MGV: A System for 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. It adheres to the Visual Information-Seeking Mantra: Overview first, zoom and filter, then details on demand. *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, color maps, multi-linked views, and a zoomable label based interface. This makes the association of geographic information and graph data very natural. To automate the creation of the vertex set hierarchy for MGV, we use the notion of graph sketches. They can be thought of as visual indices that guide the navigation of a multi-graph too large to fit on the available display. *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 [1].

Keywords: external memory, visualization, massive data sets, graphs, hierarchies.

1 Introduction

One of the great visualization challenges today is the representation and fluid navigation of complex systems [19]. These include the World-Wide Web [9], the Internet backbone [10], telephone call graphs [27], co-authorship and citation networks of scientists [16, 17, 14], the intersection graph of boards of directors of large companies [18], the topology of food webs [3, 4], electrical power grids, cellular and metabolic networks [5, 6, 7, 8] and the neural network of certain nematode worms [15]. There has been renewed interest in the study of the structure and dynamics of complex networks [2]. Processing, querying, exploring and visualizing these massive data sets pose a series of interesting computational and visual challenges.

We concentrate on data sets that have an underlying multi-digraph structure that is very large but of sparse density and low diameter. Usually, the application dependent information can be modeled as a special collection of edge attributes. Our focus is on the basic multi-digraph structure. This impacts the data set storage organization and the retrieval of its associated information. Geographic information systems, telecommunications traffic, World-Wide Web and Internet data are prime examples of the type of graphs whose navigation can be guided by our approach.

1.1 The bottlenecks

When visualizing massive data, two of the most fundamental issues are those associated with the *I/O* and *screen* bottlenecks [12]. 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

[1]Portions of this work were presented at the IEEE Information Visualization Conference, Salk Lake City, Utah, October 2000 [12]

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[2]. 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.

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 [27]). 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.

Having a hierarchical partition of the edge set is essential for processing but it is not enough for visual navigation. To ease the screen bottleneck what is required is a mapping from the edge partition to the available display. This amounts to a second level clustering of the vertex hierarchy T where each tree vertex has out-degree bounded by a parameter d that is display size dependent. This is one of the aspects that the notion of *graph sketches* encapsulates.

1.2 Approach

In order to deal in a unified manner with both the I/O and the screen bottlenecks, we base our work on a computational metaphor called *hierarchical graph slices* and on a corresponding visual metaphor called *graph sketches*. 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 edge browsing.

In order to use *edge slices* as an effective tool for visual navigation we require a mapping of a hierarchical partition of the edges of the input graph G into a hierarchical partition of the screen space. Each such a mapping is called a *Graph Sketch* [13]. Good graph sketches offer simple views of a very large graph macro-structure. These views are zoomable and parameterized by specific subgraph thresholds. When the obtained subgraph is small enough to fit on the available screen, the graph representation and its processing can be varied. These representations may include 2D needle-grids, Star-Maps, 2D surfaces in R^3 or conventional graph drawings. 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. When facing a dense subset of edges, we use color maps and 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 matrix and color map based techniques that are particularly helpful in visualizing dense slices. We also introduce some novel *sketch* representations of massive

[2]See J. Abello and J. Vitter [31] for a recent review of this subject.

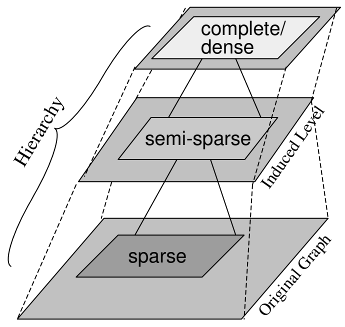

Figure 1: Graph layers of a hierarchical decomposition. Layers in deeper levels tend to be sparser.

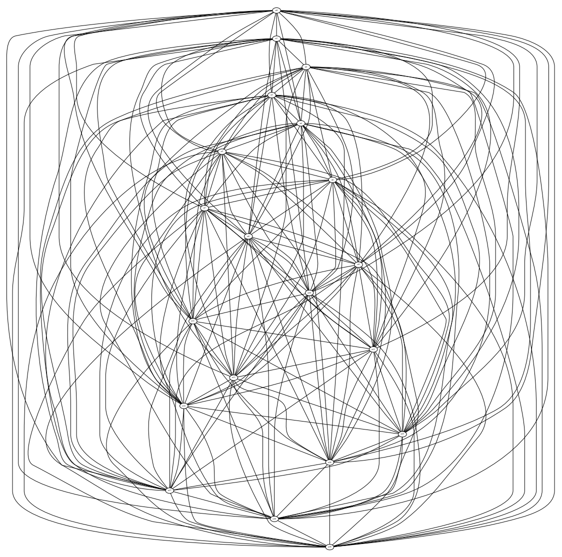

Figure 2: Traditional nodes-and-edges representation of a fully connected graph with 20 nodes.

multi-digraphs that can be used to drive their navigation. *Graph Slices* and *Graph Sketches* provide a unified view of computation and visualization on very large graphs. *Graph Sketches* are particularly useful when a pre-defined hierarchy tree T is not known in advance. They provide visual indexes that guide the navigation of a multi-digraph too large to fit on the available display.

After a vertex hierarchy T is computed, the corresponding graph-layers can be updated incrementally. They are suitable for the processing, navigation and visualization of external memory graphs [27] 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 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 and customizable color maps.

- 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 5.2) allow the superposition of graph neighborhood information on a given geography.

- Visual aggregation can be obtained by special views, such as our multi-comb view (Section 5.3) or by an adaptation of the circle of segments technique [25].

- The effective navigation of very large graphs is demonstrated by the use of *Graph Sketches* based on tree maps and orthogonal bars.

- Users can plug-in alternative visualizations of the hierarchical graph slices, and can apply their own filters to the slices.

1.3 Related Work

Drawings that display a graph completely have the advantage of showing global structure; however for large graphs such drawings become impractical. On the other hand, partial drawings allow the display of larger graphs but they fail to convey their global structure. Multi-level views offer the possibility of drawing large graphs at different levels of abstraction. The higher the level of abstraction, the coarser the provided graph view. Compound and clustered graphs have been considered in [33, 34, 35, 36]. The use of binary space partitions to produce graph clusters was introduced in [32]. However, the quality of the corresponding multi-level drawings depends heavily on the initial embedding of the graph on the plane.

Recently, force-directed methods have been considered for the drawing of large graphs. In [37], a hierarchy of subgraphs is associated with a large graph. One fundamental step of the algorithm is the use of the all-pairs shortest paths making its applicability to very large graphs limited since its space complexity is quadratic. The scaling of displacement vectors is based on an expensive Newton-Rapson optimization method.

In [38], some of the limitations of force-directed based methods for drawing large graphs are addressed. A central idea is to produce graph embeddings on Euclidean spaces of high dimensions and then projecting them into a two or three dimensional subspace. The method is based on a maximal independent set filtration of the vertexes of the graph and it is not apparent how to obtain, in an I/O efficient manner, such a filtration in the case of external memory graphs.

The work presented here grew out of the graph surfaces metaphor presented in [30]. 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 [42], Wills [29]). We present some methods for computing from the input graph, hierarchy trees that can be turned into efficient *Graph Sketches*. *Graph Sketches* can be viewed as visual navigation aids that help the user drive a computation towards a feasible set of answers.

