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

Daniel Archambault*
University of British Columbia & INRIA Bordeaux Sud-Ouest

Tamara Munzner†
University of British Columbia

David Auber‡
University of Bordeaux

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 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.

Keywords: Graph Visualization, Proximity, Graph Hierarchies

Index Terms: H.5.0 [Information Systems]: Information Interfaces and Presentation—General G.2.2 [Mathematics of Computing]: Discrete Mathematics—Graph Algorithms

1 INTRODUCTION

Many systems engineered to explore and create graph hierarchies detect subgraphs in the input graph as a basis for hierarchy construction. These approaches search for topological features [2, 5] or features based on attribute data [18, 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 graph structure. Examples of these global questions would include: *Where are the trees in this graph?* or *What parts of the Internet are backbone servers?*

However, these approaches do not have provisions for browsing parts of the graph near, in a graph theoretic sense, a node or subgraph of interest. 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. In a graph context, the user searches for nodes near a node or subgraph: *How does my computer network connect to the Internet?* or *What flights are there to Vancouver?* We term this notion **proximity**. In TugGraph, we exploit graph hierarchies to browse proximity to a feature.

A **graph hierarchy** or **hierarchy**, shown in Figure 1(a), 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**, the grey swath shown in Figure 1(a), 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 drawn opaquely in the graph view at the bottom. 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, after computing the metanodes which contain the proximal components, the hierarchy cut is lowered, revealing the metanodes containing those proximal components as shown in Figure 1(b).

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 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 primary contribution of TugGraph is a technique, and algorithms implementing the technique, for exploring a region of the graph located near a feature. We demonstrate TugGraph on input graphs with hundreds of thousands of nodes and millions of edges. The system is built on the GrouseFlocks [6] architecture, which supports both the navigation and creation of graph hierarchies. TugGraph does not require the entire graph to be drawn beforehand, allowing exploration to begin immediately.

2 PATH-PRESERVING HIERARCHIES

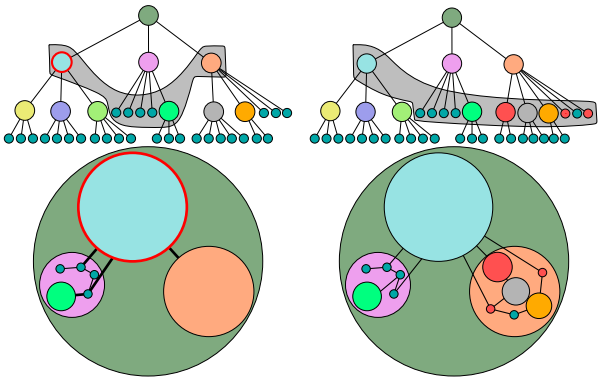
A **path-preserving hierarchy**[1] [6], shown in Figure 2, is a specific type of graph hierarchy that must respect two properties:

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 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.

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

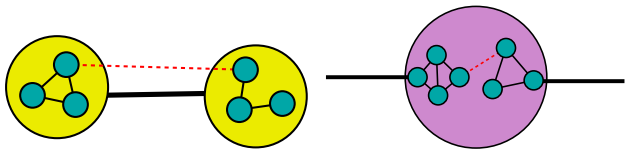
*email: daniel.archambault@inria.fr
†email: tmm@cs.ubc.ca
‡email: auber@labri.fr

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



(a) Input (b) Output

Figure 1: Result of one TugGraph operation, with hierarchy view above and graph view below. **(a)** A cut node of the hierarchy is selected, in this case, the highlighted node in light blue as shown. **(b)** The result of the TugGraph operation on this node. All the nodes in red consist of leaves or proximal components of the light blue node.



(a) Edge Conservation (b) Connectivity Conservation

Figure 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.

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 Figure 2(a). Connectivity conservation guarantees that there exists a path in the original graph through the metanode on the hierarchy cut as shown in Figure 2(b). Both edge and connectivity conservation are required in order to visualize paths and proximity information. 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.

