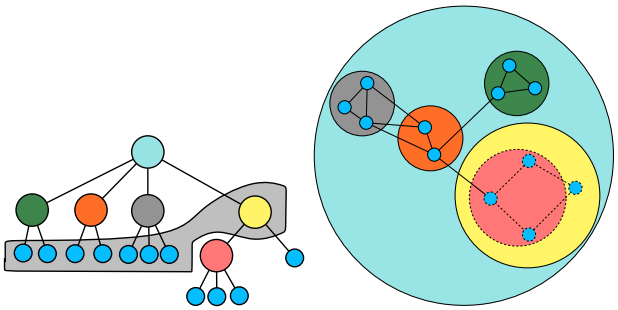
As the user navigates the current hierarchy, the user may request to open a metanode which is too large to lay out interactively. One way to ensure that exploration remains interactive in these cases is to coarsen the graph below the metanode to a size specified by the user as acceptable for visualization purposes. We design our coarsening algorithm to work on graphs with a few large features and many small ones. The algorithm assumes that large features are more interesting to the user and preserves them directly below the metanode being opened. Smaller features are abstracted away inside metanodes. The algorithm assumes an unweighted graph. It performs some topological, feature-based



(a) Reform-Below-Cut Initial State



(b) After Recursive Delete



(c) Reform-Below-Cut Complete

Fig. 9. Example of the Reform-Below-Cut hierarchy modification operator. We show the actions taken on a single metanode in the cut, the yellow node 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)**.

clustering, namely tree detection, like in ASK-GraphView [2], but it does not recursively detect topological features. Rather, our 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 10, attempts to preserve these major features and merges smaller features into metanodes. 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.

For the purposes of describing our algorithm, let $g$ be the graph contained inside the metanode which is being opened. The graph $g$ is too large to lay out because the number of nodes in it exceeds some user-specified threshold. Our algorithm is divided into two stages: a first stage which performs tree detection and the second stage which performs edge contraction.

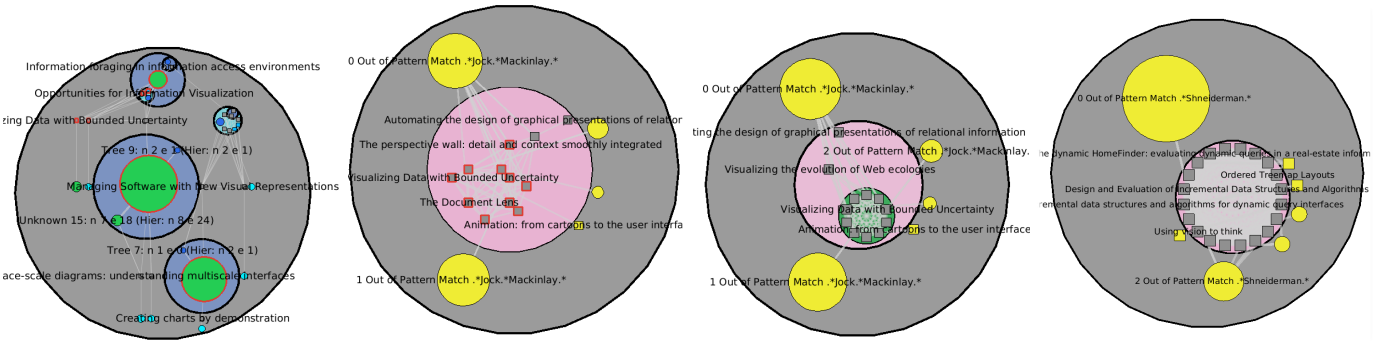
```
coarsen (Graph g, int thresh)
  while (g.|N| > thresh)
    detectTrees (g, thresh) {This halts if below threshold}
    g.N ← sortNodesBySubgraphSize (g.N)
    marked ← ∅
    contract ← ∅
    for all (n ∈ g.N)
      if (g.|N| − |contract| > thresh)
        break;
      if (n ∈ marked)
        continue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

Fig. 10. 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 $O(N \log^2 N + E \log N)$.

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 which coarsen very slowly using edge contraction and and would dominate the resultant coarsened graph if left in for the coarsening procedure.

The second stage performs a form of edge contraction on $g$ which 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, we scan all adjacent edges whose opposite nodes 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. The edge contraction procedure described above requires $O(|N| \log |N|)$ time for the node sort and $O(|N| + |E|)$ time to determine which edges to contract. Thus, if each pass of edge contraction removes half of the nodes of $g$ has complexity $O(\log |N|(|N| \log |N| + |N| + |E|))$ or $O(|N| \log^2 |N| + |E| \log |N|)$.