The use of hierarchies for the exploration of large graphs is explicitly mentioned in [28]. 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 present the notion of *graph sketches* and in Sections 4 and 5, we discuss the correspondence between the slice hierarchy and the different visual representations. Section 6 contains a Breadth First Search based sketch that has been used to detect dense subgraphs

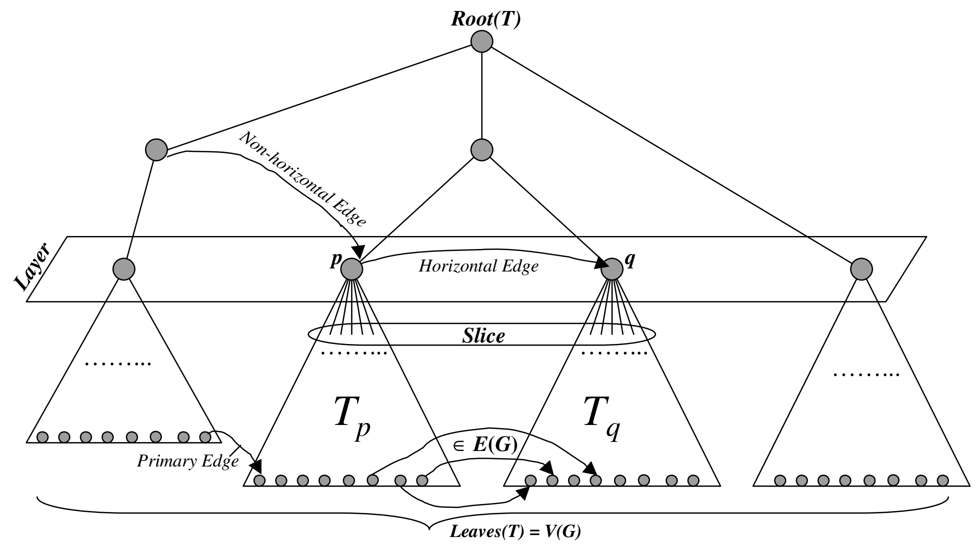

Figure 3: Hierarchical Graph Decomposition. Leaves of the tree are vertices of the original graph, and internal nodes represent information associated with the subgraph induced by its descendant leaves.

on very large graphs. The components of the Java-3D visualizer and the main interface issues are the contents of Section 7. Section 8 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 $m : E \to N$ 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* 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 [30], 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 a bottom-up fashion [30, 32] in time

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

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 ([27]) 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 Slice 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 via Graph Sketches

In order to visualize data sets with sizes of two or three orders of magnitude (typically around 250 million records) larger than the screen resolution (typically about one million pixels), it is imperative to use a decomposition of the visual space that reflects some structural view of the data. A *Graph Sketch* is a screen zoom-able macro-view of a very large graph. The goal is to use the *sketch* to guide the search for "interesting" subgraphs. The *sketch* should be tailored to the task at hand. For example, if the goal is to find dense subgraphs, the *sketch* needs to incorporate some notion of distance. This in turn, affects the type of recursive clustering that shall be used to define the sketch. In the applications section, we discuss a sketch that has been used successfully in the detection of dense subgraphs in very large graphs. In general, a good deal of ingenuity will be necessary to design *sketches* that become effective visual navigation aids. With this framework in mind, designing a good navigation sketch for a particular problem becomes the central algorithmic question that needs to be resolved before a useful interactive visualization can be proposed. In this context, visualization is no longer just a presentation aid; it becomes part of the computational process.

A *sketch* for a graph G is essentially a planar multi-digraph defined on a partition V_0,V_1,... V_k of $V(G)$ that is embeddable on the available pixel array. For a given graph problem P, if a solution on G can be obtained from the restrictions of P to the V_i's, then in principle one can use divide and conquer to search for a solution to P. The planar graph separator theorem ensures that for certain class of graph problems there are non-trivial sketches that guide the assembly of global solutions from local ones.

Given an algorithm that computes a *sketch* for a graph G, it can be used recursively to generate a tree T, such that $Leaves(T)$ represent a refinement of the original partition defining the *sketch*. This hierarchy tree T determines a hierarchical partition of $E(G)$. This means that a detailed view of an sketch edge can be obtained by zooming into it. In other words, from the initial planar embedding of the sketch one can zoom in locally into any of the edges. This locality provided by the planar clustering allows the user to explore the multi-digraph edge hierarchy in a fluid manner. Of course, all of this is possible if the detailed view of a macro-edge can be computed efficiently. We will discuss in a later section how and in what circumstances this is possible. We introduce next the $sketch$ related notions more formally.

3.1 Definitions of Graph Sketches

- A multi-digraph $G' = (V', E', m')$ is called a k-$view$ of a multi-digraph $G = (V, E, m)$ if V' is a partition of V with k subsets, where (u', v') is an edge in E' iff there exists u in u' and v in v' such that (u, v) is an edge in E. It is required also that $m'(u', v') = \sum_{(u,v) \in E(G)} m(u, v)$ for $u \in u'$ and $v \in v'$.

- For a given graph problem P, a k-$view$ of a multi-graph G is called P-*good*, if any solution to P on G can be efficiently computed from the solutions to P, on the subgraphs induced by each of the sets of the partition that define the k-$view$.

- A d-$sketch$ of a multi-graph $G = (V, E, m)$ is a planar embedding of a k-$view$ of G where k is no more than $\sqrt{d}$ and d is a display dependent parameter.

  The goal is to devise good d-$sketches$ for problems on very large graphs.

- Recall the definition of $LH(G, T)$ from the previous section. It consists of a collection of *layers* representing an equivalence relation on $E(G)$. In order to navigate $LH(G, T)$ we need to make sure that each node of the underlying hierarchy tree T has no more than $\sqrt{d}$ where d is the number of pixels on the display array D. This corresponds to computing a *d-sketch* for each sub-graph of $LH(G, T)$, that is obtained by zooming into a multi-edge of the form $(p, p, m(p, p))$. This computation only needs to be done in advance for the first couple of layers of $LH(G, T)$. A good mental picture is to think of transforming T into another hierarchy tree T' where the degree of each of its internal vertices has been reduced at the expense of higher depth. This in turn corresponds to having an equivalence relation on $E(G)$ that is a refinement of the original one. Call the corresponding multi-graph $LH(G, T')$. With this in mind, we start with a *d-sketch* of the first layer of $LH(G, T')$ and expand the self-loop multi-edges $(p, p, m(p, p))$ recursively leaving the remaining horizontal multi-edges to be computed on demand. In other words, the self-loops act as a basic coordinate system for the entire data set. When their corresponding *edge-slices* get computed, the fundamental data required by the other horizontal multi-edges is simultaneously pre-processed in preparation for future queries. Denote the collection of self-loops $(p, p, m(p, p))$ in $LH(G, T')$ by $SLH(G, T')$). Next we discuss how algorithms that compute $sketches$ can be used recursively to generate hierarchical partitions that are refinements of the original partition defining the $sketch$.

3.2 Constructing $SLH(G, T')$

Given a procedure **Construct-Sketch(G)** that produces a *d-sketch* for G with partition V_0,V_1,...,V_d, **Construct-Sketch** is invoked for each i on the subgraph induced by V_i. It is important to notice that all these invocations are independent of each other and that by the end of the computation of the *d-sketch*, only references are kept from each obtained multi-edge to the actual input data that it represents. Only the subgraph to be expanded needs to reside in memory. Care needs to be taken to carry with each call a mapping from the current vertex names to the local ones. The depth of the recursion is controlled by the number of available pixels d, a time or space budget, and problem defined parameters. When the recursion is finished a data structure representing the obtained hierarchy tree and a mapping from the tree leaves to the partition of $V(G)$ that they represent is produced. This data structure may reside in memory or on disk depending on the amount of available RAM. Notice that the complexity of constructing $SLH(G, T')$ depends strictly on the complexity of the procedure **Construct-Sketch** and on the quality of the obtained partition determined by T'.

