Tugging Graphs Faster: Efficiently Modifying Path-Preserving Hierarchies for Browsing Paths

Daniel Archambault, Tamara Munzner, David Auber

Tugging Graphs Faster: Efficiently Modifying Path-Preserving Hierarchies for Browsing Paths

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

Abstract—Many graph visualization systems use graph hierarchies to organize a large input graph into logical components. These approaches detect features globally in the data and place these features inside levels of a hierarchy. However, this feature detection is a global process and does not consider nodes of the graph near a feature of interest. TugGraph is a system for exploring paths and proximity around nodes and subgraphs in a graph. The approach modifies a pre-existing hierarchy in order to see how a node or subgraph of interest extends out into the larger graph. It is guaranteed to create path-preserving hierarchies, so that the abstraction shown is meaningful with respect to the underlying structure of the graph. The system works well on graphs of hundreds of thousands of nodes and millions of edges. TugGraph is able to present views of this proximal information in the context of the entire graph in seconds, and does not require a layout of the full graph as input.

Index Terms—Graph visualization, proximity, graph hierarchies.

1 Introduction

Many systems engineered to explore and create graph hierarchies search for subgraphs globally in the input graph as a basis for hierarchy construction. These approaches search for topological features [2], [5] or features based on attribute data [23], [6] associated with the nodes and edges. The graph hierarchy is recursively constructed by globally searching for subgraphs fitting the desired criteria and is thus suited for overviews of the graph structure. In a computer networking scenario, one could ask: *What is the high-level, topological structure of the Internet?* or *How do servers known to be in France connect to the Internet?*

However, these approaches do not have provisions for browsing parts of the graph topologically near a node or a subgraph. As an analogy, consider a library. Global approaches would be able to find all books on a given topic by keyword search, but frequently there are relevant books which are discovered by browsing the shelf near books on the topic. In our computer networking scenario, this activity corresponds to browsing nodes near a node or subgraph: *What is the topological structure near servers known to be in France?* We term this notion **proximity** in this paper, and in TugGraph, we exploit graph hierarchies to browse proximity to a node or subgraph of interest.

- *D. Archambault is with the INRIA Bordeaux Sud-Ouest, Universite de Bordeaux I, 351 cours de la Liberation, Bat A30, 33405 Talence, France. E-mail: daniel.archambault@inria.fr.*
- *T. Munzner is with the UBC Department of Computer Science, University of British Columbia, ICICS/CS Building, 201-2366 Main Mall, Vancouver, B.C. V6T 1Z4. E-mail: tmm@cs.ubc.ca.*
- *D. Auber is with the LaBRI, Universite de Bordeaux I, 351 cours de la Liberation, Bat A30, 33405 Talence, France. E-mail: auber@labri.fr.*

Manuscript received 1 July 2009; revised 15 Sept. 2009; accepted 27 Oct. 2009; published online 7 Apr. 2010.
Recommended for acceptance by H.W. Shen.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2009-07-0134.
Digital Object Identifier no. 10.1109/TVCG.2010.60.

A **graph hierarchy** or **hierarchy** is defined as a recursive grouping placed on the nodes in the input graph. For example, in a computer networking scenario where nodes are servers and edges are connections between servers a hierarchy would recursively group servers into subnetworks, networks, and finally the Internet. **Metanodes** are the interior nodes of this hierarchy which contain a subgraph. In our networking example, metanodes are nodes representing the subnetworks and networks. The **leaves** in the hierarchy are the nodes of the input graph. In our computer networking example, these are the servers. TugGraph creates metanodes that contain elements of the underlying graph that are directly connected to the subgraph or node of interest by an edge. We will call these subgraphs **proximal components**.

Interactive systems use hierarchy cuts to present meaningful abstractions of the input graph. A **hierarchy cut** defines which metanodes and leaves will be shown in the drawing of the graph. In the graph drawing literature, a hierarchy cut is frequently called an antichain. **Cut nodes** are nodes that appear on the hierarchy cut. These nodes are drawn opaquely in the abstract view. Nodes above the hierarchy cut are transparent and display the structure of the hierarchy using containment while nodes below the hierarchy cut are hidden from view. By manipulating the hierarchy cut, users can control which parts of the graph are abstracted away. In TugGraph, we tug out nodes adjacent to a selection, as shown in Fig. 1a and place their connected components into metanodes as shown in Fig. 1b.

Usually, the subgraph of interest is small compared to the size of the entire graph. In our networking example, the network at UBC is contained in a small number of metanodes compared to the rest of the Internet. Drawing the large metanodes is difficult as they contain hundreds of thousands of nodes and edges. Many coarsening techniques exist to handle this case [2], [6], but these techniques do not consider elements near a node in the hierarchy. Our metaphor is to tug out nodes adjacent to UBC from the larger Internet

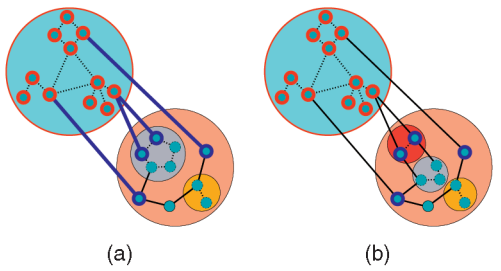

Fig. 1. TugGraph selection and decomposition. Structure within normally-opaque cut nodes shown with dotted lines. (a) Selection of the light blue metanode propagates to the hierarchy leaves, which are all beneath the current cut. (b) The hierarchy is modified by *tugging* out adjacent leaves, bringing them to the top level, and placing their connected components into metanodes.

components, and the process can be repeated to summarize paths. The challenge is the efficient computation of this summary in a way that if a path exists in the hierarchy cut, the path exists in the input graph. Not all graph hierarchies guarantee this property, and we define the space of these **path-preserving hierarchies** in the next section.

The TugGraph system, originally presented at *IEEE PacificVis* [7], introduced the TugGraph technique, and algorithms implementing the technique, for exploring a region of the graph located near a feature. In this paper, we present TugGraph as described in the conference proceedings and extend the work in two ways. First of all, we present an improved algorithm and system implementation that speeds up the TugGraph approach by modifying, rather than completely deleting and reconstructing, the hierarchy each time a node or metanode is tugged. Secondly, we re-evaluate the system on the same three data sets, plus an additional data set, to show how these algorithm modifications improve the execution speed of TugGraph. The new algorithm performs two to four times faster than the previous approach on the same machine.

2 Path-Preserving Hierarchies

A **path-preserving hierarchy**[1] [6], shown in Fig. 2, is a specific type of graph hierarchy that 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 l_1 and l_2 such that l_1 is a descendant of m_1 and l_2 is a descendant of m_2.
2. **Connectivity Conservation:** Any subgraph contained inside a metanode must be connected.

Hierarchies that ensure both of these properties guarantee that paths in the hierarchy cut also exist in the underlying input graph. Edge conservation guarantees that all edges in the hierarchy cut are witnessed by at least one edge in the input graph as shown in Fig. 2a. Connectivity conservation guarantees that there exists a path in the original graph through the metanode on the hierarchy cut as shown in Fig. 2b. Both edge and connectivity conservation are required in order to visualize paths and proximity information.

1. In the GrouseFlocks paper, the term used was topologically preserving hierarchy. Subsequently, we found path-preserving hierarchy a better term for this concept.

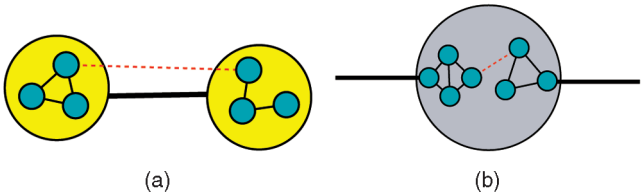

Fig. 2. Edge conservation and connectivity conservation are required to preserve paths in hierarchy cuts. (a) Edge conservation ensures that edges exist between metanodes of a hierarchy cut if and only if there exists one or more edges between descendants of the metanodes. (b) Connectivity conservation ensures that paths exist through metanodes. If the red dashed edge did not exist, there would not be a path from the bold metaedge on the left through to the bold metaedge on the right.

