Visualizing Massive Multi-Digraphs

James Abello Jeffrey Korn

Information Visualization Research
Shannon Laboratories, AT&T Labs-Research
{abello,jlk}@research.att.com

Abstract

We describe *MGV*, an integrated visualization and exploration system for massive multi-digraph navigation. *MGV*'s only assumption is that the vertex set of the underlying digraph corresponds to the set of leaves of a predetermined tree T. *MGV* builds an out-of-core graph hierarchy and provides mechanisms to plug in arbitrary visual representations for each graph hierarchy slice. Navigation from one level to another of the hierarchy corresponds to the implementation of a drill-down interface. In order to provide the user with navigation control and interactive response, *MGV* incorporates a number of visualization techniques like interactive pixel-oriented 2D and 3D maps, statistical displays, multi-linked views, and a zoomable label based interface. This makes the association of geographic information and graph data very natural. *MGV* follows the client-server paradigm and it is implemented in C and Java-3D. We highlight the main algorithmic and visualization techniques behind the tools and point out along the way several possible application scenarios. Our techniques are being applied to multi-graphs defined on vertex sets with sizes ranging from 100 million to 250 million vertices.

Keywords: visualization, massive data sets, graphs, hierarchies, out-of-core algorithms.

1 Introduction

Processing, querying, exploring and visualizing massive data sets poses a series of interesting computational and visual challenges. A variety of these data sets can be modeled as very large but sparse multi-digraphs with a special collection of application dependent edge attributes. Geographic information systems, telecommunications traffic and internet data are prime examples of the type of data our system is targeted to handle.

Sheer size is the first fundamental issue that needs to be addressed when the data to be dealt with is considered massive. In our case, for one of the data sets, we receive a stream of about 275 million records daily yielding about 450GBytes per month. Having access to several SGI Origin-2000 servers, 5 terabytes of disk and an SGI Onyx connected to a $5120 * 2048$ power wall certainly helps in the processing but it does not circumvent two important bottlenecks: I/O bandwidth and screen real estate.

The I/O bottleneck is caused by the substantial difference between CPU speeds and external memories. Algorithms whose performance is stated in terms of not just the input size, N, but also in terms of the size of main memory, M and of the disc block transfer size, B, are called external memory algorithms[1]. With this framework in mind, the first requirement for a data set to be considered massive is that its size (N) must be larger than the size of available RAM (M). In the case of multi-digraphs, N is essentially $O(|E|)$ where $|E|$ is the number of edges of the underlying graph.

[1]See J. Abello and J. Vitter [10] for a recent review of this subject.

An intermediate case, quite relevant in practice, occurs when the set of vertices fits in RAM but not the edge set (this is called the semi-external case in [6]). The justification for this model relies on the increased availability of large RAMs. For example, the essential information associated with 250 million vertices fits nicely in 2GBs of RAM. In this case, in principle, one can process any secondary storage multi-digraph with vertex set up to this size.

The screen bottleneck is caused by the simple fact that the amount of information that can be displayed at once is ultimately limited by the number of available pixels and the speed at which the information is digested by a user. Even though a large number of pixels diminishes the screen bottleneck, it does not help the user's visual processing abstraction unless the display metaphor incorporates some global data set semantics. Luckily, a variety of massive multi-digraphs are implicitly defined on vertex sets that correspond to the leaves of a predefined hierarchy T. When we can induce such a hierarchy we can use it to guide the exploration and visualization of the data set. This is done by defining an inherited equivalence relation on the multi-digraph edge set (see Section 2). The hierarchy makes it possible for a user to digest one manageable portion of the data at a given time.

In order to deal in a unified manner with both the I/O and the screen bottlenecks, we base our work on a metaphor called *hierarchical graph slices*. The main idea is to build a hierarchy of multi-digraph *layers* on top of the input multi-graph. Each layer is obtained by coalescing disjoint sets of vertices at a previous level and aggregating their corresponding weighted edges. A collection of edges in a layer whose aggregation produces an edge at the next higher layer is called an *edge slice*. Several "natural" operations provide hierarchical browsing. Each edge-slice is small enough to be represented visually in a variety of ways, such as a 2D needle-grid, a 2D star-grid or star-map, a 2D surface in R^3 or a conventional graph drawing. Slices have different properties depending on their depth, as shown in Figure 1. Slices at a greater depth are represented by more pixel hungry representations. Representations can be chosen automatically based on properties of the data, or can be plugged in manually by a system user.

Many of our visualizations depart strongly from the conventional visual graph representation that draws graphs as nodes and edges, unless the slice being considered is very sparse and defined on a very small number of vertices and edges. In our hierarchical decomposition, when facing a dense subset of edges, we use adjacency matrix based visualizations since they are likely easier to digest. Conventional graph representations like the one shown in Figure 2 are of very limited use for the range of sizes being considered in this work. This paper presents new techniques that are particularly helpful in visualizing dense slices.

When a hierarchy T is fixed, the corresponding graph-layers can be updated incrementally. They are suitable for the processing, navigation and visualization of external memory graphs [6] whose vertex sets are hierarchically labeled.

A by-product of the hierarchical graph-slices metaphor is that a commercial relational database can be used to query the multi-digraph hierarchy with very little extra effort. Also, hierarchical

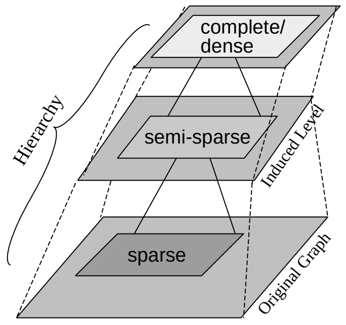

Figure 1: Graph layers. Layers in deeper levels tend to be sparser.