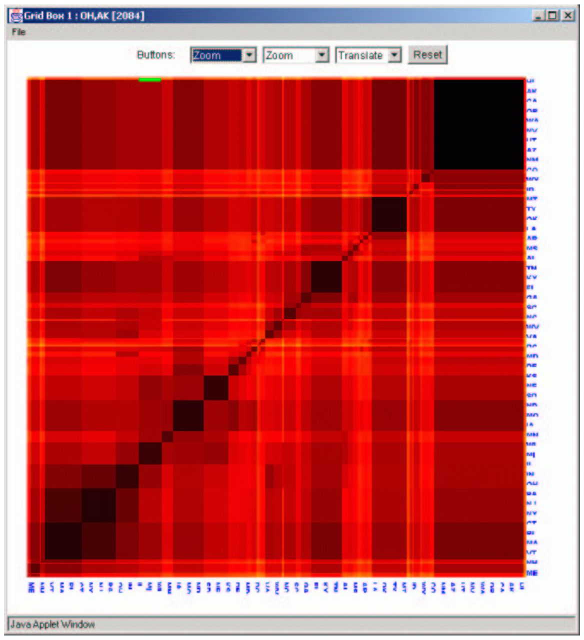

Figure 4: Location-based graph sketch. Each axis represents the states of the US, and color and size represent the density of phone calls made between pairs of states.

3.3 Computing the edge slice hierarchy associated with a horizontal multi-edge not in SLH(G,T')

Given a hierarchy tree T' and a disc resident index from multi-edges $(p, q, m(p, q))$ to their corresponding subgraphs $G([p], [q])$, the computation of the edge-slice hierarchy represented by $(p, q, m(p, q))$ is obtained by using the procedure **Construct-LH(G, T')** of *Lemma 1*. It takes as input a graph and a hierarchy tree and produces as output a multi-level index structure to the corresponding edge-slice hierarchy. This procedure runs in time

$$O(|V(G([p], [q]))| * Level((p, q)) + |E(G([p], [q]))|)$$

It is worth noting that this computation depends mainly on the level of the multi-edge. This suggests that a measure

of quality of a hierarchy for navigation purposes shall take into consideration the number of horizontal edges that are non-self loops at every level together with the size of the subgraphs they represent. So sparseness of each horizontal subgraph at each layer appears to be an important guiding principle. Criteria that involve the number of horizontal edges at each level together with the sizes of the subgraphs they represent can be used to devise algorithms that transform a hierarchy tree into another with the objective of improving the quality of the underlying vertex partition. We will address these and related issues in a future paper.

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 sketch hierarchy.

3.4 Sample Sketches

As discussed previously the main task to design a *good sketch* for a problem P is to devise a partitioning scheme of the input graph that guarantees that the space of solutions for P can be obtained by a suitable combination of the space of solutions of P restricted to the subgraphs induced by each set in the partition. Of course, this may not be the case for all problems and we know of no easily computable criteria to classify a problem as partitionable (in the sense described here). Nevertheless, we provide concrete examples of sketches for some NP-hard problems.

- The most direct example of a *sketch* comes from graphs whose vertices have associated a geographic location. A classical example is the graph whose vertices are telephone numbers and the edges consists of phone calls among them. In this case, the hierarchy T on the vertex set is pre-established and consists of the subdivision of the earth in continents, countries, states, counties, towns, etc. An embedding of the vertex hierarchy is provided by a cartographic map. The difficulty with this embedding is that higher level slices become very dense and if we want to draw the slice edges we are forced to live with edge crossings. The idea is not to draw the edges at all. In [12], a *star map* drawing was proposed, in order to place the underlying graph on top of the map embedding. An alternative view can now be provided by using a matrix based sketch. The rows and columns of the matrix are ordered according to a Peano-Hilbert ordering determined by the geographic position of the vertices. This allows for the visual correlation of the two views via multi-linking (see Figure 4). In summary, matrix based views correspond to a mapping of a partition of the edge set of G into a partition of the display. Edge crossings are not an issue anymore.

- Consider the problem of finding a largest cardinality clique in an arbitrary connected graph G. A *B*readth *F*irst *S*earch tree of G determines a partition of $V(G)$ defined by distances from the BFS root. The corresponding multi-graph is planar (in fact, ignoring directions, it is simply a path) and the number of sets in the partition is just the depth of the BFS tree. So the only condition that could fail for this multi-graph to be considered a *sketch* is that the depth of the BFS tree is larger than $\sqrt{d}$ where d is the number of available pixels. In this case, successive folding of the path can be done until it fits on the available screen. More generally, any planar k-view can be transformed into a related planar d-view where $d < k$.

  The assertion that a BFS based partition of $V(G)$ is a *good d-sketch*, for the maximum clique problem, follows from the observation that cliques of G are by definition induced subgraphs where all the vertices are at distance exactly 1. Therefore, cliques can span at most two consecutive levels of any BFS tree. A corresponding screen embedding is presented in the applications section(Section 6).

- The Network Decomposition Problem presented in [45] consists of finding a coloring of $V(G)$ with a distance parameter l such that:

  Each color class is partitioned into an arbitrary number of disjoint clusters,

  The shortest path distance between any pair of nodes in a cluster is at most l and

  Clusters of the same color are at least distance 2 apart.

  The goal is to find such a decomposition of a network where both the number of color classes and the distance parameter l are both $O(log(n))$ where n is the number of vertices in G.

  Despite the apparent similarity between this problem and the clique problem such a decomposition can be found in optimal time $O(|E| + n)$ by a simple greedy construction. This decomposition can be used as a base for a *sketch* but it is not clear for what class of graph problems this is a *good* sketch in the sense defined in this section.

4 Visual Exploration

We are able to explore very large graphs by starting with very simple structural macro-views and then navigating them via hierarchical *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 better 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 5, 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 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.