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 [12], multi-focal fisheye approaches where metanodes are expanded and viewed in the context of the entire graph [19], topological fisheyes where abstract versions of the graph are presented far away from a focus centre [15], linking the graph hierarchy to a separate treemap view [1], interactively visualizing hierarchies of small world clusterings [20], and visualization of complex software in three dimensions using level of detail techniques [9].

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 [11] 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 towards exploring the global structure of the graph and the topological or attribute features present in it.

The DA-TU system [13] of Huang and Eades is a force-directed approach which biases the hierarchy cut towards its hierarchy structure; Auber and Jourdan [8] support interactive hierarchy editing; and the Clovis system [18] 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 [14, 16]. 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.

3.3 Subgraph Highlighting in Graphs

Selection and other techniques that exploit pre-attentive channels such as motion [21] or colour using hover queries [17] can be used to highlight parts of a graph including paths. In contrast, Boutin *et al.* [10] 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. Firstly, a significant portion of the entire of the graph must be drawn before visualization can begin. Secondly, 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.

4 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 to reveal adjacent input graph components. It is outlined in red in Figure 3(a). On this input, the algorithm operates in five stages:

1. Compute the set of nodes in the input graph, or leaf nodes in the hierarchy, that are descendants of the source. This set of nodes is the **source set** denoted S.
2. Discover the set of leaf nodes of graph-theoretic distance one from the source set that are not elements of the source set themselves. This set is the **proximal set** denoted P.
3. Determine the set of cut metanodes that contain elements of the proximal set. This set is the **proximal cut set** denoted C.
4. For each element n of the proximal cut set, place nodes of the proximal set inside their own metanodes respecting the constraints of a path-preserving hierarchy directly below n.
5. Reconstruct the hierarchy for all other leaf nodes that are descendants of metanodes of the proximal cut set but not elements of the proximal set.

Figure 3(h) shows the result of the five stages on a metanode selected on the graph hierarchy. When describing each step, the complexity of each step is presented. The execution of TugGraph produces a modified graph hierarchy and cut. Elements of the proximal set, all of which were below the cut supplied as input, appear in their containing proximal cut metanodes that are moved above the hierarchy cut.

4.1 Computing the Source Set

The source set S is the set of nodes of the input graph that are descendants of the selected node on the hierarchy cut. If the selected node is a leaf, S contains one element: the selected node. If S is a metanode, as in Figure 3(a), the algorithm traverses the graph hierarchy top down from the selected metanode to discover all leaf descendants as shown in Figure 3(b). These leaves are the source set S, outlined in red in Figure 3(c).

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

4.2 Computing the Proximal Set

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

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

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 Equation (1). The result of this part of the algorithm is shown in Figure 3(d).

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

4.3 Computing the Proximal Cut Set

The algorithm derives the proximal cut set C from the proximal set. The proximal cut set is the set of metanodes currently present on the hierarchy cut that contain elements of the proximal set. This set is computed by traversing the graph hierarchy bottom up from each proximal set element up to the first cut metanode ancestor. Figure 3(e) 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 stage is $O(|P| + |M_P|)$.

