Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems (P. M. Pardalos, Editor), pp. 1-16

Navigating Graph Surfaces

James Abello[1] (abello@research.att.com)
Information Visualization Research
Shannon Laboratories, AT&T Labs-Research, USA

Shankar Krishnan (krishnas@research.att.com)
Information Visualization Research
Shannon Laboratories, AT&T Labs-Research, USA

Abstract

A broad spectrum of massive data sets can be modeled as dynamic weighted multi-digraphs with sizes ranging from tens of gigabytes to petabytes. The sheer size of these data repositories brings with it interesting visualization and computational challenges. We introduce the notion of *graph surfaces* as a metaphor that allows the integration of visualization and computation over these data sets. By using out-of-core algorithms we build a hierarchy of graph surfaces that represents a virtual geography for the data set. In order to provide the user with navigation control and interactive response, we incorporate a number of geometric techniques from 3D computer graphics like terrain triangulation and mesh simplification. We highlight the main algorithmic ideas behind the tools and formulate two novel mathematical problems that have surfaced along the way.

Keywords: surfaces, graphs, massive data sets, out-of-core algorithms

[1] Portions of this work were presented at the International Congress of Industrial and Applied Mathematics, ICIAM, Edinburg, July 1999 and as a plenary lecture at the conference on Approximation and Complexity in Numerical Optimization, University of Florida, March 1999.

1 Introduction

Massive data sets bring with them a series of special computational challenges. Many of these data sets can be modeled as very large but sparse directed multi-digraphs with a special set of edge attributes that represent particular characteristics of the application at hand. Understanding the structure of the underlying multi-graph is essential for storage organization and information retrieval.

The substantial difference between CPU speeds and disk transfer rates makes the input/output(I/O) between main and external memories an increasingly significant bottleneck. Some suitable partitioning of the vertex and edge set helps to alleviate this bottleneck. As a first approximation to partitioning the underlying graph one could use external memory algorithms for computing its connected components [2]. However, it has been observed [1] in data sets collected in the telecommunications industry that, very soon, a giant component emerges suggesting that we may be witnessing a behavior similar to the one predicted by random graph theory even though the analyzed data sets are certainly not random. Therefore, even though connectivity based decomposition diminishes somehow the I/O bottleneck, in the general case we face a connected graph that does not fit in main memory.

We propose graph surfaces as a metaphor that unifies visualization and computation on weighted multi-digraphs. Several "natural" operations provide hierarchical browsing. This is achieved by mapping a multi-digraph to a hierarchy of surfaces. This gives flexibility in the handling of the I/O and screen bottlenecks. When a hierarchy is fixed, the corresponding graph surfaces can be updated incrementally. They are suitable for the maintenance, navigation and visualization of external memory graphs [2] whose vertex sets are hierarchically labeled. Examples include multi-digraphs arising in the telecommunications industry, internet traffic and geographic based information systems.

In the second section we provide a preview of graph surfaces. The algorithmic basis of the computational engine, its fundamental operations and I/O performance are discussed in sections 3 and 4. Section 5 discusses the transfer from multi-digraphs to height fields and offers some rationale for our approach. Section 6 provides the main elements of the height field triangulation algorithm being used to obtain a surface. The description of our scheme to visualize graphs that do not fit in memory and several intertwined interface issues are the contents of section 7. Section 8 discusses applications and conclusions and presents two interesting research problems that appeared along the way.

2 What is a Hierarchical Graph Surface?

The main idea is to view a weighted multi-digraph as a discretization of a two dimensional surface in $\mathcal{R}^3$. Under this view, and for a fixed ordering of the vertex set, the corresponding rectangular domain is triangulated and each point is lifted to