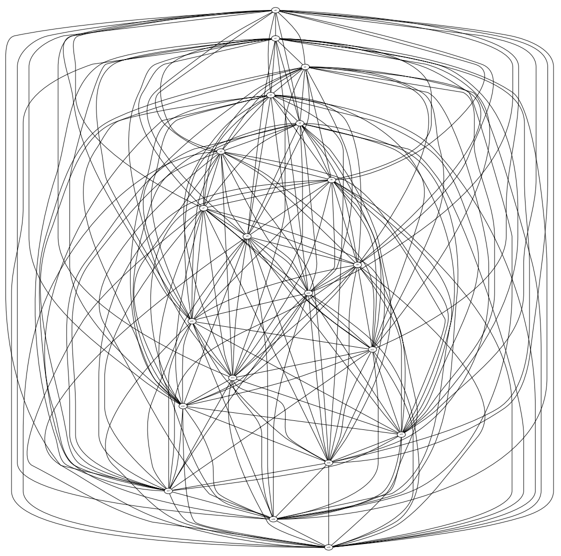

Figure 2: Traditional nodes-and-edges representation of a fully connected graph with 20 nodes.

graph-slices are amenable to distributed visual exploration.

Our current prototype (termed *Massive Graph Visualizer*) is a system with the following highlights:

- It handles hierarchical views of massive multi-digraphs.
- It consists of a C-computational engine (server) and a Java-3D visualizer (client), which may reside on separate machines. In fact, the visualizer can run on multiple desktops allowing different users to navigate a massive data set independently.
- It provides a drill-down zoom-able interface together with a collection of multi-linked views.
- Context is maintained by using multiple cameras. One provides an overview and the others trail each other depending of a user specified zooming interval. A persistent history of previous navigations of the hierarchy is maintained.
- In the case of geographical data, displays such as the star-map (Section 4.2) allow the superposition of graph neighborhood information on a given geography. This offers an alternative to the conventional approach of explicitly drawing the edges among specified positions on a given map.
- Visual aggregation can be obtained by special views, such as our multi-comb view (Section 4.3) or by an adaptation of the circle of segments technique [3].
- Users can plug-in alternative visualizations of the hierarchical graph slices, and can apply their own filters to the slices.

1.1 Related Work

The work presented here grew out of the graph surfaces metaphor presented in [9]. The primary difference is that 2D surfaces are not easy to refine locally. By choosing different representations for the higher levels of the hierarchy we get very fast local refinement, a very intuitive visual aggregation operation and visually pleasant animations of data set evolution.

The vertex set of our hierarchy is a super-set of the vertex set of the underlying multi-digraph. This makes our approach quite different than other graph visualizations based on spanning trees of the underlying graph (see Munzner [16], Wills [8]). The use of hierarchies for the exploration of large graphs is explicitly mentioned in [7]. Our work can be viewed as an automation of these ideas that provides a uniform overall view of massive graph data together with scalable, efficient and flexible visual navigation tools.

The layout of the paper is as follows. In Section 2, we discuss graph slices, the main elements of the computational engine, and its fundamental operations and I/O performance. In Section 3, we discuss the correspondence between the slice hierarchy and the different visual representations. The components of the Java-3D visualizer and the main interface issues are the contents of Sections 4 and 5. Section 6 points out some future research directions.

2 Hierarchical Graph Slices

In order to handle very large graphs, a hierarchy of *multi-digraph layers* is constructed. Each *layer* represents a multi-digraph obtained from an equivalence relation defined on the edge set of the input multi-graph. Each *layer edge* represents an equivalence class of edges at the previous layer. Each such equivalence class constitutes what we call an *edge-slice*. Zooming operations are provided that allow the user to explore the *graph slice hierarchy* in a fluid manner.

We introduce these concepts more formally next. Figure 3 illustrates our definitions.

2.1 Definitions

- For a multi-digraph G, let $V(G)$ and $E(G)$ denote the set of vertices and edges of G respectively. It is assumed that a function $m : E \rightarrow N$ assigns to each edge a non-negative multiplicity. With these conventions a multi-digraph is a triplet $G = (V, E, m)$.
- For a *rooted* tree T, let $Leaves(T)$ = set of leaves of T. $Height(T)$ = maximum distance from a vertex to the root of T; $T(i)$ is the set of vertices of T at distance i from the root of T. For a vertex $x \in T$, let T_x denote the subtree rooted at x. Vertices p and q of a *rooted* tree T are called *incomparable* in T if neither p nor q is an ancestor of the other.
- Given a multi-digraph $G = (V, E, m)$ and a rooted tree T such that $Leaves(T) = V(G)$, the multiplicity of a pair of vertices p and q of T is $m(p,q) = \sum_{(x,y)\in E(G)} m(x,y)$ for $x \in Leaves(T_p)$ and $y \in Leaves(T_q)$. An *incomparable* pair (p,q) is called a *multi-edge* when $m(p,q)$ is greater than zero. When both p and q are at the same distance from the root of T, the multi-edge is called *horizontal*. A *non-horizontal*

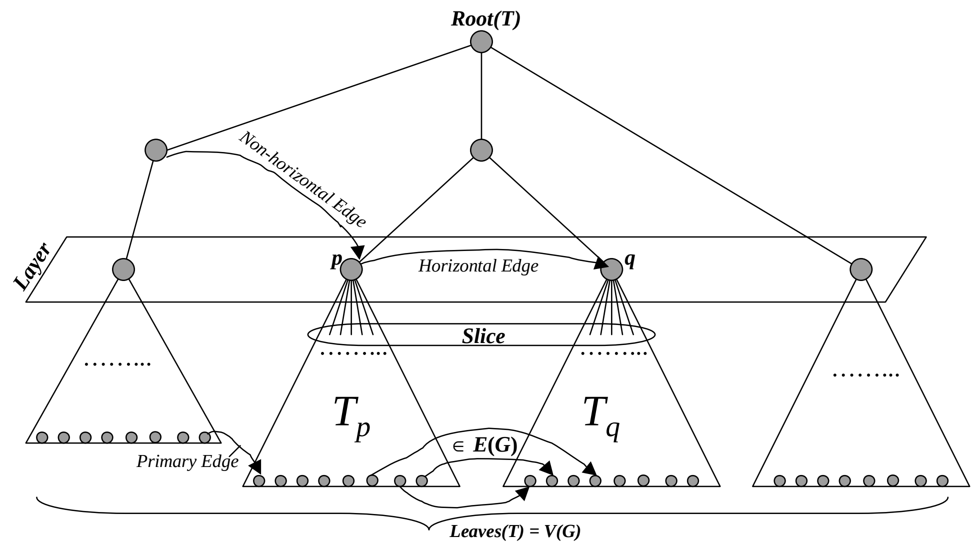

Figure 3: Hierarchical Graph Decomposition

multi-edge between vertices p and q where p is a *leaf* and $Height(q) > Height(p)$ is called a *primary crossing* multi-edge.