5 Slice Views

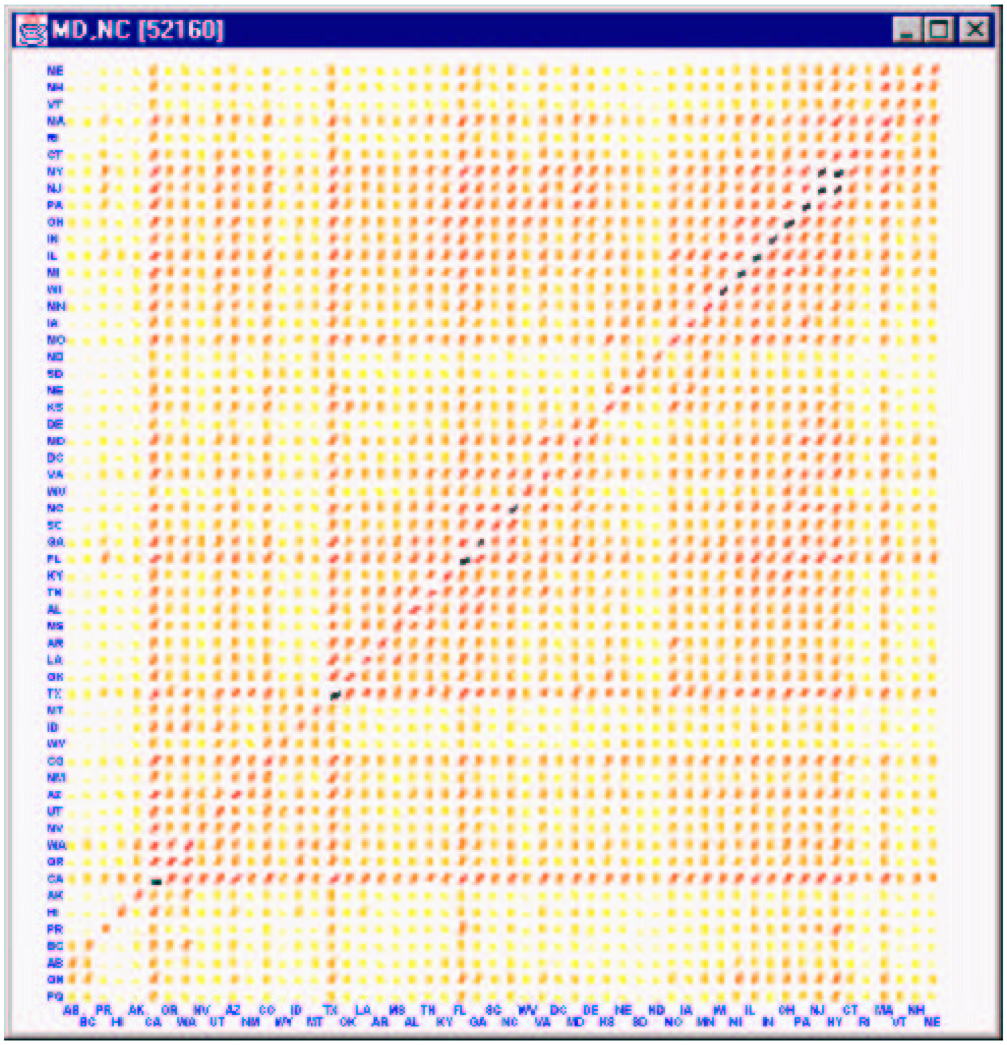

Figure 5: A needle grid view of call data. Each axis represents the states of the US, and density of phone calls made between pairs of states is represented with a needle with multiple visual cues: color, angle, and length.

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.

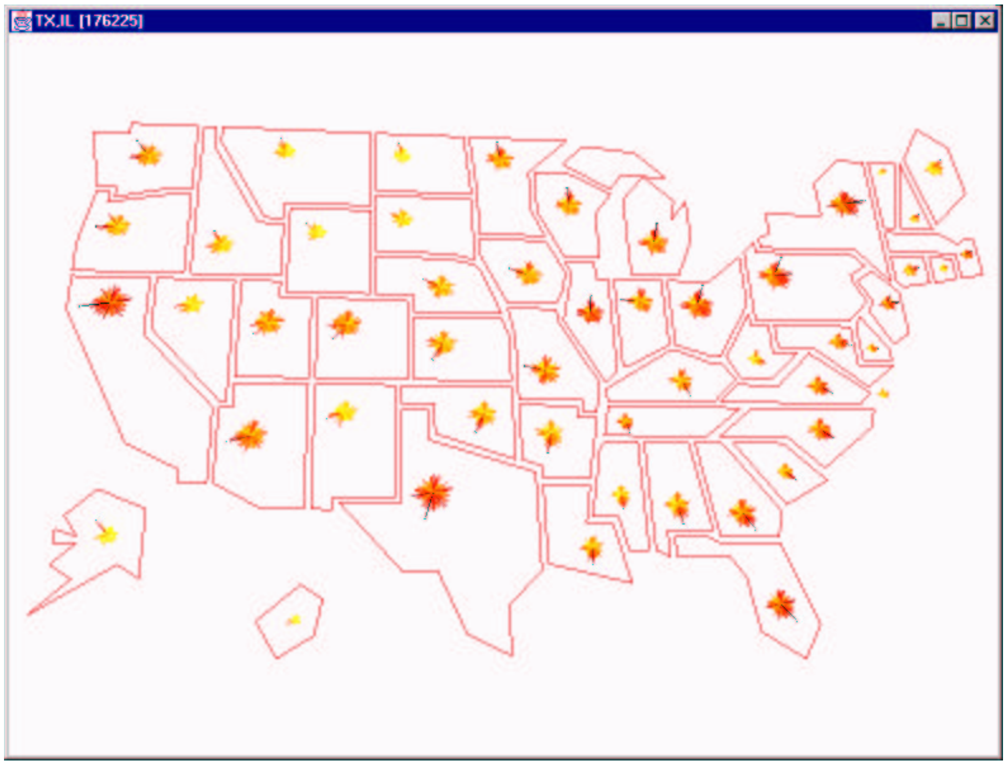

Figure 6: A Star-Map view of call data, superimposed with geographic information. For each state, a star is drawn, which consists of line segments that represent phone call density to each other state. The circular order of the states is the same for all stars.

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 this section[3]; we describe other applications in Section 6.

5.1 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

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

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 5 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 5, 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.

5.2 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 6. 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.

5.3 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 7. 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.

5.4 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 8, 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

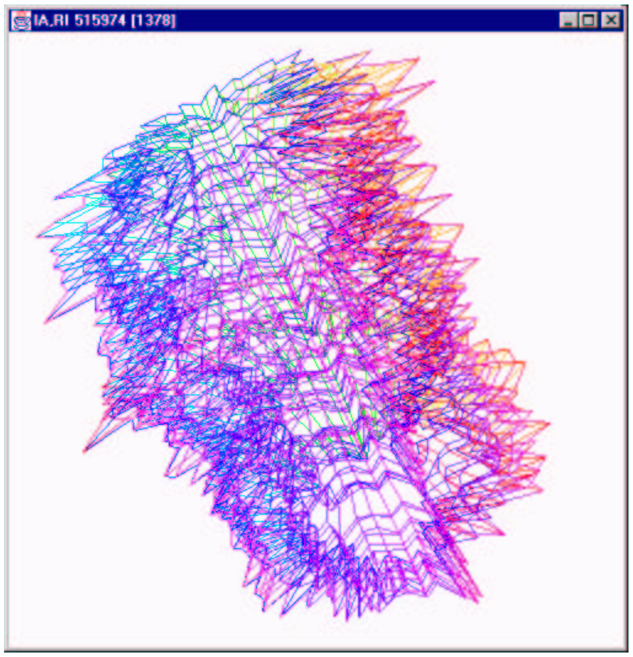

Figure 7: Multi-Comb View of call detail data at the state level. This can be thought of as stacking all of the stars in the Star-Map view of Figure 6 on top of each other to form a 3D volume.

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

5.5 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 9. 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.

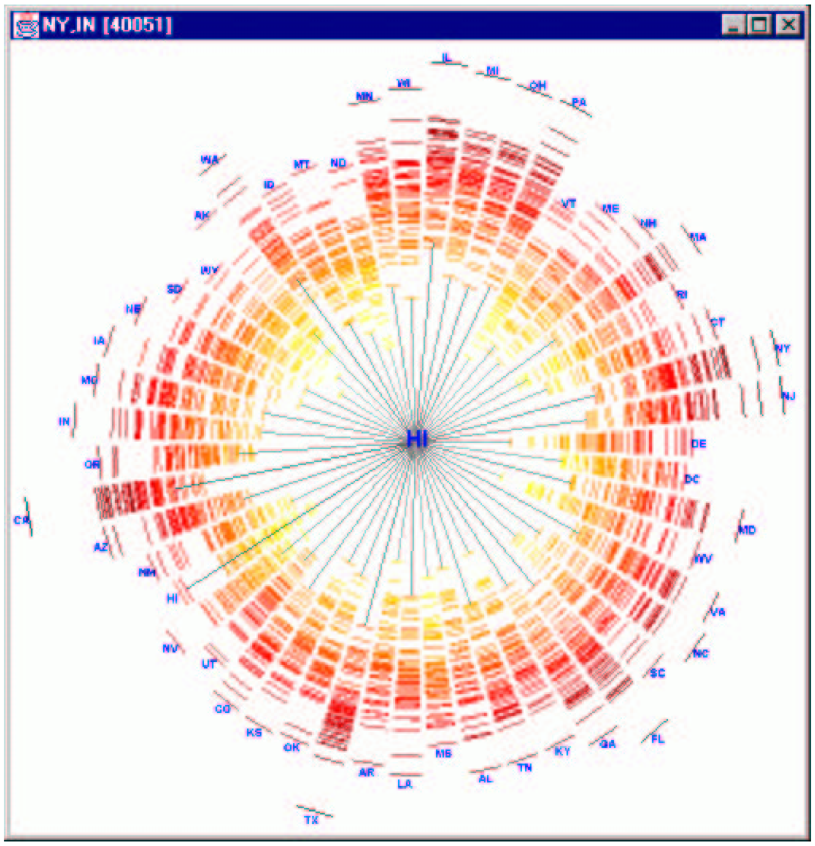

Figure 8: Multi-Wedge view of call detail data. Each wedge shows the distribution of calls for one state, and can be compared to the star of a particular state.