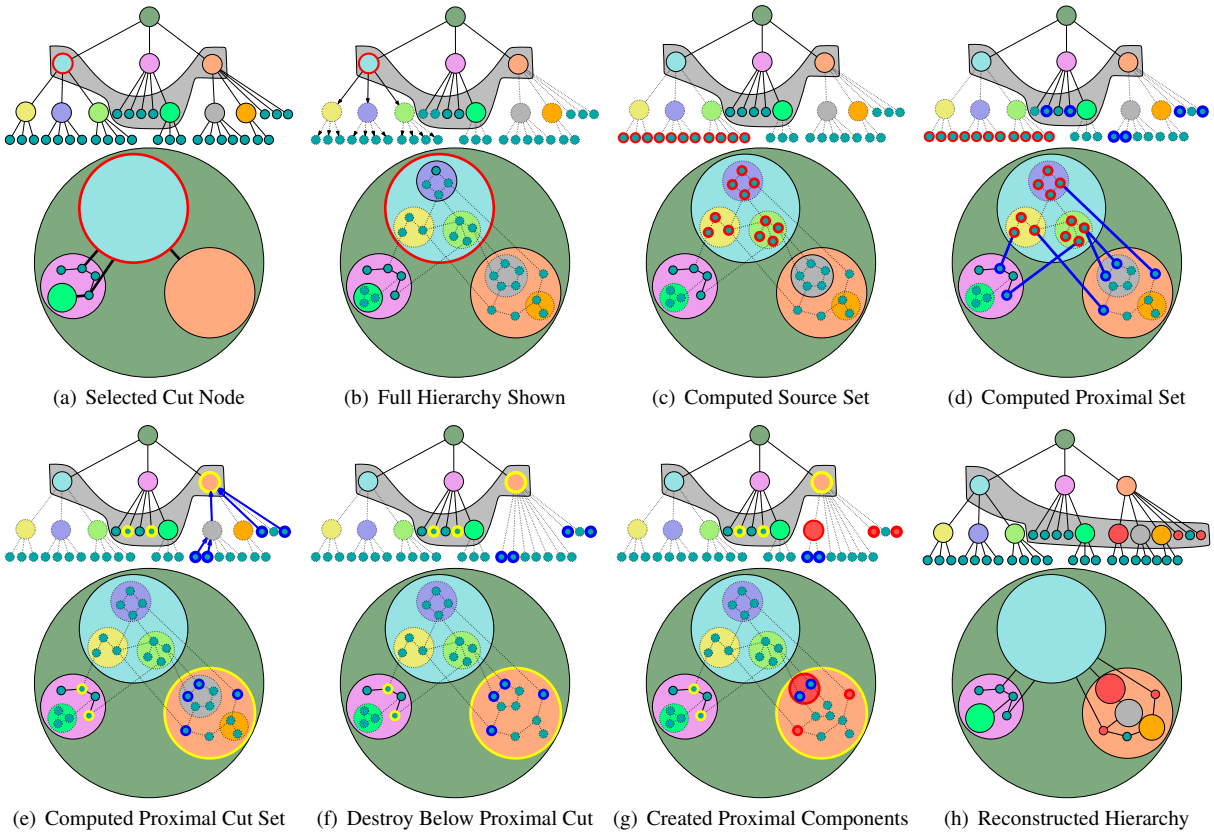
4.4 Computing the Proximal Components

Once the algorithm has determined the proximal cut set and the proximal set, it will proceed to reconstruct the hierarchies below the proximal cut set such that the elements of the proximal set are in metanodes that respect the rules of a path-preserving hierarchy. These subgraphs are proximal components as every element is an element of the proximal set, meaning they are directly connected by an edge to an element of the source set.

Figure 3(e) shows the input to this stage of TugGraph. The proximal set, P, is outlined in blue, while the proximal cut set, C, is outlined in yellow. Before proceeding, a copy of the graph hierarchy is created so that it can be reconstructed below the proximal cut nodes in the last phase.

In Figure 3(f), the hierarchy below every element of the proximal cut set is destroyed. The resulting hierarchy below any proximal cut node is always a set of leaves. If the cut proximal element is a leaf of the graph hierarchy, it remains unaffected by this step as there is no hierarchy below it to destroy. This step is identical to a recursive delete operation as described in GrouseFlocks [6].

The algorithm then computes the proximal components as shown in Figure 3(g). These components are the set of induced subgraphs by nodes of the proximal set and each induced subgraph is placed inside its own metanode. An induced subgraph is defined by a set of nodes, in this case the nodes of P, and any edge that links a pair of nodes in P. The result is a set of connected subgraphs. If each connected subgraph is placed in its own metanode, it respects connectivity conservation. If every edge that connects a node in the



(a) Selected Cut Node (b) Full Hierarchy Shown (c) Computed Source Set (d) Computed Proximal Set

(e) Computed Proximal Cut Set (f) Destroy Below Proximal Cut (g) Created Proximal Components (h) Reconstructed Hierarchy

Figure 3: 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.