(a) Topological (b) Jock.*Mackinlay (c) Jock D. Mackinlay (d) Ben Shneiderman

Fig. 11. Comparison of four hierarchies with respect to the underlying `Coauthor` graph. The topological hierarchy 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, the unselected ones in the figure. In hierarchy **(c)**, a Merge-At-Cut is performed on this selection. The green metanode contains all *Jock D. Mackinlay* papers while the two leaves in the pink metanode are *Jock Mackinlay* papers. Hierarchy **(d)** shows Ben Shneiderman papers, in pink, separated from the rest of the graph in yellow. Refer to Figure 6 for a decomposition of the same dataset for papers coauthored by Stuart K. Card.

## VI. Metanode Labeling and Colouring

Enforcement of connectivity conservation through a merge operation may create more than one metanode which can be confusing to the user. To disambiguate these situations, we encode the type of selection which created this metanode in its colour and label.

Metanode labels consist of three parts. The first part is a unique number, identifying the metanode. The second part consists of the type of selection used to create the metanode. If the selection used was a pattern match, the label contains one of two strings: *In Pattern Match* which indicates the contents of this metanode matched the regular expression or *Out of Pattern Match* which indicates the contents of this metanode did not. If the metanode was derived from a category selection, the keyword *Category* is present. The third part of the label indicates which selection created this metanode. In the case of pattern match, the matching or unmatching regular expression is present in the label. In the case of category, the labels contain the category of the metanode. For example, if the category search was completed on movie genre and this metanode was a set of action movies, the category would be *action*.

Metanode colour is mapped to whether or not the entered regular expression was successfully matched. For pattern match, it maps whether or not the metanode matches the regular expression. For category, it considers empty string categories as unmatched and all other categories as matched. The system currently rotates through three match/unmatch pairings for metanode colour: pink/yellow, green/light blue, and orange/dark blue. The colour pairs are used in the indicated order and cycle through after the third selection.

## VII. Comparison of Hierarchies

We implemented GrouseFlocks using the Tulip graph drawing libraries [6] and Grouse [4]. In this section, we compare the interactively created hierarchies of GrouseFlocks to the static topological feature hierarchies of Grouse. ASK-GraphView [2] produces similar hierarchies as it detects the same set of topological features as Grouse in the same order. These tools were selected from the previous work because they present the only tools which allow steerable exploration of a static input graph and an associated hierarchy. We used a 3.0GHz Pentium IV with 3.0GB of memory running SuSE Linux with a 2.6.5-7.151 kernel. We used a subset of the InfoVis 2004 contest dataset [15] and the InfoVis 2007 contest data set [24].

The InfoVis 2004 dataset [15] consists of papers and their authors with additional information such as keywords, abstract, title, and location of publication. We generated a subset of this dataset 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 the two papers had at least one author in common. We call this graph `Coauthor` in the analysis.

The InfoVis 2007 dataset [24] consists of movies and information about them including: cinematographer, director, female actors, male actors, genre, Oscars, and sizes of each of these categories. From this data, we generated two movie graphs. 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 the largest connected component of all movies which had ratings, containing 9,475 nodes and 140,721 edges. The second dataset consisted of the largest connected component in the entire movie database, `Movie Large`, containing 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.