Notice that a *horizontal* multi-edge $(p, p, m(p, p))$ represents the subgraph of G induced by $Leaves(p)$ and $m(p, p)$ is its aggregated multiplicity.

- For G and T as above, the *hierarchical graph decomposition* of G, given by T, is the multi-digraph $H(G, T)$ with vertex set equal to $V(T)$ and edge set equal to the edges of T union the multi-edges running between *incomparable* pairs of T.

  Because $H(G, T)$ contains a very large collection of *multi-edges* that can be computed from the *horizontal* and *primary crossing* multi-edges as defined above, we take the approach of maintaining just these multi-edges and computing the remaining ones on demand. This sub-multigraph is denoted by $LH(G, T)$. $LH(G, T)$ can be viewed as a collection of *layers* representing an equivalence relation on $E(G)$. Each *layer* contains *horizontal* multi-edges only. The *primary crossing* multi-edges indicate inter-layer data relations. It is precisely this layered view of a graph what allow us to explore it visually.

- For G and T, as above, the *i-layer* of G is the multi-digraph with vertex set $T(i)$ and all the corresponding *horizontal* multi-edges.

- For a multi-edge (x, y) of an *i-layer* its *edge-slice* is the sub-multigraph of the *(i+1)-layer* whose nodes are the children of x union the children of y, and whose multi-edges are those in the *(i+1)-layer* running between these nodes.

- A good mental picture of what the definitions convey is that each multi-edge (p, q) has below it a hierarchy of *edge-slices* where each level represents an aggregation of previous levels and where the bottom most level is the subgraph of G with vertices $Leaves(T_p)$ union $Leaves(T_q)$ and edges of G running between them. This is the justification for naming this section *Hierarchical Graph Slices*.

2.2 Constructing $LH(G, T)$

The procedure **Construct** $LH(G, T)$, presented in [9], takes as input a stream of edges representing a multi-digraph G and a rooted tree T such that $Leaves(T) = V(G)$. It returns as output, a disk resident, multi-level index structure to the edges of $LH(G, T)$.

Lemma 1. $LH(G, T)$ can be constructed in time

$$O(|V(G)| * Height(T) + |E(G)|)$$

in a bottom-up fashion [9, 12]. Space requirements are similar, making $LH(G, T)$ an efficient data structure to use for our visualization system.

Because $LH(G, T)$ is really T plus the collection of layers of G given by T, we can think of each layer as being represented by a two dimensional grid and T as a road map to navigate the slice hierarchy.

2.3 Handling the I/O bottleneck

When G is an external memory graph residing on disk there are three cases to consider: (1) T fits in main memory, (2) T does not fit but $V(G)$ does, and (3) $V(G)$ does not fit. The first two cases correspond to what is called the semi-external version and the third one is referred to as fully external. We center our discussion in the first two cases since they suffice for our applications. The third case is not fully understood yet and its solution may take something of a breakthrough both at the algorithmic and at the systems level. In the first case, the edges of G are read in blocks and each one is filtered up through the levels of T until it lands in its final layer. This can be achieved with one pass.

In the second case, a multilevel external memory index structure is set up to represent T as a parent array according to precomputed breadth first search numbers. Filtering the edges through this external representation of T can be done in no more than $Height(T)$ scans over the data.

As pointed out in the introduction, the increased availability of large RAMs makes it realistic to assume that the vertex set fits in main memory. With multi-gigabyte RAMs being a reality and using our approach, one can process in principle any secondary storage multi-digraph defined on hundreds of millions of vertices.

2.4 Navigating the Hierarchy

The condition that $Leaves(T) = V(G)$ guarantees that every $T(i)$ determines a partition of $V(G)$ with every higher level being just a partial aggregation of this partition. This implies in turn that from any given layer one can move to any of the adjacent layers by partial aggregation or by refinement of some sets in the corresponding partition. This is precisely the information that is encoded in $LH(G, T)$. Namely, from any given multi-edge e in $LH(G, T)$ one can obtain the set of edges in G that are represented by e. This is the only operation that is needed to navigate since vertices in T can be easily replaced by their children by just following the tree edges. *Non-primary crossing* edges between non-leaves of the tree can be expanded by using the basic operations defined below. The I/O complexity is proportional to the difference in height between the two end points.

The main navigational operations used by the computational engine are:

- **Replacement:** Given a vertex u in T, $replace(u)$ substitutes u by its children. This can be implemented by generating edges $\{(u, u_i) : u_i$ is a child of u in $T\}$ and vertices children(u).

- **Vertex zoom:** Given a vertex u in T with children $u_1, u_2, ..., u_k$, *zoom*(u) generates $\{(u, u_i)$: u_i is a child of u in T and pairs (u_i,u_j) such that in the input multi-digraph the set of edges from $Leaves(u_i)$ to $Leaves(u_j)$ is non-empty$\}$.

- **Edge zoom:** Given an edge (u, v), *zoom*((u, v)) is defined as follows: $\{$delete the edge (u, v); $replace(u)$; $replace(v)$; add all the edges in the next layer that run from the children of u to the children of $v\}$.

Suitable inverses of the operations above can be defined provided certain restrictions are obeyed. For example, the inverse of $replace$ is defined, for a set of vertices, only if they are on the same layer and if they constitute all the children of a vertex u.