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 6). This offers a high degree of correlation between graph theoretical information and the underlying geography.

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.

We have also applied MGV to smaller data sets such as the year 2000 US Presidential Election results. Figure 10 shows a Multi-Wedge representation of the results at the state level. Such a view shows which states had close elections (such as Florida) and in which states third party candidates did well (such as Nader in Oregon).

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.

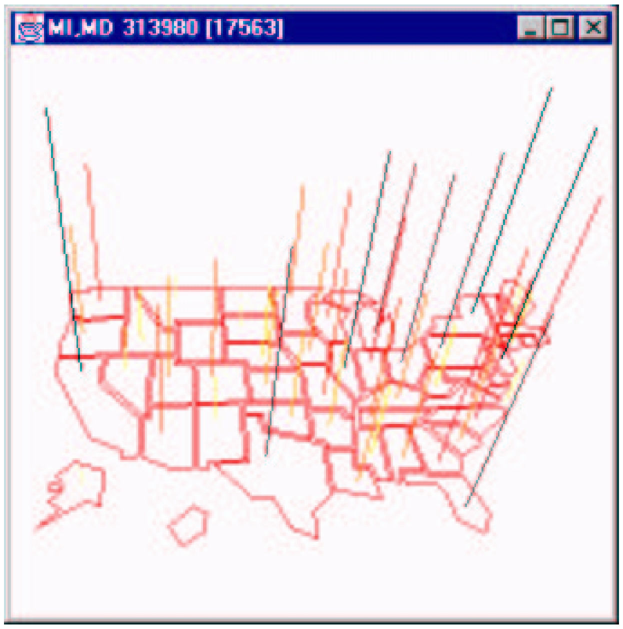

Figure 9: Aggregate view of the data represented in Figure 6. Instead of drawing star segments around a circle, they are stacked on top of each other.

6.1 A BFS sketch to detect dense subgraphs

As discussed in Section 3.4, a breadth first search partition of a large graph G produces a *good sketch*. This section describes experiences with such a sketch, as reported in [13].

We assume for the purpose of illustration that the diameter of the graph is smaller than the larger diagonal of the available screen space. This is not such an unreasonable assumption since it has been observed experimentally that call detail, Internet and web based data graphs [9] share some properties, namely, they are very large but sparse, with low diameter and very skewed degree distribution [10]. Moreover, a giant component emerges in a similar manner to the way that it emerges in random graphs ([20],[11]). This giant component has logarithmic diameter.

A natural question is then to detect on graphs with these characteristics large subgraphs with edge density above certain threshold t. Ideally one is interested in large subgraphs of density 1 (cliques). Since this problem is NP-complete one wonders if involving the user more directly into the exploration process could yield better dense subgraph detection. With this in mind, we set out to use a BFS based sketch augmented with a density color map to guide the search. The basic components of the interface are illustrated in Figures 11 and 12. One of the sketches consists of mapping each vertex of the hierarchy, to a box placed diagonally inside its parent's box with the side lengths of the two boxes being in the same proportion as the ratio of the cardinalities of their corresponding sets of descendant leaves. Because, the sketch is based on a BFS view of G, the subgraph consisting of the edges between consecutive levels, gets naturally assigned to the only adjacent boxes that are determined by consecutive boxes on the diagonal. Each box is painted according to a density based color map. When zooming on a box, its interior is partitioned according to its children and the color map is recomputed according to its children densities. The diagonal boxes corresponding to the leaves of the hierarchy tree can be though of as a coordinatization of the visual space. When a subgraph is detected that reaches the desired threshold, the subgraph representation is switched to a matrix based visualization (Figure 4). If more detailed connectivity is desired a conventional drawing representation can be invoked (Figure 12). The overview (Figure 15) is always present and a highlighted bar indicates the level, in the hierarchy tree, at which the exploration is taken place. Experimentally we have been able to detect, in call detail graphs, that the largest cliques also have logarithmic size.

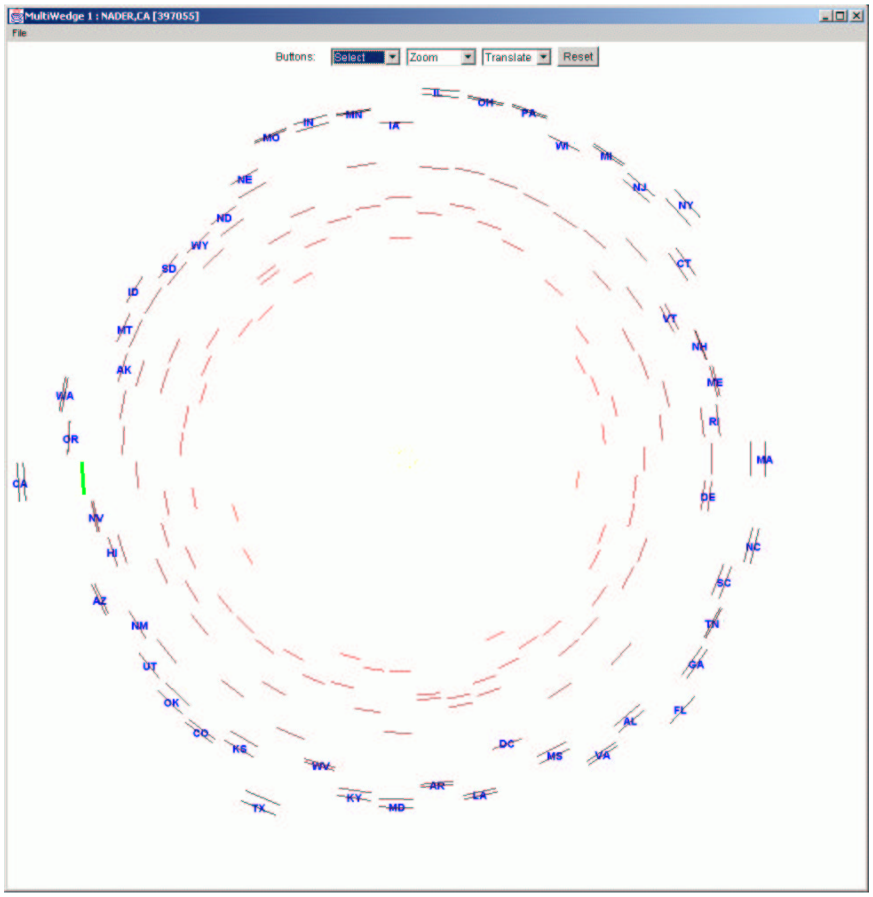

Figure 10: A Multi-Wedge view of the US Presidential election results. Each wedge shows the distribution of the election results for one state for the four major candidates. Notice that only three edges are visible in Florida's wedge because Gore and Bush's votes nearly overlap.

6.2 Sketch Maintenance

In order to effectively use $sketches$, the following pre-processing steps are necessary.

- Compute an external memory BFS. This can be done in $O((|V| + |E|/B) * log(|V|/B) + sort(|E|))$ *I/O*'s by using a modification of a data structure originally proposed by [46]. B is the size of the disk block.