### A. Coauthor

Figures 11 and 12 show the `Coauthor` dataset. Figure 11(a) shows a topological decomposition from Grouse; the results from ASK-GraphView would be similar. After opening a few metanodes, we performed a search for *Jock Mackinlay* on the dataset, and metanodes containing nodes with this attribute are highlighted in red. We see that these nodes are scattered across

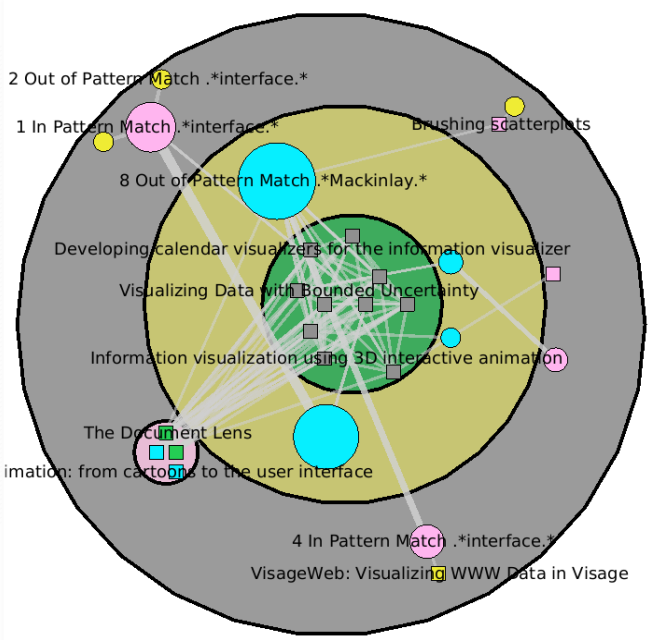



Fig. 12. A hierarchy on the `Coauthor` graph, decomposing it into two levels. First, papers which contain the keyword attribute *interface* in pink are separated from and those which do not, in yellow. The second level separates papers with *Mackinlay* as an author, in green, from the two which do not, in cyan. There is one large component of papers which do not contain the keyword *interface* on which Mackinlay is an author: the green component inside the yellow component. There are four papers which contain the keyword interface on which Mackinlay is an author: the expanded pink component in the lower left corner.

many metanodes, which is unsurprising because the author attribute data was not taken into account during the topological decomposition. The other attribute searches described below were similarly scattered across many metanodes.

Figures 11(c), and 11(d) 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 we originally explored this graph, we did not see complete graphs for Mackinlay: his name included a middle initial on some papers, and did not include a middle initial on others. Figure 11(b) shows a selection for *Jock.*Mackinlay* which captures all Jock Mackinlay's papers. We then perform a second search for *Jock D. Mackinlay* and those papers are highlighted in red in the figure. We then perform a Merge-At-Cut operation to group all of these papers into a single metanode. The result is shown in Figure 11(c). Figure 11(d) shows the complete subgraph of Ben Shneiderman found after only one search.

Figure 12 shows a different hierarchy generated on top of the same `Coauthor` graph. We first performed a Reform-Below-Cut operation after searching for *interface* in the keywords attribute. Matches are pink, and non-matches are yellow. We then performed a second Reform-Below-Cut operation after searching for *Mackinlay* in the authors attribute, with matches in green and non-matches in cyan. We see that 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, and we can see from the label that one is *The Document Lens*.

### B. Movie Small

We now show how GrouseFlocks can be used to investigate actors and the types of movies in which they appear.

Figure 13 shows the `Movie Small` dataset. Figure 13(a) shows a standard topological decomposition of the dataset. This hierarchy illustrates the topological features of the dataset, but it does not give information about the movies in which an actor appears. In this figure, we have performed a search for all female actors containing the name Stone. Any cut metanode which contains such an actor is highlighted red. As a topological decomposition does not consider this attribute information, the data is scattered all over the hierarchy. We see that exploring a single static hierarchy, as is supported by systems such ASK-GraphView and Grouse, does not provide a useful partition for this task.

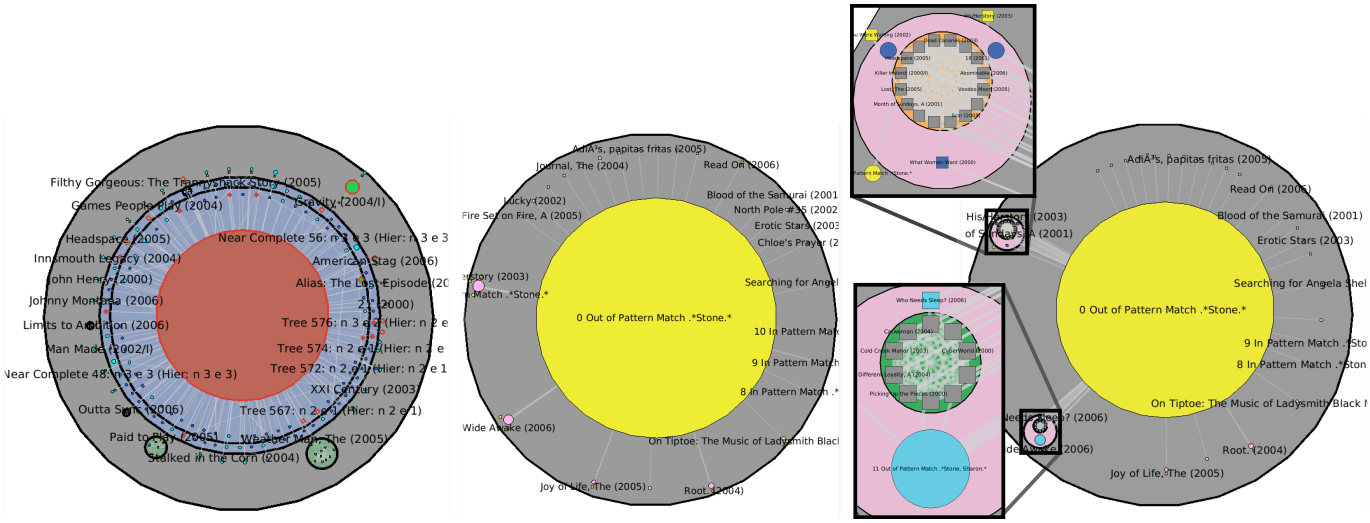
By using the GrouseFlocks and the attribute data of this dataset, it is possible to divide the datasets into sets of movies in which an actor has appeared. In Figure 13(b), we have first divided the dataset into sets of female actors with the name *Stone*. Subsequently, we perform second and third Reform-Below-Cut operations to reveal movies containing *Sharon Stone* in the bottom inset of Figure 13(c) 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.