3 Visual Navigation

When we are visualizing data sets that are two to three orders of magnitude larger (say around 250 million records) than the screen resolution (typically about one million pixels), it becomes imperative to use a hierarchical decomposition of the visual space, particularly if we require real-time interactivity. In our case, we achieve fast response by navigating an input graph via its slices.

Our system allows the user to begin with a visualization of an initial layer, and interactively focus on selected edges which can be zoomed in to produce a visualization of a slice from the next layer down the hierarchy. Currently, the system uses a mouse/keyboard input interface. Using joysticks and gestures to navigate the environment is a possibility worth exploring. The best representation for a particular slice depends on properties of the graph representing that slice, so our system allows a variety of visualization techniques to be used for each slice. In the case of highly dense slices, which are usually encountered in higher layers of the slice hierarchy, we are often best off using adjacency matrix style visualizations since the number of edges is too large to effectively use the traditional nodes-and-edges visualization.

In our experience, the process of drilling down on slices works well to explore the real world multi-digraphs we are dealing with. Such data sets have highly skewed distributions, and this skewness can be directly observed by the visual cues in our 2D and 3D representations. For example, when we are dealing with phone records (calling frequency or total minutes of call), we are naturally interested in areas of larger edge weights. Looking at the grid representation shown in Figure 4, we can quickly determine such edges using the inclination and color of the sticks. We can then zoom into these sticks to obtain more refined views.

We now describe in more detail our scheme to visualize very large multi-digraphs. In this context, *large* refers to data sets that do not fit into main memory. Our system consists of two main components: *the C computational engine* and *the Java-3D graphical engine*. Given a large graph as input, the computational engine uses the approach outlined in the previous sections to cluster subgraphs together in a recursive fashion and generates a hierarchy of weighted multi-digraphs. The edge-slices in each layer of this hierarchy are sufficiently small to fit in main memory.

A typical large and realistic data set may have a number of interesting patterns and trends that information visualization and data mining applications want to explore. However, providing all this information in one shot might be too difficult to analyze or understand. In our metaphor, we *amortize the visual content* in every scene with the constructed graph hierarchy. Further, the reduced size of each edge-slice makes it possible to provide the necessary *real-time feedback* in such an exploratory setting. As the user traverses deeper into the hierarchy, the scene displayed becomes more detailed in a restricted portion of the data set.

The graphical engine has two primary functions - generating graph representations for individual slices in $H(G, T)$ using the navigation operations defined in the previous section, and displaying appropriate visual cues and labeled text. One of the aims is to help the user have intuitive understanding along with complete navigation control.

We now describe the main visual primitives that allow a user to move from one level of the hierarchy to another while changing the visual representation if necessary.

Zooming

As the user is viewing a particular slice, he/she can use the mouse or keyboard to pan, rotate, or zoom the image. A threshold can be set which defines between which zoom factors the visualization is valid. If the user zooms far enough in or out to exceed the threshold, a callback is invoked which replaces the current slice with a new slice. When zooming, the computation engine retrieves a new slice representing the closest edge to the center (which is where we are zooming into) and the slice is placed on a stack. When zooming out the corresponding slice is retrieved from the stack.

Views

A variety of visualizations can be used to display a given slice. A default is chosen automatically based on properties of the graph, but the user is presented with a list of visualization types that can be selected. If an alternate view is selected, the current visualization is substituted by the chosen replacement. Our system keeps track of the preferred view in case the user navigates to other slices and then returns to a slice. Moreover, several mechanisms are provided that allow the user to plug-in his/her own slice representation.

When multiple views of a slice are used simultaneously, they can be linked together. As the mouse passes over elements in one view, other views highlight the corresponding elements in their view.

Selection

The user interface allows for nodes to be selected with the mouse. A list of selected nodes is maintained by the system which can be used by different visualization methods. Typically, the selection is used to display a sub-graph of the current slice. For example, if we are displaying a graph whose nodes are all states in the US, we could