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.

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

4.6 Worst Case Complexity

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

5 Colouring and Node Sizes

Many TugGraph operations can be executed on an input graph one after the other and in conjunction with Reform Below Cut operations of GrouseFlocks. In order to distinguish multiple tugs, the system rotates through the colours: purple, tan, blue, green, and light blue. Proximal components are a more saturated version of the colour while all other components are less saturated. Open metanodes are bounded in a background disk of the same colour.

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 datasets. The small components are the few nodes adjacent

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

Figure 4: Summary of asymptotic complexity of TugGraph stages. The sets S is and P are the source and proximal sets. The sets D_S and D_P are the sum of the degrees of all nodes in the S and P sets respectively. The sets M_S and M_P consist of the sets of metanodes that exist above S and P to the elements of C.

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

TugGraph is implemented using the Tulip graph drawing libraries [7] and GrouseFlocks [6]. We compare TugGraph to existing systems on three datasets.

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

The `Net05` dataset [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` dataset is an IMDB subset centered around Sharon Stone only considering movies in the years 1998 through 2001. In this graph, nodes are actors and there exists an edge between two nodes if those actors acted in a movie together in those years. Actor name is the only attribute on the nodes. The dataset has 38,997 nodes and 1,948,712 edges.

For each dataset, a result is presented using TugGraph and the result or part of the result is highlighted in the 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 colours. We used a 3.0GHz Pentium IV with 3.0GB of memory running SuSE Linux with a 2.6.5-7.151 kernel.

6.1 Airport

We browse how the flight paths between Columbus and Vancouver are interconnected in `Airport`. Figure 5 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. Figure 5(a) shows that even finding the airports one hop away is buried very deep in a hierarchy of topological features. Figure 5(b) demonstrates that GrouseFlocks is better able to solve this problem. The system decomposes the graph into three components initially: Vancouver, Columbus, and other airports. Since there is no attribute information for node proximity, coarsening is used to explore the airports adjacent to both Vancouver and Columbus as shown in Figure 5(c). The solution improves on that of Grouse, but the airports one hop away from Vancouver are still scattered over the hierarchy.

Figure 5(d) and 5(e) present the results using TugGraph. The process starts with same initial decomposition shown in Figure 5(b). First, the source node `Vancouver` is tugged, extracting the airports one hop away from it. In Figure 5(d), 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. Figure 5(e) 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.