If hierarchies do not respect the path-preserving properties, paths that exist in the underlying graph may not appear in the view provided by the graph hierarchy. Also, paths that do not exist in the underlying graph may appear to exist in the graph hierarchy view. A complete presentation of these properties is presented in GrouseFlocks [6], but we present an example here. Fig. 3 shows a cycle at the top level of the hierarchy. If the red edge does not exist, connectivity conservation is not respected. As a result, this top level cycle illustrated by the graph hierarchy does not really exist in the input graph. Similarly, if edge conservation is not respected, we are unsure if edges between metanodes in the hierarchy actually exist. By enforcing these constraints, TugGraph ensures paths extending out from the node or subgraph of interest exist in the graph.

3 Previous and Related Work

As TugGraph uses GrouseFlocks [6] and graph hierarchies to explore proximity to a node or subgraph, we present previous work on graph hierarchy exploration in Section 3.1. We also present some techniques for extracting subgraphs proximal to nodes in a larger graph in Section 3.2 and highlighting techniques for subgraphs in the context of a large graph in Section 3.3.

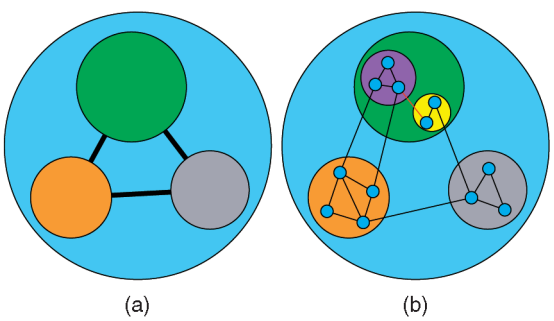

Fig. 3. The importance of edge and connectivity conservation in understanding paths when using a graph hierarchy. A more complete version is presented in GrouseFlocks [6]. (a) Top level of graph hierarchy consisting of three metanodes with a cycle. (b) Underlying graph and graph hierarchy. If the red edge does not exist, there appears to be a cycle at the top level of the hierarchy when no such cycle exists in the underlying graph. If edge conservation is not respected, we are unsure if metaedges on the cut really exist. If connectivity conservation is not respected, as in this example, we are unsure if paths in the underlying graph actually exist.

3.1 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 which can be interactively modified to display metanodes and leaves. In this section, we present systems which use this technique to explore large graphs.

3.1.1 Existing Layout Required

These systems exploit properties of a precomputed layout 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], multifocal fisheye approaches where metanodes are expanded and viewed in the context of the entire graph [25], topological fisheyes where abstract versions of the graph are presented far away from a focus center [17], linking the graph hierarchy to a separate treemap view [1], interactively visualizing hierarchies of small world clusterings [27], and visualizing complex software in three dimensions using level of detail techniques [10].

All of these techniques use a static layout that is computed once up front, and a static hierarchy computed using the position of the vertices in the drawing. Exploiting this static layout has the advantage of quick and fluid interaction. However, computing this layout for a large graph can be computationally expensive. Also, elements near to each other, in a graph theoretic sense, may be quite distant in the precomputed layout, as a full drawing cannot always map graph theoretic and euclidean distance well. TugGraph computes the layout, like other steerable systems, on the fly. Steerability allows the layout computation to take these focus centres into account, allowing for a more compact presentation of paths and proximity.

3.1.2 Steerable Exploration

Steerable systems compute the graph layout dynamically during exploration and do not require a pre-existing layout. Therefore, they can more readily be adapted for exploring paths where the source and destination nodes are not known in advance.

Steerable systems have been developed to visualize search engine query results [12] and graph hierarchies formed by detecting topological features [2], [5]. These systems do not support hierarchy editing, which allows users to customize their graph hierarchies. Steerable hierarchy editing is required in order to create the proximal components as described in the introduction.

3.1.3 Steerable Hierarchy Editing

Several systems have been developed to edit graph hierarchies using topological or attribute information. These systems are directed toward exploring the global structure of the graph and the topological or attribute features present in it.

The DA-TU system [14] of Huang and Eades is a force-directed approach which biases the hierarchy cut toward its hierarchy structure; Auber and Jourdan [9] support interactive hierarchy editing; and the Clovis system [23] supports interactive clustering of an input graph based on querying the attribute values associated with the nodes and edges of the graph. GrouseFlocks [6], the system on which TugGraph is based, allows for the interactive exploration and creation of graph hierarchies based on attribute data. The system uses Reform-Below-Cut operations which divide based on attribute data associated with each node. GrouseFlocks resorts to coarsening with global topological feature detection and edge contraction when the hierarchy cut is too large to be explored interactively.

However, none of these systems have been adapted to browse proximity in the original graph beyond manual selection. In TugGraph, we develop a system that modifies an existing hierarchy to better illustrate proximity information in the underlying graph.

3.2 Other Notions of Proximity

Many other works, primarily in the data mining literature, focus on good formalisms for proximity to elements of a graph [15], [16], [18], [19]. These approaches provide algorithms to find the nodes which exist between or around a node or subgraph of interest in a large graph. The smaller subgraph can be extracted and drawn for the purposes of visualization. However, the context of how this subgraph connects with the rest of the network is lost and the work does not focus on interactive techniques. Although we use a simpler notion of proximity in this work, direct adjacency, these techniques could be adapted to allow TugGraph to display these forms of adjacency. TugGraph could be adapted to handle these notions of adjacency by substituting the adjacency computation with the decomposition computed by one of these methods.

3.3 Subgraph Highlighting in Graphs

Selection and other techniques that exploit preattentive channels such as motion [28] or color using hover queries [22] can be used to highlight parts of a graph including paths. In contrast, Boutin et al. [11] support filtering using a graph hierarchy based on an interactive choice of focus node but aimed at showing a global overview rather than local structure.

These techniques are very effective but have two drawbacks. First, a significant portion of the entire of the graph must be drawn before visualization can begin. Second, large amounts of visual clutter are still a barrier to comprehension of the set of nodes in the context of the entire graph and the approaches cannot exploit spatial position to emphasize proximity. However, these techniques may be used to better emphasize paths and proximity.

3.4 Subgraph Highlighting by Dynamic Layout

There exist a number of techniques which modify an existing layout of the graph to show local connectivity around a node. If the user performs a number of these operations in sequence, these techniques can help facilitate the navigation of paths in a graph.

The Bring Neighbors Lens [26] is a lens which adjusts the layout of a graph, bringing nodes directly adjacent to a focus node spatially close. Although the technique was not designed for path navigation, it does allow for the visualization of nodes nearby a graph element. Bring & Go [21] is an interactive technique following a particular path in a graph. When a node in the graph is selected, the layout is adjusted so that the nodes directly adjacent to the focus node are