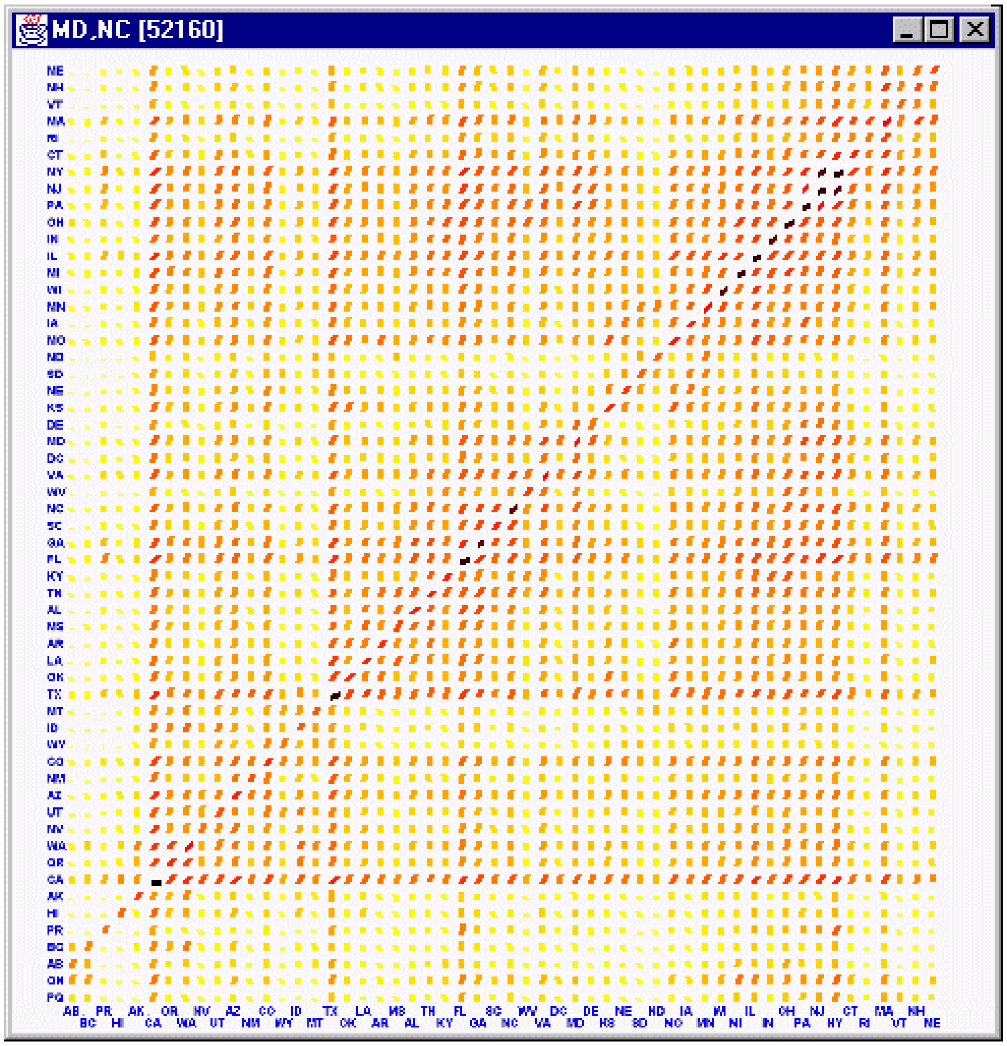

Figure 4: A graph slice represented as a needle grid. Edge values are shown with multiple cues: the segment color, segment length, and segment orientation.

select a handful of states we are interested and limit our display to only those nodes and related edges. When the selection changes on one view of a graph, it is appropriately updated on corresponding linked views.

Slice Computation

Our computation engine does not need to compute the entire $H(G,T)$ a priori, since it is likely that a user will only navigate through a subset of the data. Therefore, our engine runs in concert with the visualization interface and acts as a server. The interface starts off by requesting an initial slice from the server. This slice is converted to a visual representation, which is navigated by the user. If the user selects to zoom into an edge, the interface sends a request to the server to obtain a new slice. The engine can compute this slice on the fly, or simply return the contents of a precomputed slice.

4 Slice Views

This section describes some of the built-in visualization techniques that can be used to display graph slices. *MGV* provides a flexible interface for defining new visualizations so we are not limited to the set of views that we describe here.

MGV works with slices in their adjacency matrix representation. Slices are visualized as a set of line segments, where each matrix element maps into a single line segment whose origin, length, color, width, etc. depend on some mapping function f. In the simplest case, we can draw the elements onto a rectangular grid, but much more sophisticated mappings are possible.

Our system automatically tracks the correspondence between edges and visual segments. Thus, the author of a visualization does not have to handle the details of user interaction. The system can determine which edges are selected through the interface. It uses this information to interactively label edges and determine which edge is to be replaced and expanded when the user zooms in.

Currently, our visual metaphors are being used in the analysis of several large multi-digraphs arising in the telecommunications industry. These graphs are collected incrementally. For example, the AT&T call detail multi-digraph, consists on daily increments of about 275 million edges defined on a set containing on the order of 260 million vertices. The aim is to process and visualize these type of multi-digraphs at a rate of a million edges per second. We will use examples from this data to illustrate the metaphors presented in

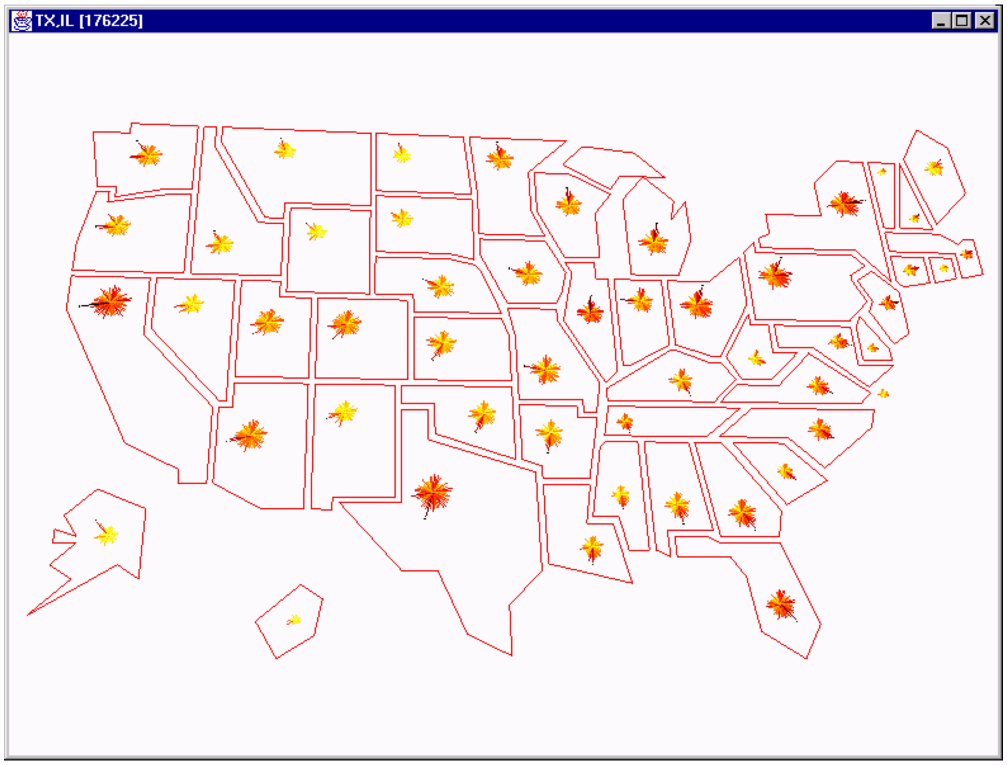

Figure 5: A star-map view of call data, superimposed with geographic information.

this section[2]; we describe other applications in Section 4.6.