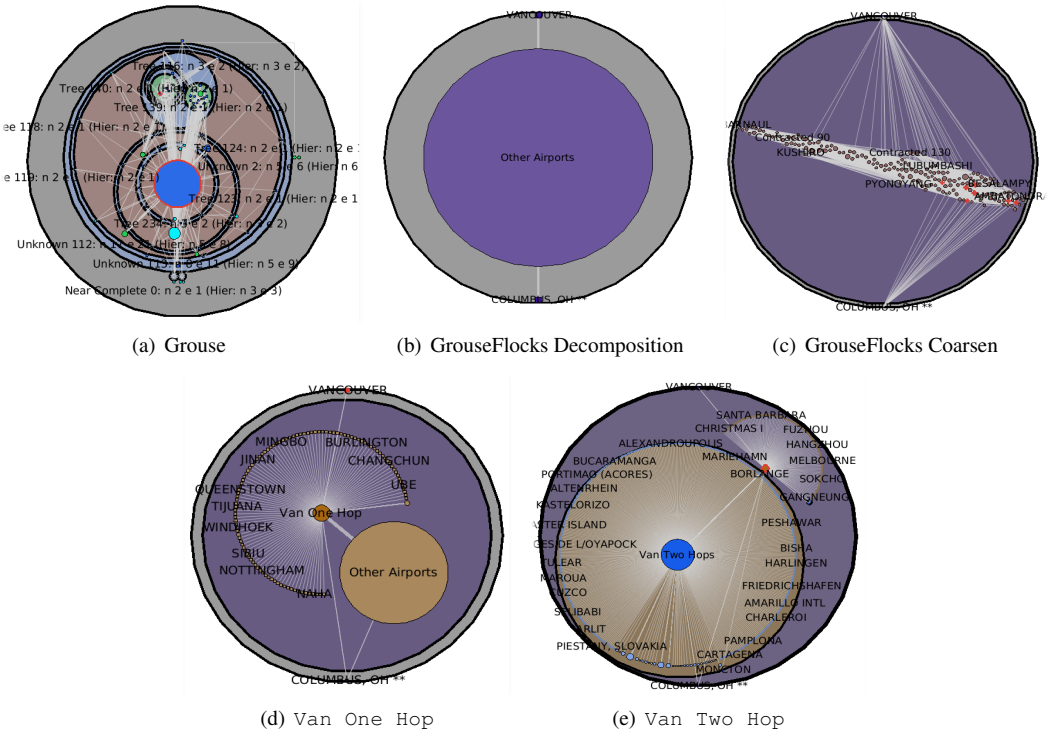
6.2 Net05

We browse the structure around the `*.net.ubc.ca` portions of the network in `Net05`. After twelve 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 dataset.

The results are shown in Figure 6. Once again, the problem is solved using TugGraph, and we use highlighting to show the solution in other approaches. Figures 6(a) and 6(b) show where the UBC servers are in the dataset and highlight the portions that are four hops away. In these figures, all nodes and edges of the dataset four hops from the `*.net.ubc.ca` servers are highlighted. As the graph has not been simplified, it is difficult to see the path in the context of the entire graph. The paths between the UBC servers that are far away from each other cannot be emphasized without redrawing the data. With GrouseFlocks, shown in Figure 6(c), the initial decomposition segments out the UBC servers into two disconnected components with the rest of the Internet in between them. As with the previous dataset, when the huge `Internet` metanode is expanded it is too complex to draw in full and must be coarsened, as shown in Figure 6(d). 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.

Figures 6(e) through 6(j) show how TugGraph can help browse the connections of UBC into the Internet to see if there is a single server that connects UBC to the Internet. Again, TugGraph starts from the initial GrouseFlocks decomposition shown in Figure 6(c). First, the `UBC` metanode is tugged, revealing that the two parts of the UBC network are still disconnected and adjacent to the two saturated tan leaves `142.103.204.2` and `ubci9-tx-vantx.bc.net`. However, there is no single connection. `ubci9-tx-vantx.bc.net` is tugged and the result is shown in Figure 6(f). `rx0wh-bcnet.vc.bigpipeinc.com` is the greatest common ancestor, joining the two disconnected components of UBC to `Internet` as no other edges connect Internet to the rest of the graph. We continue browsing this connection in Figures 6(g) through 6(j) by tugging on the nodes named in the

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



(a) Grouse (b) GrouseFlocks Decomposition (c) GrouseFlocks Coarsen

(d) `Van One Hop` (e) `Van Two Hop`

Figure 5: 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 coloured using the type of topological feature they contain. **(b)** In GrouseFlocks and TugGraph, Vancouver and Columbus are coloured 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.

captions and outlined in red in each of these figures.

6.3 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 twelve hours of execution time. GrouseFlocks could be used for this exploration, but would produce images very similar to the ones already shown. We thus show only a TugGraph result in Figure 7. For this dataset, logscale node sizes are used.

Figure 7 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.

6.4 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 two seconds to complete.