### C. Movie Large

In Figure 14, the 17,192 movies and 220,321 edges between them have been decomposed topologically and by attribute data. In Figure 14(a), a topological feature hierarchy like those used in ASK-GraphView and Grouse is explored using the system. The remaining figures depict the exploration of movie genres and directors and actors associated with them.

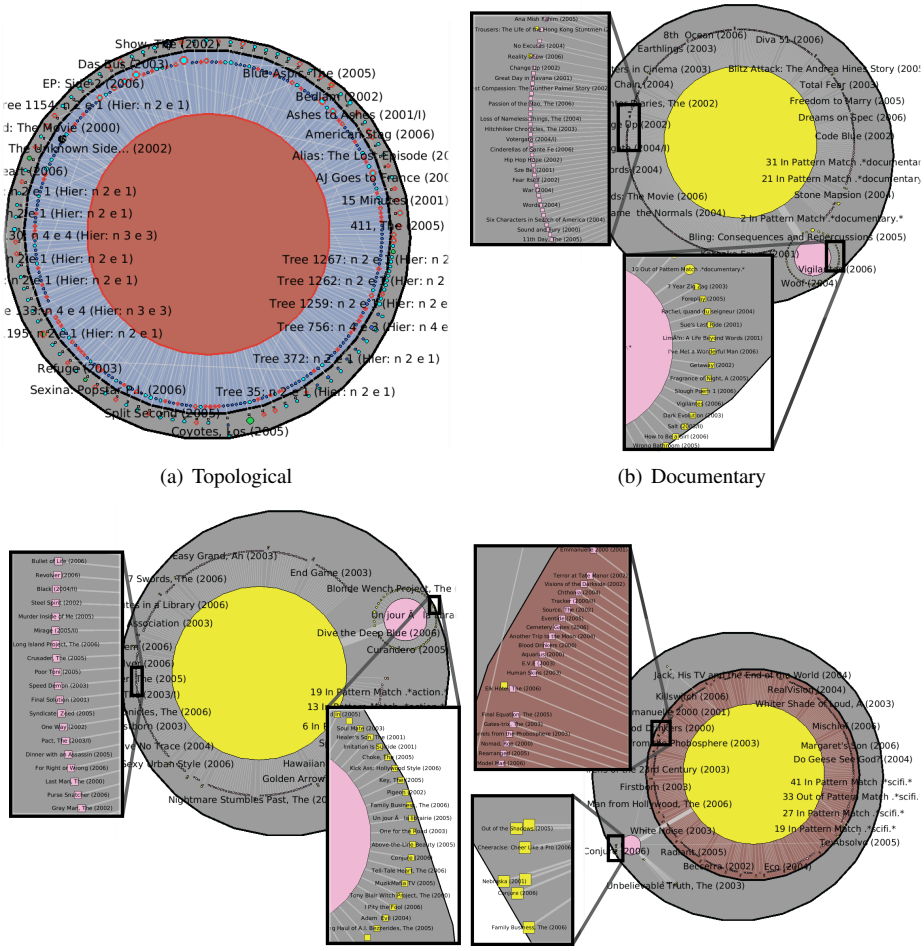
The topological decomposition of the graph, presented in Figure 14(a), demonstrates much of the clique structure in the dataset. Cliques represent the set of movies a single actor or a group of actors have acted in together. Beyond answering questions about the types of movies an actor or a group of actors participates in, it is difficult to get any more information from these drawings. We have selected the documentaries in this hierarchy and see that they are spread through the topological hierarchy. Thus, this topological hierarchy is not suitable for investigating movie genre.