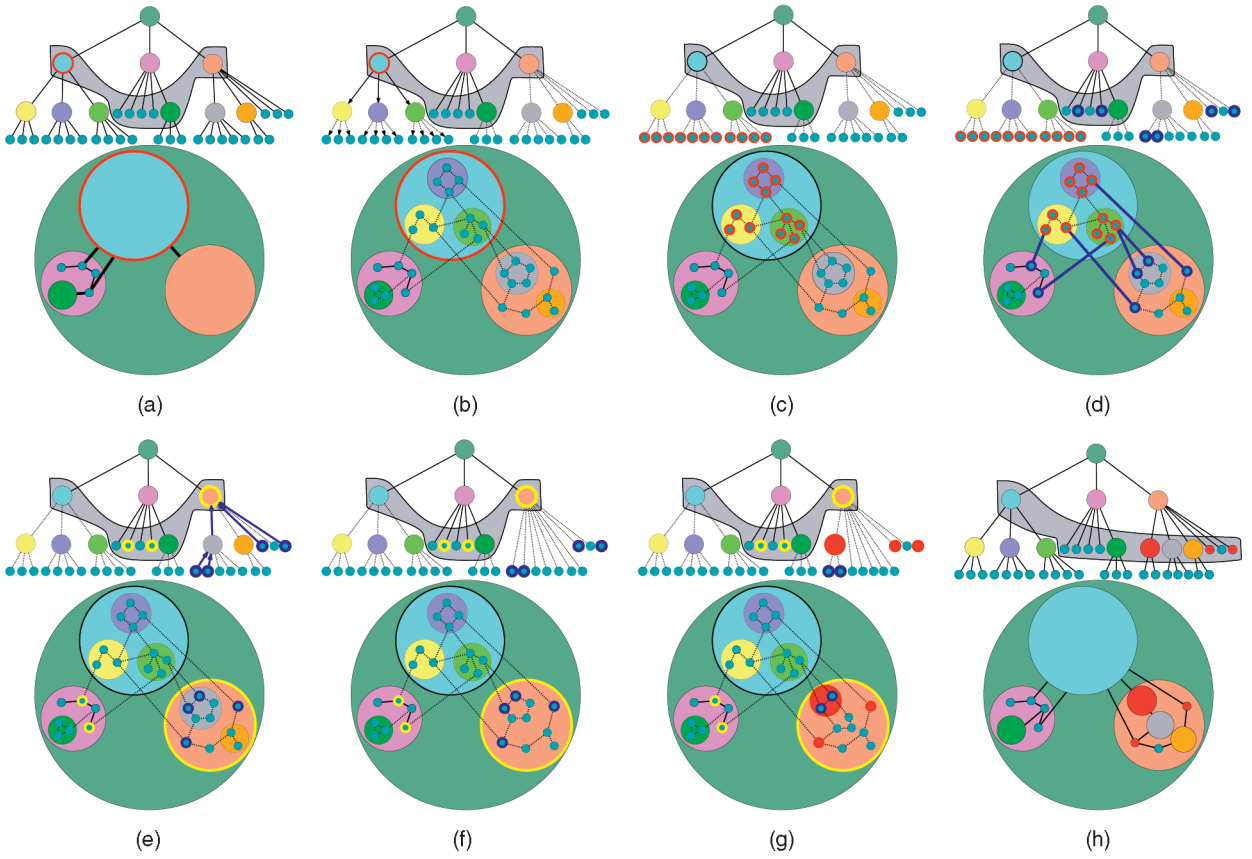

Fig. 4. All steps of the TugGraph algorithm with hierarchy above and graph below. (a) The input to the algorithm with the node the user has clicked on outlined in red. (b) The same selection is shown, but with the full graph visible to the leaves. Dashed lines are used for the parts of the graph hierarchy below the hierarchy cut. (c) Elements of the source set S are outlined in red. (d) Each element of the proximal set P is outlined in blue. The edges that were used to create the proximal set are highlighted in blue as well. (e) The proximal cut set C is outlined in yellow. The hierarchy edges used to compute the proximal cut set are in blue in the hierarchy view. (f) The hierarchy is destroyed below each proximal cut element. (g) The proximal components are computed. These components are outlined in red in the figure. (h) The final hierarchy is presented.

brought in close to it. Subsequently, an adjacent node can be selected and the view is smoothly animated with zoom-in and out for context. McGuffin and Jurisica [20] present several subgraph selection techniques. Their *spread neighbors* technique interactively modifies the layout by placing nodes at increasing distance from a focus node on concentric rings around it. This selection technique is probably the closest of the above-described methods to TugGraph.

All three of these techniques have the advantage that the full layout of the graph is computed once and upfront. Thus, all modifications to the layout can be computed interactively. However, the users are required to wait for a full layout of the graph to be computed beforehand. Also, as the graph drawing algorithm does not know which paths will be explored at the time of the layout, nodes on the path being explored may be spatially distant, making simplification of the graph that does not participate in the paths more difficult. Therefore, one could say that TugGraph makes a trade off where perinteraction cost is more expensive then the above techniques, but we have the benefits of more flexibility in graph layout and in abstracting away of parts of the graph that do not participate in the paths being explored.

4 Algorithm

In this section, we present the original TugGraph algorithm [7], and then the modified algorithm and system improvements that accelerate the execution of the algorithm by a factor of about two to four.

In both versions of the algorithm, TugGraph takes a graph and a hierarchy as input. If multiple connected components exist, each component is stored inside its own metanode at the root of the hierarchy drawn with component packing. The user then clicks on a node in the hierarchy cut to obtain proximity information about it. This **source node** is a node of the input graph or a metanode of the hierarchy that is tugged. A tug on this source node selects all adjacent leaf nodes and places them into metanodes directly below metanodes on the cut. The hierarchy is modified accordingly, ensuring that it is still path-preserving. The cut is then lowered, revealing the components adjacent to the source node.

4.1 Tugging with Hierarchy Reconstruction

The input to the original TugGraph algorithm, as presented at *PacificVis*, is shown in Fig. 4a. On this input, the algorithm operates in five steps:

Step	Complexity
Computing Source Set	$O(\|M_S\|+\|S\|)$
Computing Proximal Set	$O(D_S)$
Computing Proximal Cut Set	$O(\|P\|+\|M_P\|)$
Computing Proximal Components	$O(\|P\|)$
Original Reconstruction	$O(\|M_P\|+\|E_P\|+\|P\|+D_P)$
New Reconstruction	$O(\|M'_P\|\log\|M'_P\|+d\|E'_P\|)$

Fig. 5. Summary of asymptotic complexity of TugGraph steps. 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. Likewise, E_P is the number of metaedges. The maximum depth of the hierarchy is d. M'_P and E'_P are the sets of metanodes and metaedges in the hierarchy locally involved in a split event. New Reconstruction is presented later in Section 4.2.

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 (Fig. 4c).
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 (Fig. 4d).
3. Determine the set of cut metanodes that contain elements of the proximal set. This set is the **proximal cut set** denoted C (Fig. 4e).
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 (Figs. 4f and 4g).
5. Reconstruct the hierarchy for all other leaf nodes that are descendants of metanodes of the proximal cut set but not elements of the proximal set (Fig. 4h).

Fig. 4h shows the result of the five steps on a metanode selected on the graph hierarchy. When describing each step, the complexity of each step is presented. 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 and leaves that are moved above the hierarchy cut. A table showing the complexity of each step is shown in Fig. 5.

4.1.1 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 Fig. 4a, the algorithm traverses the graph hierarchy top down from the selected metanode to discover all leaf descendants as shown in Fig. 4b. These leaves are the source set S, outlined in red in Fig. 4c.

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 node. Then, this traversal takes $|M_S|+|S|$ time as each leaf and metanode is scanned exactly once.

4.1.2 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, let E_u be the set of edges adjacent to a vertex $u \in S$; the proximal set P is defined as follows:

$$P = \{v|(u,v) \in E_u, u \in S, v \notin S\}. \quad (1)$$

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 (1). The result of this part of the algorithm is shown in Fig. 4d.

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

4.1.3 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. Fig. 4e 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 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 step is $O(|P|+|M_P|)$.

4.1.4 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.

Fig. 4e shows the input to this step 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.

In Fig. 4f, 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 element of the proximal cut set 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 GrouseFlocks [6].