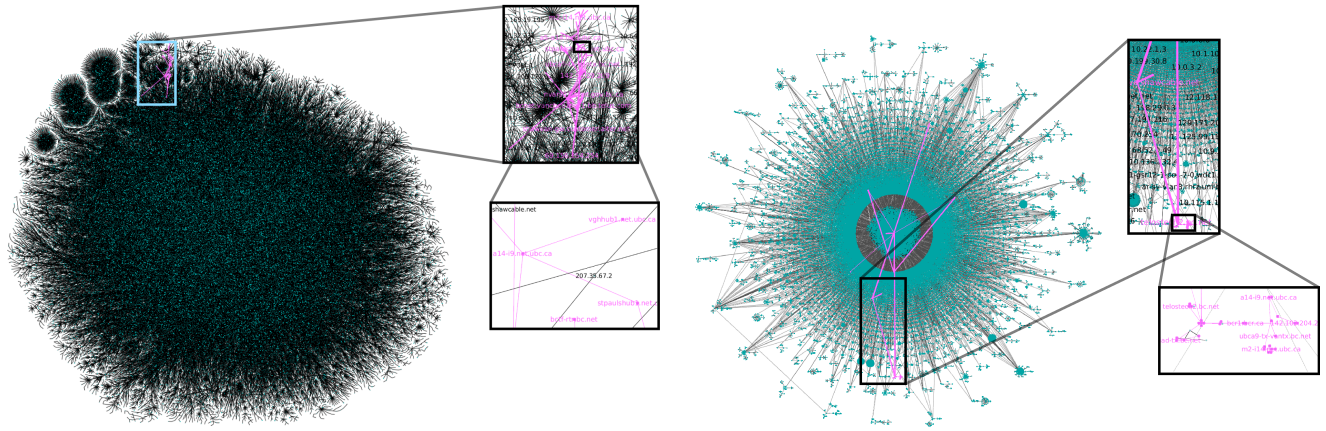
To draw the `Net05` dataset LGL and SPF took 12 hours and 30 minutes respectively to draw the entire graph. GrouseFlocks took 115 seconds for the initial decomposition and 15 seconds to produce the coarsened graph. After selection and decomposition into UBC and non-UBC servers, TugGraph took about 20 seconds to tug out each proximal component along the path.

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

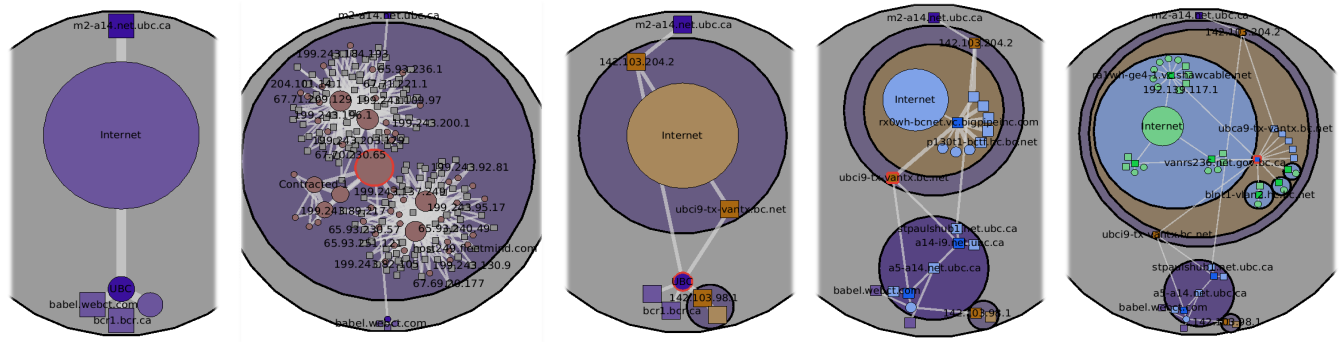
7 Discussion

In the three example datasets, 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.