In the remaining diagrams of Figure 14, 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 14(b), the pink components are documentary metanodes while the yellow components are metanodes which 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, we have two large subgraphs




(a) Topological (b) Stone (c) Stone + Sharon + Dee Wallace

Fig. 13. Comparison of a single topological hierarchy to multiple hierarchies produced by GrouseFlocks. Hierarchy **(a)** shows a standard topological hierarchy of the `Movie Small` dataset. The red selection shows the location of female actors with *Stone* in their name distributed all over the hierarchy. Hierarchy **(b)** shows a hierarchy where movies which contain at least one female actor with the name *Stone* are grouped into pink metanodes and movies which do not are yellow. In Hierarchy **(c)**, 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.




(a) Topological (b) Documentary

(c) Action (d) Science Fiction

Fig. 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 topological decomposition in used. The decomposition provides some insight into the graph's topological structure, and illustrates some cliques of actors, but does not help understand the data with respect to its attributes. In **(b) - (d)**, the graph is split into metanodes by movie genre: documentary, action, and science fiction. We see 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.

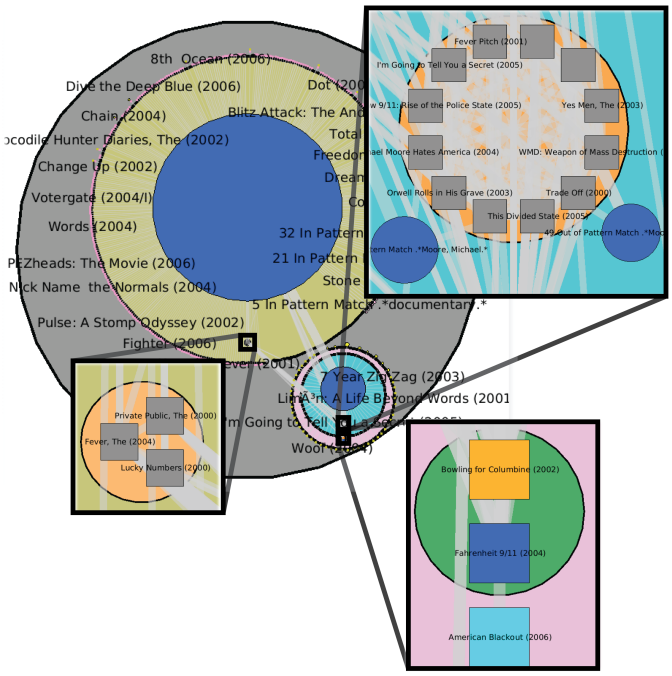



Fig. 15. Further refinement of the documentary genre hierarchy shown in Figure 14(b). We separate movies directed by *Michael Moore* and then movies acted in by *Michael Moore*.

with few nodes being outside these metanodes. This trend persists for action movies as seen in Figure 14(c), and science fiction movies as seen in Figure 14(d). In the science fiction genre, subgraph structure is more fragmented, and therefore, coarsening was required. The coarsened metanode has been opened and is the brown open metanode in the diagram.

In Figure 15, we further refine the documentary hierarchy of Figure 14(b) by performing two more Reform-Below-Cut operations. We first find movies directed by Michael Moore by selecting from the *director* attribute, with matches in green and non-matches in cyan. We then separate movies in which he acted by selecting from the *male actor* attribute, with matches in orange and non-matches in blue. After creating this new hierarchy, we see 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 in this dataset, 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, we see the three movies in which he acted which are not documentaries: *The Private Public*, *Lucky Numbers*, and *The Fever*.

Using GrouseFlocks, we were able to create a graph hierarchy which illustrated information about Michael Moore and the movies in which he acts and directs. This information is not visible in the topological graph hierarchy as the relevant nodes are scattered across many metanodes, making it difficult to see these relationships in the graph. Interactive creation and modification of multiple hierarchies on top of the same base graph allowed us to iteratively refine hierarchies until we could define the precise one that allows us to easily gather information about a specific question of interest.

## VIII. Initial User Feedback

Many of the operations discussed in this paper were motivated by the task requirements of a computer networking researcher, who has a rich collection of datasets capturing server communication patterns for both the Internet and smaller intranets[1]. However, these datasets are sufficiently complex that techniques presented here only partially address those requirements, so a full design study addressing these tasks remains as future work.