4.1 Needle Grid

One way to view a slice is as a real non-negative matrix A whose entries are normalized in a suitable fashion. Each matrix entry $A(i,j)$ is represented as a vector $r(i,j)$ with origin at (i,j) and whose norm is obtained via a continuous and non-decreasing mapping n. The angle $ang(i,j)$ that $r(i,j)$ forms with the horizontal axis x is predetermined by the order of the entries in the matrix A. We constrain the range of $ang(i,j)$ to run between $-\pi$ and 0. One such possible mapping n is the one provided by your car speedometer except that now the needle increases in length as it rotates from $-\pi$ to 0. We refer to the vector $r(i,j)$ as the *needle* corresponding to the value $A(i,j)$.

A rectangular grid with the needles, representing the values $A(i,j)$, placed at their corresponding origins (i,j), is called the *needle-grid* representation of the given matrix or a *needle slice*. (see Figure 4 for an example). Note that the grid view for a particular graph is not unique. It depends on the ordering of the matrix elements.

For our set of phone call data in Figure 4, we can make some interesting observations. First, we see high values along the diagonal. This indicates a higher call volume for interstate calls in general. We have arranged the order of the matrix elements to conform to a Peano-Hilbert path through the US map. In this way, clusters around the diagonal correspond to country regions with high calling traffic. We can also observe asymmetries in the edge density and that could be areas with differing densities of AT&T customers. In general, patterns at higher levels of the hierarchy can be used as exploration guides at lower detail levels.

[2]Values have been changed in this paper to protect sensitive information.

4.2 Star Maps

The *star-map* view rearranges each row or column of our matrix into a circular histogram rooted at a single point. The histogram is arranged such that the first value is drawn at 0 degrees and values are evenly spaced such that the final value is drawn at 2π. This results in a star-like appearance. We refer to each element of a star as a *star segment*. Star segments have a length proportional to the value of the edge it represents. Additionally, the color of the star segment is dependent on the value to provide an additional visual cue.

Each star represents a row or column, depending on which type of star visualization is chosen. The position in which each star is placed is arbitrary; however, if available, we can make use of geographic data associated with each node in the graph. For example, suppose we are looking at call detail data, where each node in the slice represents a particular state. We could supply latitude and longitude for each node and arrange the stars on a USA map, as shown in Figure 5. In this case, we are placing the star representing the row (or column) j at the geographic position of j.

The star-map conveys a different type of information than the needle grid. It is particularly well suited to focus on a particular subset of vertices and detect easily among them those ones with higher or lower incoming or outgoing traffic. By moving the mouse over the segments, the corresponding vertex labels get activated. In the call detail data, we notice some states with one or two star segments that are larger than the others. Moving the mouse over the segments reveals which states these are.

4.3 Multi-comb

The multi-comb view can be thought of as an extension of the star map. With star maps, an entire row or column of the matrix is drawn such that it appears as a single object (in the shape of a star) but it represents a collection of values. Taking this a step further, we can turn an entire matrix into a "single" object by placing the collection of stars that compose the matrix on top of each other along the z axis and connecting the endpoints of the corresponding star segments. An example is provided in Figure 6. This single object represents an aggregate view of a graph with hundreds of million of edges.

An advantage of this view is that we can compare rows or columns depending if we look along the star segments at a particular z or if we look at all the z values for a particular star segment. When we consider all the z values for a single star segment, it resembles a comb, which is why we term this view the multi-comb view. This view is useful in providing animations of data set evolution.

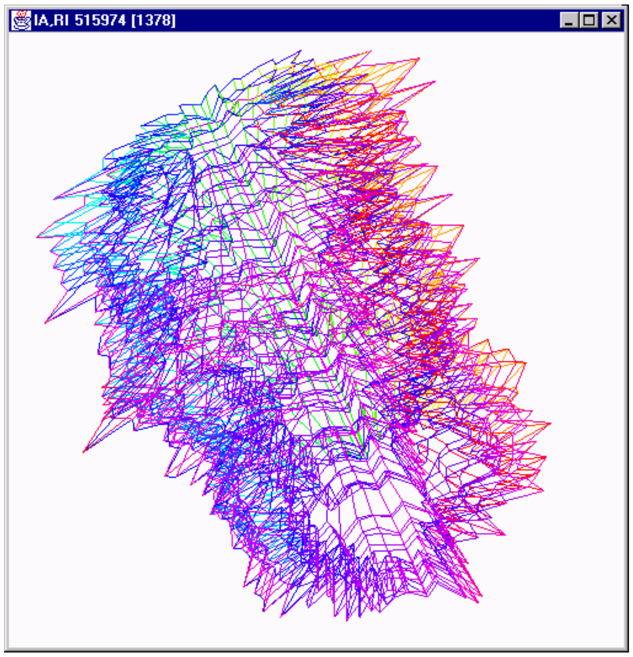

Figure 6: Multi-Comb View of call detail data at the state level.

4.4 Multi-wedge

The multi-wedge view is a different way to overlay stars on top of each other. Instead of putting each star at a different z value as we do with the multi-comb, we draw a single star as ticks instead of segments, where each tick is placed at the endpoint of that segment. The resulting picture, as shown in Figure 7, is a circular histogram with a distribution spectrum on each star segment, which we call a *wedge*. From this view, we can see the min and max values for a star line (which is a row or column), standard deviation, median, mean, etc. This is a two dimensional view, which is preferable to the multi-comb for static visualizations. The colors of the ticks represent the value of the back-edge in the multi-graph. When the matrix is symmetrical, the colors of ticks will occur in order. Thus, we can easily detect asymmetries with this coloring convention.

In our example, we can look at the calling distributions for each state. We again see that intrastate calling is typically a lot greater than interstate calling, but this view reveals the rest of the distribution varies a lot by state. Looking at the distributions can tell us which states have more regional calling patterns. For example, North Dakota makes a lot more calls to Minnesota than to any other state, but California has a more even distribution to the other states.