The algorithm then computes the proximal components as shown in Fig. 4g. 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

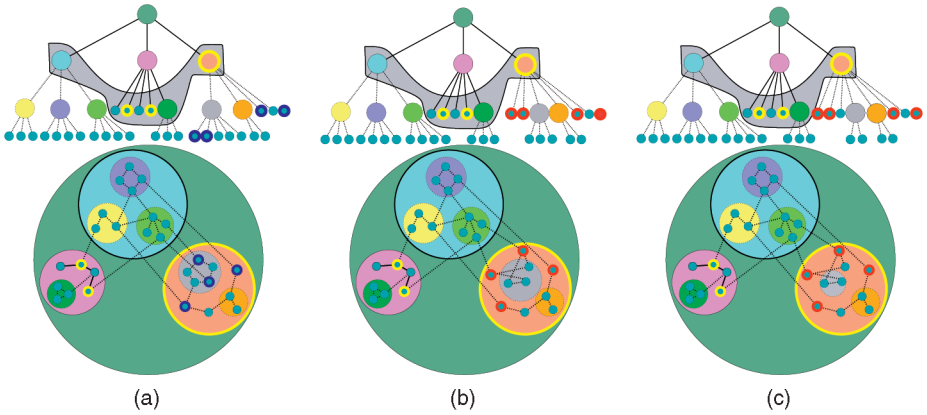

Fig. 6. New steps for hierarchy modification. (a) The input is the proximal set as in the previous version of the algorithm. (b) Proximal components are tugged out and edges are deleted. The gray metanode is now disconnected. (c) Gray metanode is split in two to preserve connectivity conservation. Edge deletions followed by splits are propagated up the hierarchy until the cut metanodes are reached.

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 $O(|P|)$ proximal components are created.

4.1.5 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 GrouseFlocks [6], where the components are divided into sets defined by the previous hierarchy. 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 step involves reconstructing the hierarchy above the proximal set. Let this hierarchy be the set of M_P metanodes and E_P the set of hierarchy edges above nodes in the proximal set. 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 $O(|M_P| + |E_P| + |P| + D_P)$.

4.1.6 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 metanodes must contain at least two nodes. In worst case, a tug can take $O(|E|)$ time. This worst case is realized when the sum 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 is $O(|N|)$.

4.2 Tugging with Hierarchy Modification

We now describe the algorithmic and implementation changes that improve the execution speed of TugGraph over its original implementation.

4.2.1 TugGraph Algorithm Improvements

Instead of destroying the graph hierarchy and reconstructing it in its entirety around tugged nodes, we present a new algorithm that replaces the three steps presented in Figs. 4f, 4g, 4h and discussed in Sections 4.1.4 and 4.1.5 to only modify the parts of the graph hierarchy affected by a tug. Fig. 6 shows the execution of this algorithm on a simple two level hierarchy.

In order to modify these hierarchy components, we place each metanode containing at least one node of the proximal set into a priority queue with the priority of each node of the hierarchy equal to its depth. At each step in the algorithm, the top element of the priority queue is removed and the subgraph it contains is processed in two ways: disconnected metanodes one level down are split and adjacent edges to a proximal node are removed. Thus, once a step of the algorithm is completed any metanode below the current level is guaranteed to be connected.

Before taking the first element off the priority queue, all the nodes of the proximal set and all the edges linking two nodes of this set are moved inside metanodes on the hierarchy cut that contain them. When constructing the proximal cut set, the algorithm additionally stores the nodes one level below the proximal cut set for nodes in and adjacent to the proximal set. Using this new mapping, we can directly connect proximal set nodes to their corresponding metanodes at this level of the hierarchy. Subsequently, we remove all proximal component nodes and their adjacent edges from the lower levels of the hierarchy, possibly disconnecting elements at this deepest level. Pseudocode for the initialization of the priority queue and its processing is provided in Fig. 7.

$q \leftarrow \{p | p = v.\text{parent}, v \in N, v \in P\}$
for all $v \in P$ **do**
  - move each v to the graph contained by its cut metanode.
  - connect second last metanode and v and other $v \in P$.

$\forall v$ flag$[v] \leftarrow$ **false**
while $q \neq \emptyset$ **do**
  $v \leftarrow q$.pop ()
  if !flag$[v]$ **then**
    flag$[v]$ $\leftarrow$ **true**
    filterEdges (**metagraph** (v))
    splitMetaNodes (**metagraph** (v))
    if v.parent $\notin C$ **then**
      q.push (**depth** (v.parent), v.parent)

Fig. 7. Algorithm to populate q with the elements of P. The first part of the algorithm loads q with all metanodes that are direct parents of leaf nodes that are elements of P. Let $G = (N, E)$ be the input graph, and let $g.N$ and $g.E$ be the node and edge sets of graph g, respectively. The second part of the algorithm processes the priority queue, removing edges and splitting metanodes below the element taken off the queue if it has not been processed previously. In this pseudocode, **metagraph** returns the graph contained by the passed metanode and **depth** returns the depth of the node in the graph hierarchy. The functions **filterEdges** and **splitMetaNodes** are described with pseudocode in Figs. 9 and 11, respectively.

At this point, the first node is taken off the priority queue and processed. The parent of the metanode is placed onto the priority queue if it is on or below the hierarchy cut as edges incident to a proximal set node could be present at any level. When a metanode is taken off the priority queue, the algorithm starts by removing any edges incident to a node of the proximal set because these edges have moved up to the level just below the hierarchy cut. Then, the algorithm checks all metanodes below the current metanode to see if they are connected. If any metanode is not connected, it is split into several components.

Fig. 6b shows how edges incident to elements of the proximal set are removed from the hierarchy, possibly disconnecting the graph. As each metaedge has a list of edges in the input graph it represents, this list is scanned once and any metaedge connected to a proximal set node is removed from the list as shown in Fig. 8. If the list of input graph edges is empty after a metaedge has been processed, the metaedge no longer subtends an input graph edge and therefore is deleted. Otherwise, the new list of input graph

filterEdges (Graph *g)
  for all $e \in \{(u,v) | (u,v) \in g.E,$ flag$[u]$ **or** flag$[v]\}$ **do**
    $OE \leftarrow$ **originalEdges**(e)
    $NE \leftarrow \{(u,v) | (u,v) \in OE, u \notin P, v \notin P\}$
    if $NE = \emptyset$ **then**
      delete(e)
    else
      setOriginalEdges(e, NE)

Fig. 9. Pseudocode to filter input graph edge lists associated with a metaedge and remove metaedges no longer subtended by an edge of the input graph. The function **originalEdges** returns the set of input graph edges associated with the metaedge and **setOriginalEdges** sets this list of edges. The function **delete** deletes an edge from the graph. Fig. 8 shows this process graphically.

edges is assigned to the metaedge and stored with it. Pseudocode for this step of the algorithm is provided in Fig. 9.

This pass of the algorithm could possibly disconnect the subgraph contained in this metanode. For example, Fig. 8 shows three connected metanodes as input, but the yellow metanode will be disconnected after the sets of original edges have been processed as indicated by the dashed metaedge. However, in this case, when its parent is processed, the metanode will be split into several metanodes that are then reconnected, as we now describe.

Fig. 6c shows how metanodes in the graph hierarchy one level below the current metanode are split into two or more metanodes, if they contain several connected components. A connected components test is run to determine if the subgraph is connected. If the subgraph is not connected, each connected component is assigned its own separate metanode at the current metanode level as shown in Fig. 10. The edge lists associated with each metaedge connected to the metanode being split are scanned to determine to which new metanode they should be attached. This scan involves tracing the input edge of the graph up to the current level of the hierarchy. If the hierarchy has depth d and the set of original graph edges of this type is E'_P, then this can cost $O(d|E'_P|)$. Pseudocode for this step is provided in Fig. 11.

The worst-case complexity of the new hierarchy modification scheme is $O(|M'_P| \log |M'_P| + d|E'_P|)$, where d is the depth of the hierarchy and $|M'_P|$ and $|E'_P|$ are the sets of metanodes and metaedges affected directly by the tug. The additional

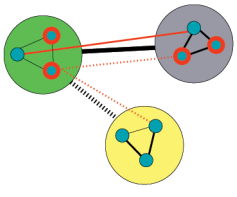

Fig. 8. Filtering metaedges. Edges in the original graph that are represented by a metaedge at this level are scanned. If at least one incident node of an edge is a part of the proximal set, the nodes highlighted in red, the edge removed from the list of edges associated with the metaedge. If a metaedge has an empty list at the end of this process, it is removed. Solid black metaedges at this level are kept while dashed black metaedges are removed. Red dashed edges are adjacent to an element of the proximal set while red solid edges still subtend a metaedge at this level.