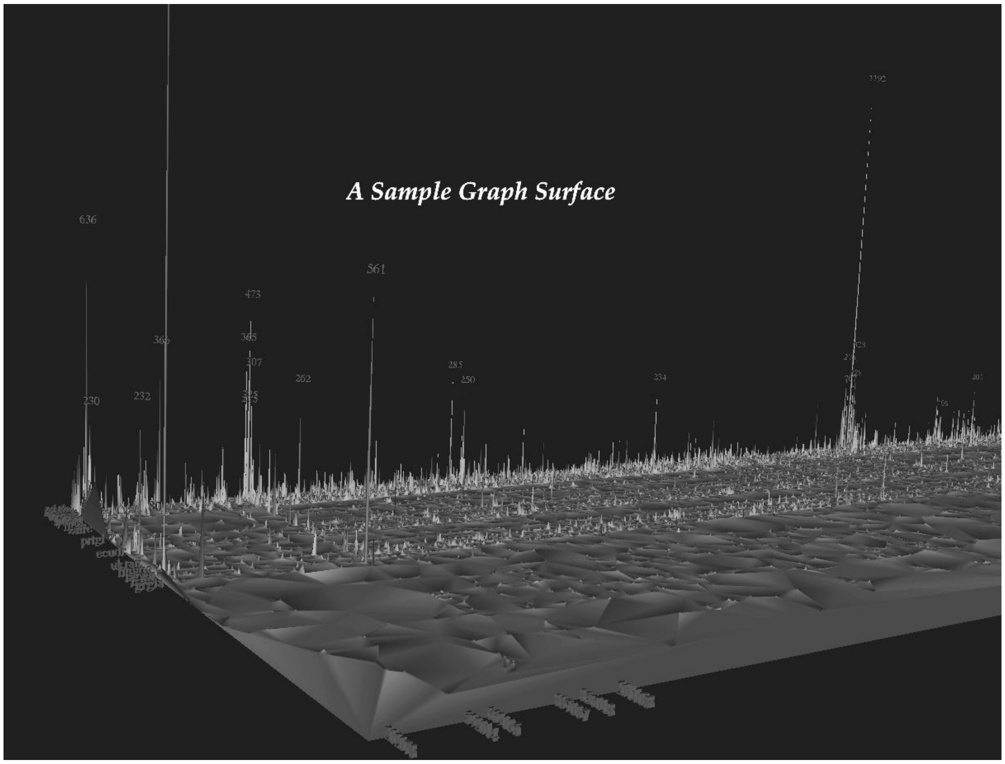

Figure 1: A sample graph surface.

its correct height. In this way, a piecewise linear continuous function is obtained (a polyhedral terrain). The polyhedral terrain is used as an approximation to a surface representing a multi-graph (see Figure 1 for an example).

In order to handle very large graphs a hierarchy of surfaces is constructed. Each of them represents a multi-digraph obtained from an equivalence relation defined on the edge set of the input graph. Operations are provided that allow the user to navigate the surfaces.

3 Problem Statement

3.1 Definitions

- For a rooted tree T, let $Leaves(T)$ = set of leaves of T (we will use $L(T)$ and $Leaves(T)$ interchangeably). $Height(T)$ = maximum distance from a vertex to the root of T; $T(i)$ set of vertices of T at distance i from the root of T. For a vertex $x \in T$, let T_x denote the set of proper descendants of x and let $Leaves(x)$ be the set of leaves in the subtree rooted at x.

- For a multi-digraph G, $V(G)$ 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.

- To simplify the exposition we concentrate on multi-digraphs. The adaptation to weighted multi-digraphs is straightforward. Similar to [8, 9], given a multi-digraph G = (V,E, $m : E \to N$) and a rooted tree T such that $Leaves(T) = V(G)$, the *i-slice* of G is the multi-digraph with vertex set $T(i)$ and a multi-edge (p, q) being defined if there exists (x, y) in $E(G)$ with $x \in T_p$ and $y \in T_q$. In the case that both x and y are nodes at the same level, the edge (x, y) is in the corresponding *slice* only in the case that $parent(x) = parent(y)$.

- The multiplicity of the edge (p, q) is $m(p, q) = \sum_{(x,y)\in E(G)} m(x, y)$ for $x \in T_p$ and $y \in T_q$. Notice that a 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 T and edge set equal to the edges of T union the edges on all the *i-slices* of G.

3.2 Constructing $H(G, T)$

Figure 2, contains pseudo-code describing the main steps needed to obtain $H(G, T)$ from G and T. The procedure **Construct** $H(G, T)$ 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 $H(G, T)$.

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

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

in a bottom-up fashion, [8].