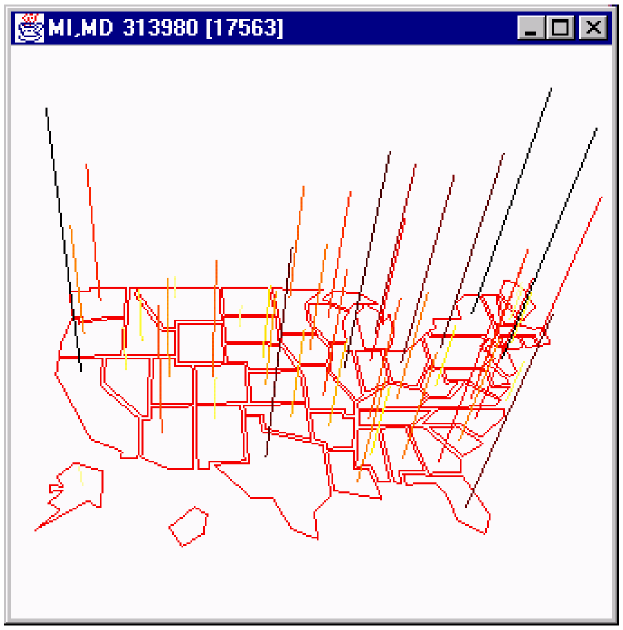

Figure 8: Aggregate view of the data represented in Figure 7.

We also see that the northern states of Idaho, Montana and North Dakota have lower phone usage than neighboring states.

4.5 Aggregate Views

Although we map each matrix entry to exactly one screen segment, we can create mappings which effectively compute certain aggregate operations. For example, suppose we are using the star map for a graph with associated geographic information and we want to replace the stars with a single bar representing their aggregate equivalent. We can accomplish this by creating bar segments for each star and placing them on top of each other along the z direction. The resulting view will appear as a single bar representing the sum of values for that row (or column), as shown in Figure 8. Additionally, a user can move the cursor on the bar to find out what are the segments that make up the bar, and can zoom in on a particular segment.

If we wish to do more complicated aggregations, such as taking the mean, median or an arbitrary function f over the values, we can accomplish this by mapping the slice into a new slice representing the aggregation and visualizing that slice. For example, if we wanted to visualize the average over each row, we would map an $m * n$ slice into a $m * 1$ slice. Our system provides a mechanism to define slice transformations, which are useful in other contexts as well. For instance, suppose we are only interested in a subset of the vertices. We can use a slice mapping to select out only the nodes we are interested in. We can also use transformations to rearrange the vertex ordering.

4.6 Applications

The navigation operations can be enhanced to perform a variety of statistical computations in an incremental manner. They can also be used to animate behavior through time. The stars-map metaphor is very useful when the vertices of the multi-digraph have an underlying geographic location (see Figure 5). This offers a high degree of correlation between graph theoretical information and the underlying geography.

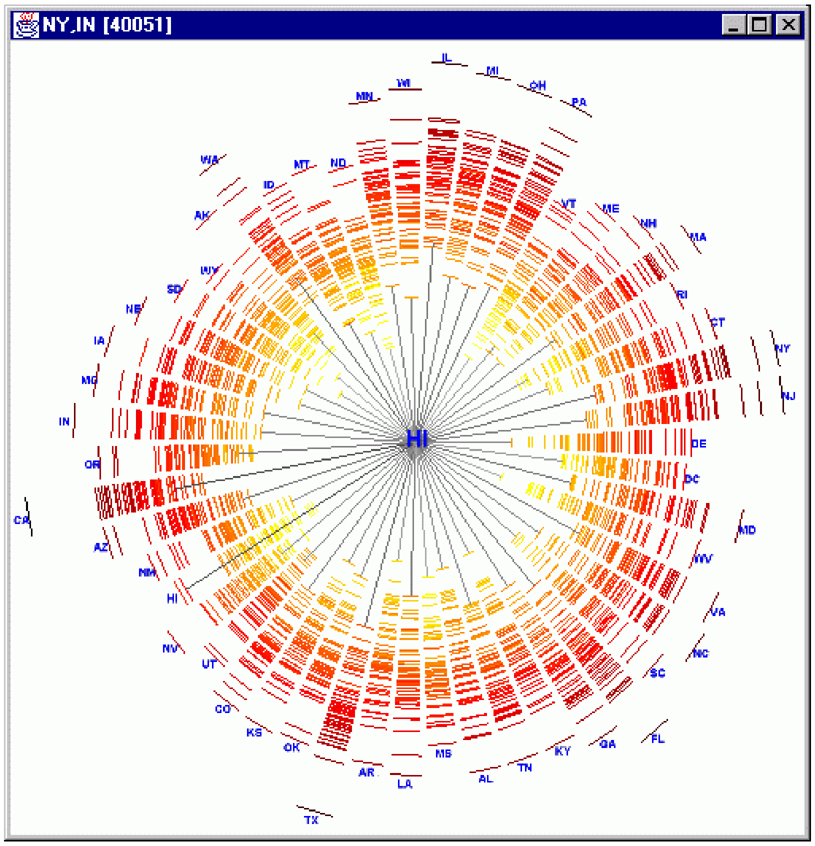

Figure 7: Multi-Wedge view of call detail data. Each wedge shows the distribution of calls for one state, and can be compared to the star of a particular state.

We currently have instantiations of *MGV* that visualize call detail data and network capacity data. We can work with a variety of other data sets as well; citation indexes, general library collections, program function call graphs, file systems and internet router traffic data are, among others, interesting data sets that can be explored using the approach described here.

Internet data is a prime example of a hierarchically labeled multi-digraph that fits quite naturally our graph metaphor. Each *i-layer* represents traffic among the aggregate elements that lie at the i^{th} level of the hierarchy (such as IP address blocks or the domain name space). We can also apply the techniques to web data. Considering pages as nodes and hyper-links as edges, we can take a set of web pages as a digraph. A portal such as Yahoo, which categorizes web sites into a hierarchy, could be used as T.