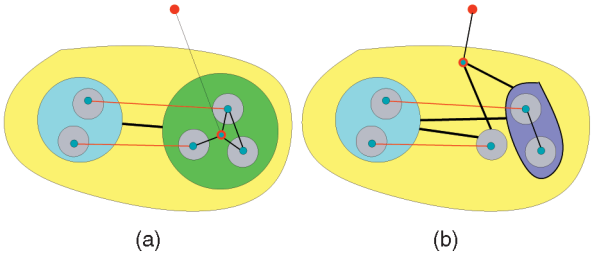

Fig. 10. A split event before and after an element of the proximal set, the blue node highlighted in red, is removed. (a) Before the node is removed. The red node at the top of the diagram is tugged. (b) After the node is removed. The green metanode is destroyed and a blue metanode is created after the split. Red edges are edges in the original graph that must be redistributed among the two newly created metaedges. This set of edges over all metanodes affected by the tug in the hierarchy is the E'_P in the complexity analysis. For diagram clarity, only relevant edges and nodes are shown.

```
splitMetaNodes (Graph *g)
  for all v ∈ {v|v ∈ g.N, flag [v]} do
    g_v ← metagraph(v)
    if !isConnected (g_v) then
      M_v ← conCompDecomp (g_v)
      g.N ← g.N ⋃ M_v
      for all e incident to v do
        OE ← originalEdges(e)
        for all e_o ∈ OE do
          e_m ← findM_v(e_o)
          g.E ← g.E ⋃ {e_m}
          setOriginalEdges(e, OE ⋃ {e_o})
      delete (v)
```

Fig. 11. Pseudocode to split metanodes in the graph g if they are disconnected and redistribute input graph edges incident to the split metanode v to newly created metaedges by the split. In this pseudocode, **isConnected** returns true if the passed graph is connected and **conCompDecomp** computes a connected component decomposition of the passed graph and places each component inside its own metanode, returning a set of metanodes M_v as a result. The function $\mathbf{findM_v}$ traces the two nodes incident to e_o through the hierarchy to this level of the hierarchy, returning the metaedge e_m that subtends it. Fig. 10 shows this process graphically.

factor of $O(\log |M'_P|)$ is incurred because our reconstruction is no longer global and processes metanodes one at a time. This logarithmic factor is not important as long as the number of metanodes affected by a tug is smaller than the total number of nodes above the proximal set or more precisely when $|M'_P| \log |M'_P| < |M_P|$. As an example, the old algorithm would perform in $O(|M_P| + |E_P|)$ time and the new algorithm would perform in $O(|M_P| \log |M_P| + d|E_P|)$ time if every node above the proximal set needs to be recomputed or when $M'_P = M_P$ and likewise $E'_P = E_P$. However, the usual case is that the number of metanodes to examine is much smaller than the total number of nodes above the proximal set, and in practice, better performance is realized.

4.2.2 TugGraph System Improvements

In this section, we describe two architectural improvements to the TugGraph system that made a difference in the execution speed of the algorithm.

In the Tulip graph drawing library [8] used heavily by the TugGraph system, metanodes are quite frequently rendered as transparent entities that allow for all levels of a graph hierarchy to be seen at once. Therefore, when any element of the hierarchy is modified, the properties of these elements need to be updated. As TugGraph renders metanodes on the hierarchy cut opaquely, we do not need to update the properties of nodes and metanodes below the cut. To increase the efficiency of our implementation, we disabled several linear scans that updated these properties. This change led to a significant speed up in the algorithm.

Second, the contents of widgets in the interface were completely destroyed and rebuilt in the previous implementation. For example, there is a list view, similar to a Windows Explorer file drop-down view, that displays the contents of the hierarchy. As the algorithm no longer needs to completely destroy the hierarchy below affected metanodes, widgets, such as these, can be updated rather than completely reconstructed. In many cases, this implementation change, made possible by the faster algorithm, offers improved performance.

5 Coloring 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 GrouseFlocks. In order to distinguish multiple tugs, the system rotates through the colors: purple, tan, blue, green, and light blue. Proximal components are a more saturated version of the color while all other components are less saturated. Open metanodes are bounded in a background disk of the same color.

We observed on our data that TugGraph tends to produce many small components and a few large components when operating on a graph. These results may be due to the small world nature of our data sets. 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. Thus, TugGraph can present nodes at a logscale node size. When logscale is used, it is explicitly indicated in the text.

6 Results

In this section, we present the performance of our implementations of TugGraph on several data sets. Section 6.1 presents the original TugGraph results as presented in the conference article. Section 6.2 presents the results of the previous implementation of TugGraph against the improved algorithm on the same machine.

6.1 Previous TugGraph Performance

TugGraph is implemented using the Tulip graph drawing libraries [8] and GrouseFlocks [6]. We compare TugGraph to existing systems on three data sets.

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

The `Net05` data set [4] shows the structure of the Internet backbone routers as generated in 2005 by Cheswick's Internet Mapping Project.[2] Nodes in this graph are servers and an edge exists if two servers exchanged packets. Each node has its server name and its IP address as attributes. It has 190,384 nodes and 228,354 edges.

The `Actors` data set is an IMDB subset centred 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 data set has 38,997 nodes and 1,948,712 edges.

For each data set, a result is presented using TugGraph and the result or part of the result is highlighted in the

2. www.cheswick.com/ches/map.

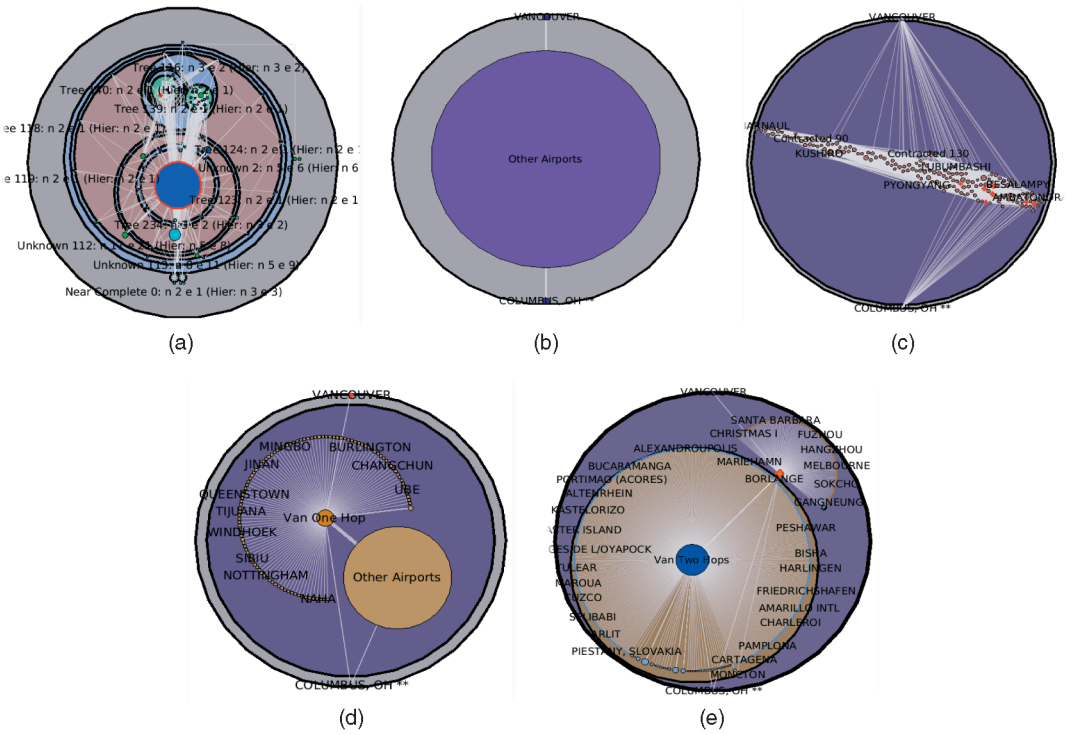

Fig. 12. Browsing paths between Vancouver and Columbus: (a) Grouse, (b) initial GrouseFlocks decomposition, (c) GrouseFlocks coarsening, and (d)-(e) TugGraph. In Grouse, metanodes of the hierarchy are colored using the type of topological feature they contain. (b) In GrouseFlocks and TugGraph, Vancouver and Columbus are colored in saturated purple while the large component in desaturated purple is the remaining airports. (c) The coarsened metanodes are brown, and the ones containing airports one hop from Vancouver are outlined in red. (d) The saturated tan `Van One Hop` contains airports one hop from Vancouver. (e) `Van One Hop` is outlined in red, and `Van Two Hops` contains airports two hops from Vancouver. `Van Two Hops` is in saturated blue.