- Build an in-core index to a disk resident data structure that contains for each level of the BFS its induced subgraph and for each pair of adjacent levels the subgraph consisting of all the edges going from one level to the other in both directions. The in-core index will only keep a reference to the disk location, the associated density function value and a few book keeping items. With this information, the corresponding screen embedding is computed as depicted in Figure 11. The corresponding portion of the current hierarchy tree T is also stored in memory. Now, for those vertices of the hierarchy tree whose associated induced subgraph fits in main memory the corresponding full hierarchy subtree is computed, using an internal memory implementation. Notice that all these computations can be made independently. For those vertices of the hierarchy tree whose vertex set fits in main memory but not its edge set, a semi-external version of BFS is invoked [31]. Those vertices of T whose associated vertex set does not fit in memory are processed again by a fully external BFS algorithm. Notice that all these computations are amenable to parallelization since they are independent. At the end of these steps, we have a disk resident representation of the hierarchy tree T and a mapping from its leaves to the actual vertices that they represent in the input graph.

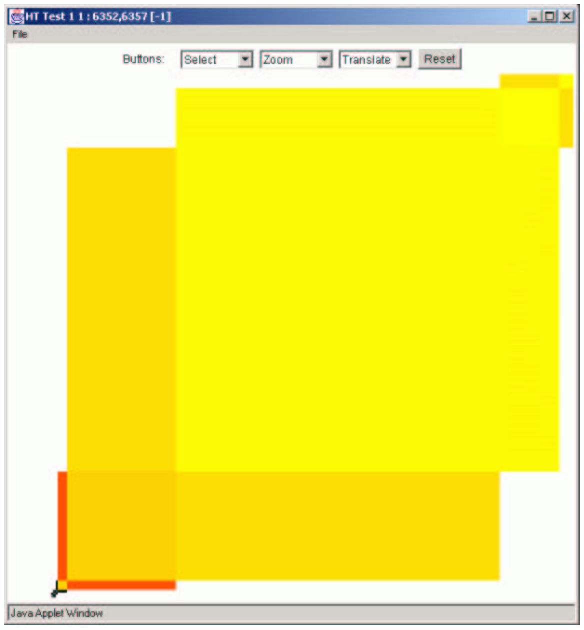

Figure 11: BFS Graph Sketch. The collection of boxes is a partition of the vertex set. The size of each box corresponds to the number of vertices in that set, and the color corresponds to the density of its induced subgraph.

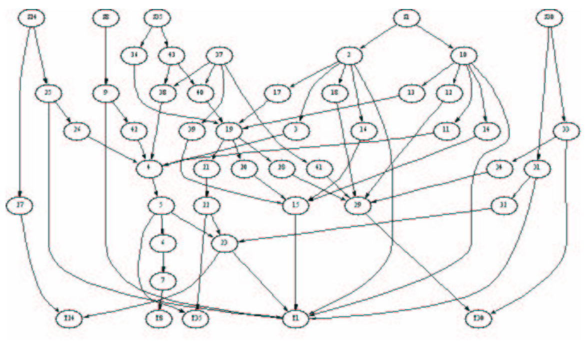

Figure 12: Traditional boxes and arrows graph

- A more economical sketch can be obtained by mapping each node of the hierarchy tree to a colored bar where the length is proportional to the size of its set of descendants leaves and where the color again encodes a map density. The collection of bars representing the set of children of a pair of bars are placed parallel to each other and in the order of their BFS levels. In the case of zooming into the children of just one bar its children are placed inside a zoomed version of the bar in a direction orthogonal to that of the parent bar. Initially, the root bar gets assigned a fixed but arbitrary direction.

  We refer to this BFS sketch embedding as the *orthogonal bars sketch*, as shown in Figure 13.

7 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,

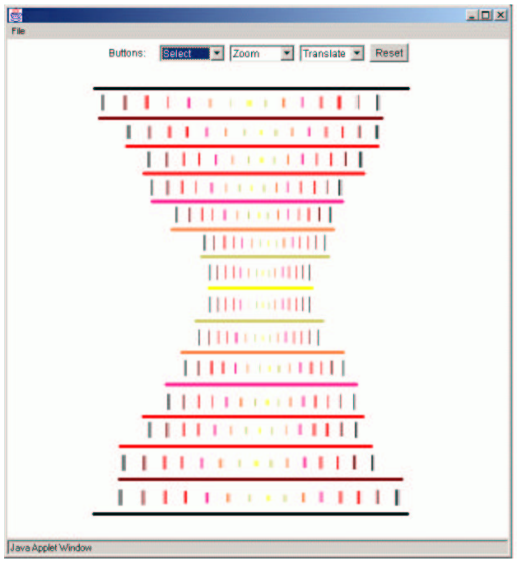

Figure 13: Orthogonal Bars Sketch. The collection of strips represents a partition of the graph. Each strip contains a collection of vertical segments representing a recursive partition of the subgraph induced by the strip. Color encodes graph density.

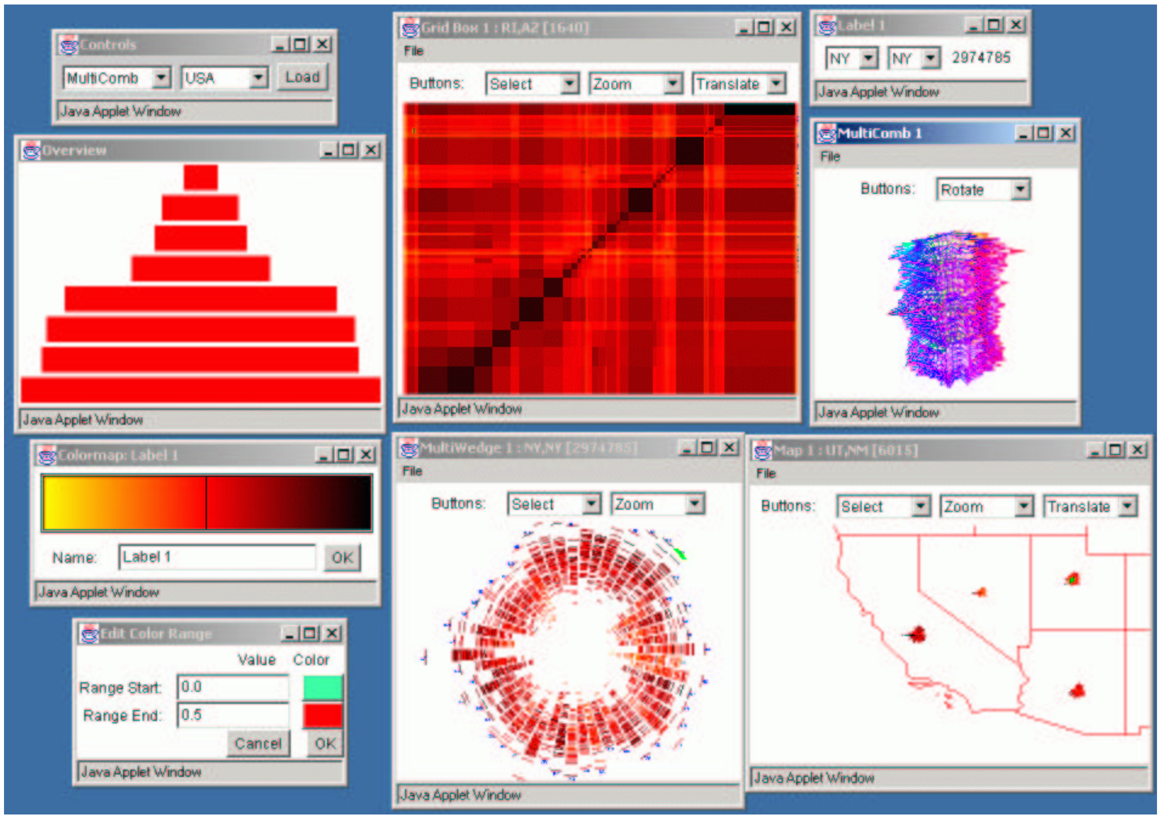

Figure 14: A screen dump of the MGV system showing some components and controls

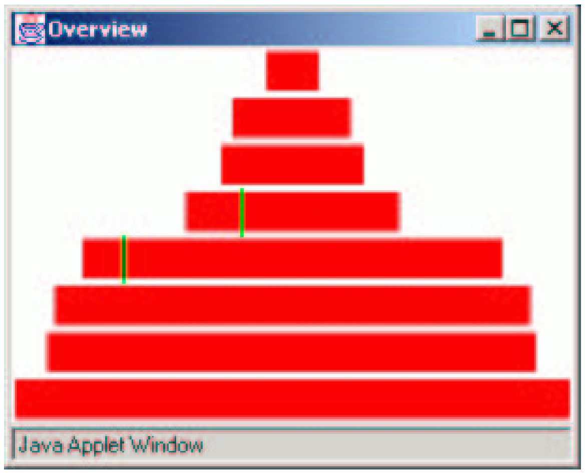

Figure 15: Overview Window. Each level of the hierarchical graph decomposition is represented with a bar. Markers are drawn in the view that correspond to the nodes in the hierarchy, representing the currently viewed slice.