We also have initial feedback from graduate students and an information visualization researcher, who experimented with GrouseFlocks. Many of the users found GrouseFlocks an intuitive way to reduce clutter in a large graph and to visualize components of interest to them. A few of the users found the pink/yellow, green/teal, and blue/orange colouring scheme of GrouseFlocks somewhat hard to follow, and one suggested instead using hue to indicate the query and saturation to distinguish matching from unmatching items.

Some people had trouble understanding how containment is used to convey information about the dataset, especially when the nesting is three or more levels deep. While our approach has the advantage of supporting an arbitrary number of composition relationships, we will consider how to add additional support for users remembering hierarchical relationships.

## IX. Discussion

GrouseFlocks heavily exploits spatial cues in order to convey the parts of the graph with similar attributes. As spatial proximity is the strongest visual cue, it clearly conveys elements of the graph with the same or similar values for an attribute. The disadvantage 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 spatial proximity and minimize motion during operations on the graph and hierarchy would be of benefit to the system.

GrouseFlocks aids users explore the exponential space of hierarchies on a large input graph. One could view GrouseFlocks as turning the problem of understanding a large graph into understanding a large space of hierarchies. However, by abstracting a large graph into a hierarchy which has meaning to a user, we eliminate the clutter of node and edge occlusion that is a major obstacle to scalable graph drawing. GrouseFlocks is a first step in guiding the user through this abstract space. Future research is required to see if hierarchy space is difficult for users to navigate.

One of the strongest aspects of GrouseFlocks is that many hierarchies can be investigated within minutes. Previous approaches required construction of an full graph hierarchy and/or layout before investigation of graph hierarchy space on an input graph could begin. Entire initial layouts or hierarchy computations may take minutes to hours before the investigation of the data can begin. As neither a graph hierarchy nor a layout is required by GrouseFlocks, the system scales to larger datasets beyond previous approaches. It only requires about a minute to investigate many possible initial decompositions for the graph hierarchy before proceeding to the next level.

[1] http://www.cheswick.com/ches/map

## X. Conclusion

We have presented GrouseFlocks, a system for automated exploration of graph hierarchy space with a fixed graph with attributes. The system generates hierarchies which are reflective of the underlying graph topology by requiring that subgraphs respect edge conservation and connectivity conservation. Navigation of the a graph hierarchy in the system is supported high level operations to help create new graph hierarchies. The system was tested on a coauthorship graph and a movie dataset. We demonstrated that different graph hierarchies provide different views of the data to users, allowing certain view of the data to be illustrated more clearly beyond the strict topological hierarchies provided by ASK-GraphView and Grouse.

In the future, we would like to investigate applications of this technique in computer networking, systems biology, and social networks. Preliminary assessment of the system in some of these areas has proved to be successful. It would be interesting to further investigate additional operations on graph hierarchies that would be intuitive to scientists in these application domains.

## Acknowledgments

The authors would like to thank Bill Cheswick for his useful feedback throughout the project. Partial funding was provided by the ACI Jeunes Chercheurs *Cube de Données: Construction et Navigation Interactive* and INRIA IPARLA. We thank the visualization research groups at the University of Victoria and the University of British Columbia for their comments.

## References

[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] 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), Sept.-Oct. 2006.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

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

[27] A. J. Pretorius and J. van Wijk. Visual analysis of multivariate state transistion graphs. *IEEE Trans. on Visualization and Computer Graphics (Proc. Vis/InfoVis '06)*, 12(5):685–692, 2006.

[28] D. Schaffer et al. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Trans. on Computer-Human Interaction (TOCHI)*, 3(2):162–188, 1996.

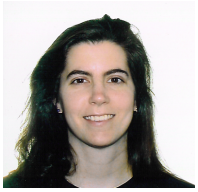
[29] B. Shneiderman and A. Aris. Network visualization by semantic substrates. *IEEE Transactions on Visualization an Computer Graphics (Proc. Vis/InfoVis 2006)*, 12(5):733–740, 2006.

[30] F. van Ham and J. van Wijk. Interactive visualization of small world graphs. In *Proc. IEEE Symposium on Information Visualization (InfoVis'04)*, pages 199–206, 2004.

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

[32] M. Wattenberg. Visual exploration of multivariate graphs. In *Proc. of SIGCHI conference on Human Factors in Computing Systems*, pages 811–819, 2006.

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



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

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