remaining systems. As TugGraph supports label editing, we manually rename proximal component metanodes created during exploration to have meaningful names. When a TugGraph operation is executed, the metanode that was tugged to generate the image is outlined in red. Proximal components are always presented in saturated colors. We used a 3.0 GHz Pentium IV with 3.0 GB of memory running SuSE Linux with a 2.6.5-7.151 kernel.

6.1.1 Airport

We browse how the flight paths between Columbus and Vancouver are interconnected in `Airport`. Fig. 12 shows the results for `Airport` under Grouse, GrouseFlocks, and TugGraph. In Grouse, the decomposition into topological features neither takes advantage of the attribute information nor the proximity information. Fig. 12a shows that even finding the airports one hop away is buried very deep in a hierarchy of topological features. Fig. 12b 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 Fig. 12c. The solution improves on that of Grouse, but the airports one hop away from Vancouver are still scattered over the hierarchy.

Figs. 12d and 12e present the results using TugGraph. The process starts with same initial decomposition shown in Fig. 12b. First, the source node `Vancouver` is tugged, extracting the airports one hop away from it. In Fig. 12d, 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 airports connected to airports 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`. One stop-over flights exist to Columbus, since both `Vancouver` and `Columbus` are connected to `Van One Hop`. However, there is no direct flight since the two airports are not connected by an edge. Fig. 12e 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.

6.1.2 Net05

We browse the structure around the `*.net.ubc.ca` portions of the network in `Net05`. After 12 hours of computation, Grouse had not finished computing a hierarchy of topological features, so we do not show it here. Instead, drawings produced by LGL [3] and SPF [4] are included as these algorithms work well on this type of data. We use logscale size nodes on this data set.

The results are shown in Fig. 13. Once again, the problem is solved using TugGraph, and we use highlighting to show

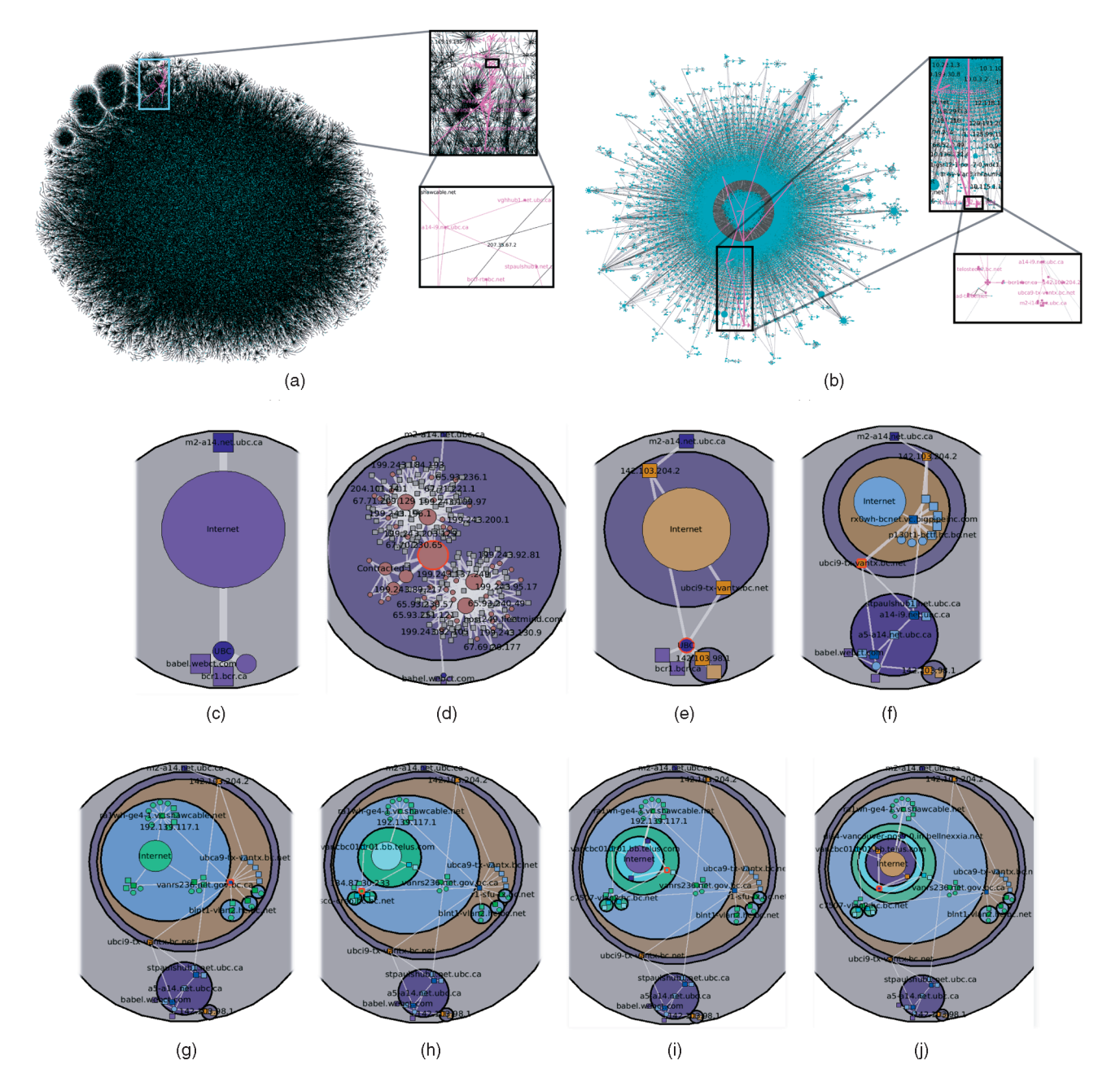

Fig. 13. Exploration of the Net05 data set using: (a) LGL, (b) SPF, (c) initial GrouseFlocks decomposition, (d) GrouseFlocks coarsening, and (e)-(j) TugGraph. In the LGL and SPF drawings, UBC servers and those four hops away are highlighted red. GrouseFlocks shows a good initial decomposition but is unable to go further since the attribute information on this data set is minimal. TugGraph, however, shows how UBC connects to the Internet. (a) LGL, (b) SPF, (c) GrouseFlocks Decomposition, (d) GrouseFlocks Coarsen, (e) Tug UBC, (f) Tug ubci9, (g) Tug rx0wh, (h) Tug c7507, (i) Tug 69.156.254.254, and (j) Tug bellnexia.

the solution in other approaches. Figs. 13a and 13b show where the UBC servers are in the data set and highlight the portions that are four hops away. In these figures, all nodes and edges of the data set 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 Fig. 13c, 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 data set, when the huge `Internet` metanode is expanded it is too complex to draw in full and must be coarsened, as shown in Fig. 13d. The servers four hops away are all inside the single large metanode outlined in red. The GrouseFlocks solution is more suitable for this task, but browsing the connections between UBC and the rest of the Internet is difficult.