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?

Figure 14 shows a running instance of MGV. Here, we see several MGV views, such as a Grid view, a Multi-Wedge, a Star-Map and a Multi-Comb. When the user first invokes MGV, two windows appear: a control window (shown in the upper left-hand corner), allowing a data set to be selected for viewing; and an overview window. The overview window maintains context and is shown in more detail in Figure 15. The overview window shows the vertex hierarchy, represented with horizontal bars, and marks the currently viewed slice.

The control window lists each of the view types, and lists each of the available data sets. Once the load button is pressed, a window appears showing the topmost slice in the hierarchy with the selected view. Using the mouse, the view can be rotated, translated, or zoomed. When the user zooms in far enough, a slice further down in the hierarchy replaces the currently viewed slice. In addition, a user can at any time change the view used to represent the current slice (e.g., from a grid to a map).

In a view, 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, as shown in the top right window (”Label”). Potentially interesting regions (i.e. *hot-spots*) 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.

Users can open an arbitrary number of views at once with MGV. Often, it is desirable to see the same edge in two different views which is why MGV provides linking functionality. Any set of views can be *linked*, which results in shared behavior between the views; as the mouse is moved in one view, the edge closest to the mouse is highlighted in all linked views, providing visual correspondence. When an edge in one view is replaced with a slice further down the hierarchy, all linked views do the same replacement.

Another important aspect of the user interface, described in more detail later in this section, is the ability to edit color maps. Each view contains a mapping between values and colors used to represent those values. For any view, the user can edit this map. This is shown in the bottom left windows of Figure 14. Changes made to the color map are immediately seen in the view that uses the map. Users can also link color maps of different views together so that a change in one map affects all linked windows.

The rest of this section walks through the MGV Java front end by examining the implementation. We give an overview of the classes, as it best illustrates our design choices.

7.1 MGVSlice

The most basic structure used is the `MGVSlice` class, which is a data structure that represents a slice. The slice is stored in an adjacency matrix representation. Consider the representation of the slice for edge $e = (v_1, v_2)$. We set up a grid where the rows are from 0 to j where v_1 has children $v_{1_0}...v_{1_j}$ and columns are from 0 to k where v_2 has children $v_{2_0}...v_{2_k}$. Each grid entry contains the weight for the appropriate edge, or is null if no such edge exists. We can map between rows or columns of the matrix to global vertex identifiers through the functions `rowToVertex`, `colToVertex`, `vertexToRow` and `vertexToCol`. The class also provides methods to get the label for matrix elements, get the latitude/longitude of a label, and read and write slices as XML from/to a disk or network. The summary of functions is below:

```
int getNumRows()
int getNumCols()
double getEdgeValue(int r, int c)
double maxEdgeVal()
double minEdgeVal()

String getRowLabel(int r)
String getColLabel(int c)
double getLat(String label)
double getLon(String label)

int rowToVertex(int r)
int colToVertex(int c)
int vertexToRow(int vertexId)
int vertexToCol(int vertexId)

static MGVSlice readSlice(int v1, int v2)
void writeSlice(OutputStream f)
```

7.2 MGVSliceView

Each view of a slice is defined as a subclass of the base class `MGVSliceView`. This class does all of the necessary book keeping and user interaction that all views share. Each new slice view needs to overwrite a small set of functions, shown below. The `drawImage` method is invoked to render the slice view as a `Geometry`, which is a Java3D primitive. A Geometry is an array of points that make up either lines, line strips, polygons or some other geometric 3D object. The methods `mapEdge` and `unmapEdge` are used to map between an edge and the position into the Geometry array that represents the edge. The function `filter` is called when a view of the slice is created, which is used to transform the incoming slice representation into a slice that the view uses. By default, the view can simply return the incoming slice, but it may choose to pick a subset of vertices to display, rearrange the vertices, or perform some other transformation.

```
// Functions that each view must overwrite
Geometry drawImage()                       // Draw image as Java3D Geometry
Geometry drawLabels()                      // Draw labels as Java3D Geometry
boolean inExpansionThreshold(double dist)  // Have we zoomed enough to expand?
MGVSlice filter(MGVSlice inSlice)          // Transform the slice

// Mapping functions implemented in base class
int unmapEdge(int v1, int v2)              // Convert edge to Geometry point
void mapEdge(int v1, int v2, int num)      // Map an edge to Geometry point
```

Filters

The class `MGVSliceFilters` provides a set of predefined filters that views can use. They are shown below.

- `static MGVSlice reverse(MGVSlice slice)`: Reverses the order of vertices in the slice.
- `static MGVSlice flip(MGVSlice slice)`: Transposes the adjacency matrix.
- `static MGVSlice merge(MGVSlice slice1, MGVSlice slice2)`: Combines two slices into a single slice.
- `static MGVSlice topN(MGVSlice slice, int percent)`: Filters out edges above the given percent threshold.
- `static MGVSlice sortByLabel(MGVSlice slice)`: Rearranges the order of vertices by sorting the labels.
- `static MGVSlice sortByLocCircle(MGVSlice slice)`: Rearranges the order of vertices using latitude and longitude to place vertices around a circle.
- `static MGVSlice sortByLocPeano(MGVSlice slice)`: Rearranges the order of vertices using latitude and longitude to correspond to a Peano-Hilbert tour of the vertices.

7.3 MGVInstance

Each new window is created using a class called `MGVInstance`. An instance shows a particular view of a slice at a given time. The user interacts with an instance by moving the mouse over the display, zooming, rotating or panning. When the mouse is moved over an edge on screen, it becomes highlighted. The user can link different windows together, which will cause the appropriate element to be highlighted in all linked displays. An instance also allows the user to expand or contract edges when the user zooms in far enough in or out (again, linked displays react accordingly).

```
MGVInstance(String name, MGVSliceView slice)    // Constructor
void link(MGVInstance inst)
void highlight(int v1, int v2)
ColorMap getColorMap()
void expand(int v1, int v2)
```

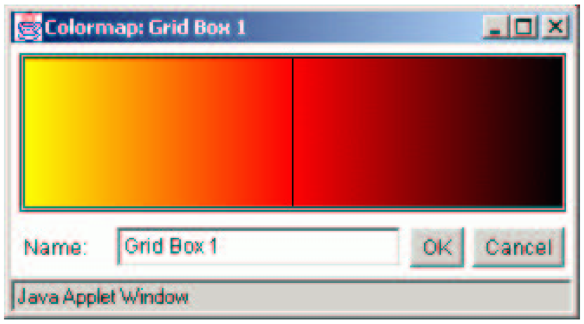

Figure 16: Color map editor user interface.

```
void contract()
void replace(MGVSliceView sliceView)
void destroy()
void setExpandMode(int mode)    // mode = {LEFT_EDGE, RIGHT_EDGE, BOTH_EDGE}
```

7.4 Colorings

MGV uses the weight of an edge to determine what color is used to visually represent that edge. The mapping between weights and colors is done by a separate module called the `ColorMap`. It is important to let the user control as much of the color mapping process as possible. The ColorMap class allows for this by providing a user interface where values can be mapped to colors. Reasonable defaults are used based on the slice's data. A side effect of the color editor user interface is that it is easy to perform certain visual queries. For example, if we are interested in looking at the top 10% of values, we can edit the color map such that the bottom 90% of values are mapped to a light grey color, and the top 10% map to a bright color. Applying this map will make the desired values stand out in the display, where they might not otherwise be seen.