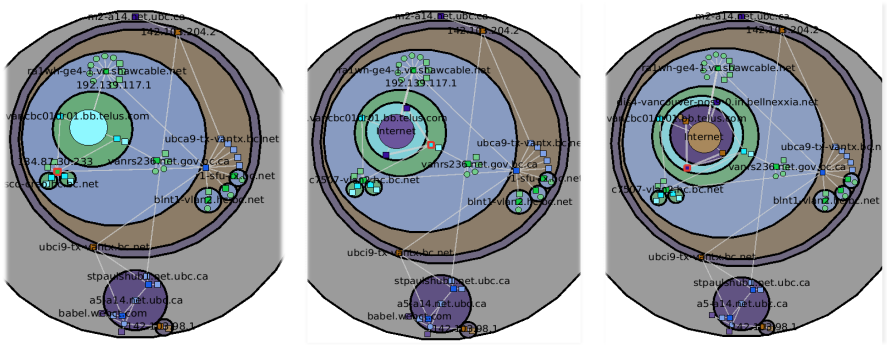
Another important observation is that the TugGraph result images do not require zoomed-in insets to show details of the graph structure. Typically in large graph visualization systems, many scales are needed to understand features in the data in a global context. Hierarchy-based visualization tools,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



(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 (j) Tug bellnexia

Figure 6: Exploration of the `Net05` dataset 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 dataset is minimal. TugGraph, however, shows how UBC connects to the Internet.

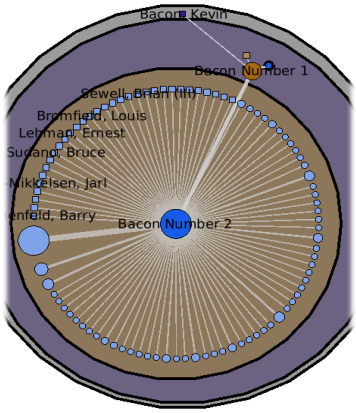


Figure 7: Bacon number trends in `Actors`. `Bacon Number 1` contains all actors with a Bacon number of one. The saturated blue node to its right and `Bacon Number 2` contain actors with a Bacon number of 2. We see that actors with Bacon numbers of one or two tend to act with each other as their components are connected.

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 Conclusion and Future Work

We have presented an 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. TugGraph does not require the entire graph to be drawn beforehand, allowing exploration to begin immediately.

TugGraph provides a fairly simple way to discover elements directly adjacent to a source node in unweighted graphs. In future work, we hope to extend TugGraph to handle weighted graphs, to handle elements multiple hops away and to speed up execution of the algorithm as response is still not interactive. This generalization would allow TugGraph to handle a wider range of data.

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 summarized in a path-preserving way. In future work, we should investigate path-preserving coarsening techniques which can effectively summarise a large number of proximal components in a simple drawing. Additionally, more compact representations for a large number of disconnected components remains future work.

Finally, user experimentation and studies with domain experts would be essential to validate the usability of the technique. In the future, 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.

Acknowledgments

The first author would like to thank the InfoVis group at UBC and the reviewers of his thesis who helped improve this work.

References

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

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

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

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

[5] D. Archambault, T. Munzner, and D. Auber. Grouse: Feature-based, steerable graph hierarchy exploration. In *Proc. of Eurographics/IEEE VGTC Symp. on Visualization (EuroVis '07)*, pages 67–74, 2007.

[6] D. Archambault, T. Munzner, and D. Auber. GrouseFlocks: Steerable exploration of graph hierarchy space. *IEEE Trans. on Visualization and Computer Graphics*, 14(4):900–913, 2008.

[7] 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.

[8] 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.

[9] 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.

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

[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] 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.

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

[14] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *Proc. of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 118–127, 2004.

[15] 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.

[16] Y. Koren, S. North, and C. Volinsky. Measuring and extracting proximity in networks. In *Proc. of SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 245–255, 2006.

[17] T. Munzner, F. Guimbretiere, and G. Robertson. Constellation: A visualization tool for linguistic queries from mindnet. In *Proc. IEEE Symp. on Information Visualization (InfoVis'99)*, pages 132–135, 1999.

[18] 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.

[19] 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.

[20] 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.

[21] C. Ware and R. Bobrow. Motion to support rapid interactive queries on node-link diagrams. In *ACM Transactions on Applied Perception*, pages 1–15, 2004.