Figs. 13e, 13f, 13g, 13h, 13i, and 13j 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 Fig. 13c. First, the `UBC` metanode is tugged, revealing that the two parts of the UBC

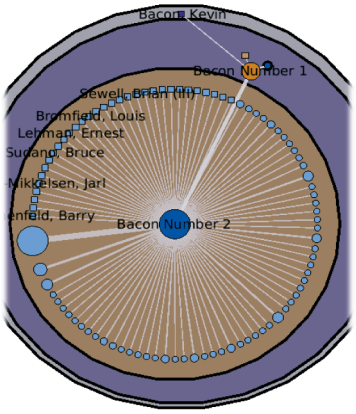

Fig. 14. Bacon number trends in `Actors`. `Bacon Number 1` contains all actors with a Bacon number of one. The saturated blue node to its right and `Bacon Number 2` contain actors with a Bacon number of 2. We see that actors with Bacon numbers of one or two tend to act with each other as their components are connected.

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. `ubci9-tx-vantx.bc.net` is tugged and the result is shown in Fig. 13f. `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 Figs. 13g, 13h, 13i, 13j by tugging on the nodes named in the captions and outlined in red in each of these figures.

6.1.3 Actors

On `Actors`, we demonstrate that we can generate an overview of Bacon numbers for any movie released between 1998 and 2001. 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, 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 minimum Bacon number of the actor. Grouse was unable to generate a hierarchy of topological features in over 12 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 Fig. 14. For this data set, logscale node sizes are used.

Fig. 14 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 a 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.

6.1.4 Timings

This section presents timing numbers for the results section.

On `Airport`, to compute the hierarchy of topological 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 2 seconds to complete.

To draw the `Net05` data set 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 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`.

6.2 New TugGraph Performance

We ran the three data sets again and measured the performance of the previous version of TugGraph with the new version. The new performance numbers are presented in Fig. 15. As the images produced by the new algorithm on the previous data sets were the same, we do not present these images. In this second set of results, both the old version and the new version of TugGraph were run on a 2.16 GHz dual core Pentium IV with 2.0 GB of memory, running Fedora Core 8 with a 2.6.26.8-57 kernel.

Our running time numbers were similar to those presented in the *PacificVis* article for the original version

Data set	|N|	|E|	Reconstruction (sec.)		Modification (sec.)		× faster
			Hier.	Total	Hier.	Total	
`Airport`	1,540	16,523	$<<$ 0.01	1.05	$<<$ 0.01	0.53	2.0
`Net05`	190,384	228,354	7	32	6	8	4.0
`Actors`	38,997	1,948,712	26	114	17	56	2.0
`Airport Deep`	1,540	16,523	1	21	$<<$ 0.01	13	1.6

Fig. 15. Table containing execution time of the old TugGraph algorithm and the new TugGraph algorithm on the three data sets used in the *PacificVis* article and a new data set. **Hier** is the time required in order to restructure/recompute the hierarchies. **Total** is the total time for the tug from start to finish. The value $\ll 0.01$ indicates that the time to compute this value was less than a hundredth of a second.

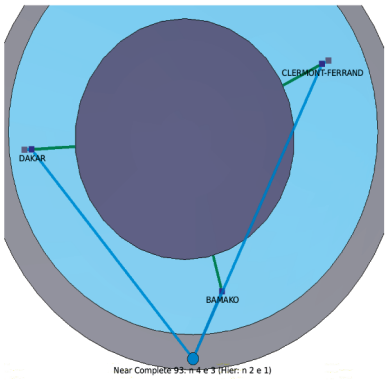

Fig. 16. A component in the first level of `Airport Deep` is tugged. The result pulls loosely connected components from near the bottom of the hierarchy to the top.

of TugGraph. Thus, we could say that, for the most part, we reproduced the results of the TugGraph algorithm on the slightly slower machine. In most cases, the new version of the TugGraph algorithm was two to four times faster than the previous version.

The largest improvement was seen in `Net05` where a factor of four speed up was obtained. The reason behind this improvement in performance is the relatively small size of the UBC network compared to the rest of the data set. Additionally, this network is loosely connected, and therefore, has few edges involved in the tug. In this case, relatively few metanodes needed to be constructed in the new version of the algorithm where nearly all of them were reconstructed in the previous version of the algorithm. The runtime improvement was mostly noted in the Total column. As such, it seems that both the algorithmic and implementation changes helped for this test case.

`Airport` and `Actors` had similar improvements in performance, both registering about a factor of two improvement. `Airport` may have seen only a factor of two improvement due to its small size. `Actors` most likely only improved by a factor of two because of its high connectivity. As the number of edges processed increases, the degree and edge terms in the complexity analysis approach $|E|$, the total number of edges in the graph, and dominate. This gives both algorithms a similar running time performance.

`Airport Deep` is a new data set for this set of algorithm test runs. A result of tugging a small top level component is shown in Fig. 16. The data set is the `Airport` clustered recursively using the strength metric. Recursive strength decomposition has been shown to be helpful for geographers when analyzing the structure of worldwide transportation networks [24]. The hierarchy above the proximal components has a depth of 20.

The small component group of cities at the top level interact with a few cities located at depths very close to 20 in the hierarchy. Although the test only shows an improvement of 1.6 times over the original TugGraph algorithm, notice the larger improvement in the execution time of the hierarchy decomposition. This improvement is primarily due to the new algorithm that modifies only nodes affected by the tug. However, it seems that further implementation improvements are needed to reduce the overall execution time of the algorithm.

7 Discussion

In the three example data sets, the dependence on the sizes of the source and proximal sets along with the sum of their degrees is apparent. The tests on `Net05` and `Actors` provides evidence that increasing average node degree affects the running time significantly.

TugGraph can be used to explore structure near a feature in the graph or the paths between two features. Although in two of the three scenarios the destination was known, we do not view knowing the destination as a requirement of the system. One could envision a use case that consists of iteratively tugging out structure from the larger graph starting from only a source node.

Another important observation is that the TugGraph result images do not require zoomed-in insets to show details of the graph structure. Typically in large graph visualization systems, many scales are needed to understand features in the data in a global context. Hierarchy-based visualization tools, including TugGraph, are able to represent the sought structure succinctly at a single scale.

Through algorithm execution time, we see that TugGraph is suitable for displaying the structure near a small set of nodes in a larger graph. TugGraph has a running time advantage over computing a hierarchy of topological features or computing a full layout with SPF or LGL. Also, the diagrams it produces are better suited for exploring connectivity near a feature, because elements proximal to the focus node are emphasized in the layout and less relevant portions of the graph are abstracted away.

8 Future Work

In future work, we believe that TugGraph can be refined in many ways to improve its performance well beyond what we have seen here. Although we have made significant advances toward the goal of interactive performance over the version of the paper presented at *IEEE PacificVis*, on data set sizes with millions of nodes and edges, we have not achieved this goal. We believe that further optimization of the software and potential algorithmic improvements may make this technique competitive with the algorithms that require a pre-existing layout. However, further research and software improvements are required to confirm this conjecture.

Frequently, the edges in a graph have weights. These weights can be used in conjunction with the topological structure of the graph to determine proximity. We hope to extend TugGraph to handle weighted graphs. It may also be interesting to allow TugGraph to display nodes of the graph that are multiple hops from a source node in a single click. However, this functionality needs to be implemented and tested to determine if it can be done efficiently and in a useful way for exploring graphs.

A graph hierarchy is able to help summarize the structure of a graph, because it groups a large number of nodes together with a common meaning. A limitation of TugGraph exists when a tug produces a large number of proximal

components that cannot be easily summarized in a path-preserving way. Situations such as these can result in a visually cluttered drawing. In future work, we should investigate path-preserving coarsening techniques which can effectively summarize a large number of proximal components in a simple drawing. Additionally, more compact representations for a large number of disconnected components remains future work.