Each view can have its own color map, or link to the same color map used by other views. Color maps can also be modified by the components themselves, in case they want to set their own default maps.

7.5 MGVManager

Finally, the `MGVManager` class brings everything together. This class provides the functionality to create new views or instances. It contains a static method called `registerView` which is called by each type of view that is in the system (typically in the static initializer of that class). By registering, a mapping is created between the name of the view (for example "MultiWedge") and the class that is used to render it.

```
static void registerView(String name, Class cls)
MGVSliceView createView(String viewName, MGVSlice level)
MGVInstance createInstance(MGVSliceView slice)
```

7.6 The Star-Map Algorithm

As mentioned previously, the *Star-Map* view rearranges each row or column of a slice into a circular histogram, resulting in a geographic map of histograms that resemble stars. In order for such a view to be useful, the order of the segments in each histogram needs to be chosen carefully. Due to the geographic nature of the underlying data, segments are arranged in a manner that has as much correspondence as possible to their underlying geographic values. MGV creates the appropriate ordering with the following algorithm:

- We are given a set of points that we wish to arrange along the circumference of a circle.

- We first compute the center of the set of points. This is done by simply averaging the set of points together.
- Next, we compute the convex hull for the set of points. All points on the hull are inserted into a priority queue with priority p.
- We compute the convex hull for the remaining points. All points on the hull are inserted into the priority queue with priority $p + 1$. This process is repeated until there are no more points.
- We then remove points from the priority queue from lowest priority to highest priority, choosing a position along the star that has an angle that is most similar to the angle formed between the center and the examined point.

This algorithm ensures that the outermost points are placed into the arrangement first, because those points are most noticeable and likely to be used as road maps. For example, for the set of points that represents states in the US, among the first states to be placed are Florida, Texas, Hawaii, California, Washington, and Maine.

8 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.

Graph Slices offer a unified view of computation and visualization on very large graphs. Very large graph visualizations need to be aware of the intrinsic algorithmic question that needs to be solved in order to provide interactive navigation that can guide a user towards the discovery of interesting graph sub-structures. Tailoring a graph decomposition to an exploration task appears to be an interesting angle that deserves further exploration. Devising useful 3D sketches is a tantalizing area of research. A question that comes to mind is: what should be the 3D counterpart to the planarity condition for 2D sketches? Given the fact that $sketches$ are mainly an abstraction of some of the properties of geographical maps, does it make sense to ask for the graph theoretical counterpart of curvilinear coordinates? What is a killer application that will benefit in concrete terms from these investigations? Are there any other interesting graph problems for which the BFS based sketches introduced here are beneficial?

Some of the metaphors proposed here open up 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 central aspect that deserves further scrutiny.

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

Acknowledgments

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

References

[1] B. Shneiderman. Information Visualization: Dynamic queries, starfield displays, and LifeLines. In *www.cs.umd.edu*, 1997.

[2] S. Strogatz. Exploring Complex Networks. In *Nature*, Vol. 410, No 8, pages 268-276, March 2001.

[3] J. Cohen, F. Briand, C. Newman. Community Food Webs: Data and Theory. *Springer Verlag*, Berlin, 1990.

[4] J. Williams, N. Martinez. Simple Rules yield Complex Food Webs. In *Nature*, Vol. 404, pages 180-183, 2000.

[5] K. Kohn. Molecular interaction map of the mammalian cell cycle control and DNA repair systems. In *Mol. Bio. Cell*, Vol. 10, pages 2703-2734, 1999.

[6] L. Hartwell, J. Hopfield, S. Leibler, A. Murray. From molecular to modular cell biology. In *Nature*, Vol. 402, pages C47-C52, 1999.

[7] U. Bhalla, R. Iyengar. Emerging properties of networks of biological signaling pathways. In *Science*, Vol. 283, pages 381-387, 1999.

[8] H. Jeong, B. Tombor, R. Albert, Z. Oltavi, A. Barabasi. The Large Scale Organization of Metabolic Networks. In *Nature*, Vol. 407, pages 651-654, 2000.

[9] A. Broder. Graph Structure in the Web. In *Networks*, Vol. 33, pages 309-320, 2000.

[10] M. Faloutsos, P. Faloutsos, C. Faloutsos. On power-law relationships of the internet topology. In *Comp. Comm. Rev.*, Vol. 29, pages 251-262, 1999.

[11] M. Molloy, B. Reed. The size of a giant component in a random graph with given degree sequence. In *Combinatorics, Probability and Computing*, Vol. 7, pages 295-305, 1998.

[12] J. Abello, J. Korn. Visualizing Massive Multi-Digraphs. In *IEEE Information Visualization Proceedings*, Salk Lake City, Oct. 2000.

[13] J. Abello, I. Finocchi, J. Korn. Graph Sketches. In *IEEE Information Visualization*, San Diego, CA, 2001

[14] P. Seglen. The Skewness of Science. In *J. Am. Soc. Inform. Sci.* , Vol. 43, pages 628-638, 1992.

[15] T. Achacoso, W. Yamamoto. AY's Neuroanatomy of C.elegans for Computation. In *CRC Press*, Boca Raton, FL, 1992.

[16] M. Newman. The structure of scientific collaboration networks. In *Proc. Natl. Acad. Sci. USA*, Vol. 98, pages 404-409, 2001.

[17] S. Redner. How popular is your paper? An empirical study of citation distribution. In *Eur. J. Phys. B*, Vol. 4, pages 131-134, 1998.

[18] G. Davis. The significance of boards interlocks for corporate governance. In *Corp. Govern.*, Vol. 4, pages 154-159, 1996.

[19] E. Wilson. Consilience In *Knopf*, New York, 1998.

[20] P. Erdos, E. Renyi. On the evolution of random graphs. In *Publ. Math. Inst. Hung. Acad. Sci.*, Vol. 5, pages 17-61, 1960.

[21] S. Strogatz. Nonlinear Dynamics and Chaos. *Perseus*, New York, 1994.

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

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

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

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

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

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] P. Eades, Q. W. Feng, X. Lin. Straight-line drawing algorithms for hierarchical and clustered graphs. In *Proc. 4th Symp. on Graph Drawing*, pages 113-128, 1996.

[35] Q. Feng, R. Cohen, P. Eades. How to draw a planar clustered graph. In *Proc. 1st Conf. Comp. and Comb., COCOON*, pages 21-31, 1995.

[36] K. Sugiyama, K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. In *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No 4, pages 876-892, 1991.

[37] D. Harel, Y. Koren. A fast multi-scale method for drawing large graphs. *TR MCS99-21*, The Weizmann Institute of Science, Rehovot, Israel, 1999.

[38] P. Gajer, M. Goodrich, S. Kobourov. *A multidimensional approach to force directed layouts of large graphs.* In *Proc. of Graph Drawing*, Lecture Notes of Computer Science, Springer Verlag, 2000.

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

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

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

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

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

[44] D. Karabeg. Parallel Algorithm Graph Reduction. *TR No. CS88-120*, University of California, San Diego, March 1988.

[45] L. Cowen. A linear time algorithm for network decomposition. *Dimacs TR series*, No 94-56, December 1994.

[46] V. Kumar, E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. *Proc. 8th IEEE SPDP*, pages 169-176, 1996.