5 Implementation

As mentioned previously, MGV is separated into a computation engine and a Java-based user interface. The engine runs as a web server, and communication takes place using the http protocol. The server encodes slices as XML which are then processed by the interface. The use of Java-3D makes the system portable and allows fast rendering of visual representations, as it is able to take advantage of hardware graphics support. In the design of the interface, we had to make decisions on some interesting questions regarding the presentation of the various visualizations:

- How do we provide context to the user while he/she is exploring a node deep in the hierarchy?
- Typically, at each level, there are a few sites that are potentially interesting. How do we communicate this in the display and encourage them to explore deeper?
- Labeling is an important issue when displaying information. How can we avoid the problem of cluttering during the display of labels?
- How can we apply geographic information associated with the data?

In our display, we maintain context in two ways. We use one window to display a delayed view, with respect to zooming, of the user's view (see Figure 9). We highlight those data portions that have been visited already to provide users with information about the extent of their exploration.

The visualization engine tracks the mouse activity of the user and displays textual information about the closest edge in a separate window.

Potentially interesting regions (i.e. *hotspots*) are highlighted in a different color to catch the user's attention. An obvious limitation of the current approach is that what *is* and *is not* interesting from a data mining point of view must be pre-determined.

In order to handle textual labels in an efficient manner we divide the set of labels into two parts, static and dynamic. Static labels are displayed at all times. They are a small fraction of the entire label set. Dynamic labels are displayed only when the user selects them. The combination of static and dynamic labels manages the excessive clutter in the display well.

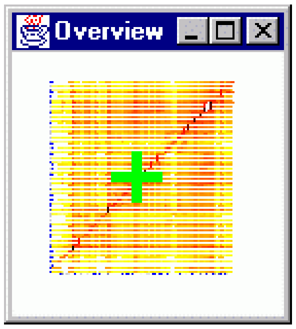

Figure 9: Overview window. The plus symbol shows the location in the parent slice of our current zooming position.

6 Conclusions

Needle-grids, star-maps, multi-combs and multi-wedges are the visual counterpart of the graph theoretical notions of edges and neighborhoods. They can be superimposed on an arbitrary layout of the vertex set of a graph without cluttering the view. They can be also used to visually represent certain type of aggregate statistics on multi-graphs. These facts coupled with a predefined hierarchy on the vertex set allow us to visually explore very massive multi-digraphs. The navigation is based on the notion of graph-slices. Graph-slices provide flexibility in terms of visual representations and visual navigation. The fact that the *MGV* client is implemented in Java3D helps make the system highly portable and extensible.

Our metaphor allows the integration of visualization and computation on a large class of massive data sets. It opens the door to the use of matrix theoretical methods for the hierarchical analysis of very large data collections. In particular, the pseudo-automatic selection of color maps depending of the statistical properties of the data at different levels of the hierarchy is one of the major issues that we are planning to address in the future.

Another natural direction to pursue is to come up with an efficient distributed memory implementation of *MGV*.

Acknowledgments

We thank S. Byers, D. Keim, S. Sudarsky and W. Sweldens for comments and discussions about this this work.

References

[1] B. Rogowitz, L. Treinish. A Rule-based Tool for Assisting Colormap Selection. In *Visualization '95 proceedings*, volume 444, pages 118-125, Oct. 1995.

[2] M. Chuah. Dynamic Aggregation with Circular Visual Designs. In *Proceedings IEEE Symposium on Information Visualization*, pages 35-43, 1998.

[3] M. Ankerst, D. Keim, H. Kriegel. Circle Segments: A Technique for Visually Exploring Large Multidimensional Data Sets. In *IEEE Conf. Visualization*, 1996.

[4] J. Abello, E. Gansner, E. Koutsofios, S. North. Large Scale Network Visualization. In *SIGGRAPH Newsletter*, Vol. 33, No 3, pages 13-15, August 1999.

[5] B. Rogowitz, L. Treinish. How not to lie with visualization In *Computers in Physics*, volume 10, pp 268, 1996.

[6] J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external memory graph algorithms. In *European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, pages 332–343. Springer-Verlag, 1998.

[7] S. Eick, G. Wills. Navigating Large Networks with hierarchies. In *Proc. IEEE Conf. Visualization*, pages 204-210, 1993.

[8] G. Wills. NicheWorks-interactive visualization of very large graphs. In *Proc. 5th Int. Symp. Graph Drawing, GD*, volume 1353 of *Lecture Notes in Computer Science*, pages 403-414, Springer-Verlag, 1997.

[9] J. Abello, S. Krishnan. Navigating Graph Surfaces. In *Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems*, P. Pardalos(Ed.), pages 1-16. Kluwer Academic Publishers, 1999.

[10] J. Abello, J. Vitter. (Eds) External Memory Algorithms. Volume 50 of the AMS-DIMACS Series on Discrete Mathematics and Theoretical Computer Science, 1999.

[11] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[12] C. Duncan, M. Goodrich, S. Kobourov. Balanced Aspect Ratio Trees and Their Use for Drawing Very Large Graphs. Lecture Notes in Computer Science, 1547:111-124, 1998.

[13] P. Eades, Q. W. Feng. Multilevel Visualization of Clustered Graphs. Lecture Notes in Computer Science, 1190:101-112, 1

[14] L. De Floriani, B. Falcidieno, C. Pienovi. A Delaunay-Based Method for Surface Approximation. *Eurographics '83*, pages 333–350, 1983.

[15] P. Heckbert and M. Garland. Multiresolution modeling for fast rendering. *Proceedings of Graphics Interface '94*, pages 43–50, May 1994.

[16] T. Munzner. Exploring Large Graphs in 3D Hyperbolic Space. *IEEE Computer Graphics & Applications*, 18(4):18–23, 1998.

[17] Y. Ansel Teng, Daniel DeMenthon, and Larry S. Davis. Stealth terrain navigation. *IEEE Trans. Syst. Man Cybern.*, 23(1):96–110, 1993.