The TugGraph metaphor relies on a sequential series of tugs to produce a graph hierarchy emanating from a single node or metanode chosen by the user. TugGraph can tug at a graph from multiple nodes or metanodes, but an interesting direction for future work would be to investigate browsing methods that are more inherently multifocus in terms of computational efficiency and visual representation.

Finally, user experimentation and studies with domain experts would be essential to validate the usability of the technique. Currently, we are investigating the readability of path-preserving hierarchies when performing certain tasks, which would help validate this work. Direct comparisons of TugGraph with other techniques which use a pre-existing layout may also be beneficial. Also, we hope to work with users in the computer networking domain, where this problem originally arose, to determine if the technique helps experts better understand their data.

9 Conclusion

We have presented an improved algorithm for our interaction metaphor where users can tease out nodes from a large graph by tugging on a feature in a path-preserving way, and we have tested the system on input graphs with hundreds of thousands of nodes and millions of edges. The new algorithm performs two to four times faster than the previous version of TugGraph published at *IEEE PacificVis*.

Acknowledgments

The first author (Daniel Archambault) would like to thank the InfoVis groups at UBC and INRIA Bordeaux Sud-Ouest as well as the reviewers of his thesis who helped improve this work further. Finally, the authors would like to thank the anonymous reviewers of *TVCG* for their comments.

References

[1] J. Abello, S.G. Kobourov, and R. Yusufov, "Visualizing Large Graphs with Compound-Fisheye Views and Treemaps," *Proc. Conf. Graph Drawing,* pp. 431-441, 2004.

[2] J. Abello, F. van Ham, and N. Krishnan, "ASK-GraphView: A Large Scale Graph Visualization System," *IEEE Trans. Visualization and Computer Graphics,* vol. 12, no. 5, pp. 669-676, Sept./Oct. 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," *J. Molecular Biology,* vol. 340, no. 1, pp. 179-190, June 2004.

[4] D. Archambault, T. Munzner, and D. Auber, "Smashing Peacocks Further: Drawing Quasi-Trees from Biconnected Components," *IEEE Trans. Visualization and Computer Graphics,* vol. 12, no. 5, pp. 813-820, Sept./Oct. 2006.

[5] D. Archambault, T. Munzner, and D. Auber, "Grouse: Feature-Based, Steerable Graph Hierarchy Exploration," *Proc. Eurographics/IEEE VGTC Symp. Visualization (EuroVis '07),* pp. 67-74, 2007.

[6] D. Archambault, T. Munzner, and D. Auber, "GrouseFlocks: Steerable Exploration of Graph Hierarchy Space," *IEEE Trans. Visualization and Computer Graphics,* vol. 14, no. 4, pp. 900-913, July/Aug. 2008.

[7] D. Archambault, T. Munzner, and D. Auber, "TugGraph: Path-Preserving Hierarchies for Browsing Proximity and Paths in Graphs," *Proc. Second IEEE Pacific Visualization Symp.,* pp. 113-121, 2009.

[8] D. Auber, "Tulip: A Huge Graph Visualization Framework," *Graph Drawing Software,* P. Mutzel and M. Jünger, eds., pp. 105-126, Springer-Verlag, 2003.

[9] D. Auber and F. Jourdan, "Interactive Refinement of Multi-Scale Network Clusterings," *Proc. Ninth Int'l Conf. Information Visualisation (IV '05),* pp. 703-709, 2005.

[10] M. Balzer and O. Deussen, "Level-of-Detail Visualization of Clustered Graph Layouts," *Proc. Sixth Int'l Asia-Pacific Symp. Visualization (APVIS '07),* pp. 133-140, Feb. 2007.

[11] F. Boutin, J. Thièvre, and M. Hascoët, "Focus-Based Filtering + Clustering Technique for Power-Law Networks with Small World Phenomenon," *Proc. Conf. Visualization and Data Analysis,* 2006.

[12] E. Di Giacomo, W. Didimo, L. Grilli, and G. Liotta, "Graph Visualization Techniques for Web Clustering Engines," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 2, pp. 294-304, Mar./Apr. 2007.

[13] P. Eades and Q. Feng, "Multilevel Visualization of Clustered Graphs," *Proc. Conf. Graph Drawing (GD '96),* pp. 101-112, 1996.

[14] P. Eades and M.L. Huang, "Navigating Clustered Graphs Using Force-Directed Methods," *J. Graph Algorithms and Applications,* vol. 4, no. 3, pp. 157-181, 2000.

[15] C. Faloutsos, K.S. McCurley, and A. Tomkins, "Fast Discovery of Connection Subgraphs," *Proc. 10th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 118-127, 2004.

[16] G. Flake, S. Lawrence, and C.L. Giles, "Efficient Identification of Web Communities," *Proc. Sixth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 150-160, 2000.

[17] E. Gansner, Y. Koren, and S. North, "Topological Fisheye Views for Visualizing Large Graphs," *IEEE Trans. Visualization and Computer Graphics,* vol. 11, no. 4, pp. 457-468, July 2005.

[18] D. Gibson, J.M. Kleinberg, and P. Raghavan, "Inferring Web Communities from Link Topology," *Proc. Ninth ACM Conf. Hypertext and Hypermedia,* pp. 225-234, 1998.

[19] Y. Koren, S. North, and C. Volinsky, "Measuring and Extracting Proximity in Networks," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 245-255, 2006.

[20] M. McGuffin and I. Jurisica, "Interaction Techniques for Selecting and Manipulating Subgraphs in Network Visualizations," *IEEE Trans. Visualization and Computer Graphics,* to be published.

[21] T. Moscovich, F. Chevalier, N. Henry, E. Pietriga, and J.D. Fekete, "Topology-Aware Navigation in Large Networks," *Proc. SIGCHI Conf. Human Factors in Computing Systems (2009),* pp. 2319-2328, 2009.

[22] T. Munzner, F. Guimbretiere, and G. Robertson, "Constellation: A Visualization Tool for Linguistic Queries from Mindnet," *Proc. IEEE Symp. Information Visualization (InfoVis '99),* pp. 132-135, 1999.

[23] T. Pattison, R. Vernik, and M. Phillips, "Information Visualization Using Composable Layouts and Visual Sets," *Proc. Asia-Pacific Symp. Information Visualization,* pp. 1-10, 2001.

[24] C. Rozenblat, G. Melançon, M. Amiel, D. Auber, C. Discazeaux, A. L'Hostis, P. Langlois, and S. Larribe, "Worldwide Multi-Level Networks of Cities Emerging from Air Traffic," *Proc. Urban Changes in Different Scales: Systems and Structures,* pp. 487-502, 2006.

[25] D. Schaffer et al., "Navigating Hierarchically Clustered Networks Through Fisheye and Full-Zoom Methods," *ACM Trans. Computer-Human Interaction,* vol. 3, no. 2, pp. 162-188, 1996.

[26] C. Tominski, J. Abello, F. van Ham, and H. Schumann, "Fisheye Tree Views and Lenses for Graph Visualization," *Proc. 10th Int'l Conf. Information Visualisation (IV '06),* pp. 17-24, 2006.

[27] F. van Ham and J. van Wijk, "Interactive Visualization of Small World Graphs," *Proc. IEEE Symp. Information Visualization (InfoVis '04),* pp. 199-206, 2004.

[28] C. Ware and R. Bobrow, "Motion to Support Rapid Interactive Queries on Node Link Diagrams," *Proc. ACM Trans. Applied Perception,* pp. 1-15, 2004.

Daniel Archambault received the BSc Hons degree from Queen's University at Kingston in 2001 and the PhD degree from the University of British Columbia in 2008. He is currently a post doctoral researcher at INRIA Bordeaux Sud-Ouest. His research interests include graph drawing, visualization, and computer graphics. He is a member of the IEEE Computer Society.

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

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