Because $H(G, T)$ is really T plus the collection of slices of G given by T, each slice is represented as a two dimensional surface and T is used as a road map to navigate the surface hierarchy.

4 Navigating $H(G, T)$

In order to describe the navigation operations we make the following notational conventions. For a multi-digraph G, A a subset of $V(G)$ and F a subset of $E(G)$, we let G_A denote the subgraph of G induced by A, $E(A)$ is the set of edges incident to vertices in A and $V(F)$ is the set of vertices incident to edges in F.

Given two non-empty disjoint subsets A and B of $V(G)$, the *cut* between A and B is

$$cut(A, B) = \{(x, y) \in E(G) \ : \ x \in A, y \in B\}$$

```
Construct $H(G,T)$
{
    base_multigraph = $G$;
    $j = depth(T)$;
    for i=j downto 0 do
    {
        B(i) = E(T(i)); /*Edges in the base_multigraph incident to vertices in T(i)*/
        G(i) = subgraph of the base_multigraph induced by T(i);
        For every edge $(u,v)$ in $E(G(i))$ do
        {
          if parent(u) = parent(v) then
          {
            delete or mark (u,v) from the base_multigraph and from B(i);
            place (u,v) in E(i)
          }
          In the base_multigraph, "replace" all vertices in T(i) that have the same parent
          by their common parent, relabel the edges in B(i) accordingly and accumulate
          their multiplicities. For every edge created by this process keep a pointer
          to the original set of edges responsible for its creation.
        }
    }
    add the edges of $T$;
}
```

Figure 2: Construction of $H(G,T)$.

Given a partition $\Pi(V(G))$ of $V(G)$, $G[\Pi(V)]$ denote the multi-digraph with vertices the blocks of $\Pi(V(G))$ and edges (V_i, V_j) whenever $cut(V_i, V_j)$ is non-empty. The multiplicity $m(V_i, V_j)$ is the sum of the multiplicities of the edges in the $cut(V_i, V_j)$.

Lemma 2. For a multi-digraph G = (V,E, $m : E \rightarrow N$) and a tree T such that $Leaves(T) = V(G)$, if u is a vertex of T and children(u) = $\{u_1, ..., u_k\}$ then

$$G_{L(u)} = G_{L(u_1)} \cup G_{L(u_1)} \cup, ..., G_{L(u_k)} \cup E(G[L(u_1), L(u_2), ..., L(u_k)]).$$

Proof. The proof follows from the definitions and the fact that the collection $Leaves(u_i)$ is a partition of the set $Leaves(u)$.

The reason for stating the previous lemma is that it provides the base for the navigation of $H(G,T)$. Namely, the condition that $Leaves(T) = V(G)$ guarantee the existence of at least one i, such that T(i) determines a partition of $V(G)$ and every higher level is just a partial aggregation of this partition. This implies in turn that from any given slice one can move to any of the adjacent slices by refinement or partial aggregation of one set in the partition. This is precisely the information that is encoded in $H(G,T)$. Namely, from any given non-tree edge e in $H(G,T)$ one can obtain the set of edges 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. We introduce next, more formally, the navigation operations.

4.1 Navigation Operations

The main operations used by the computational engine are,

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

- Given a vertex u in T with children u_1, u_2, ... , u_k, *zoom*(u) is defined by {replace(u), generate G[L(u_1), ... , L(u_k)]}. An alternative view of *zoom*(u) is that it generates $\{(u, u_i)$: u_i is a child of u in T and (u_i,u_j) such that $cut(L(u_i), L(u_j))$ is non-empty}.

- Given an edge (u, v), *zoom*((u, v)) is defined as follows: {delete the edge (u,v); *replace*(u); *replace*(v); add cut({L(u_1), ... ,L(u_k)} , { L(v_1), ... ,L(v_k)}) from the graph G[L(u_1), ... , L(u_k), L(v_1), ... , L(v_j)].

One could define a restricted replacement of u and v to take only the vertices involved in the corresponding *cut* but this is only a conceptual minor point. Suitable inverses of the operations above can be specified 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 level and if they constitute all the children of a vertex u.

4.2 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. 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 slice. 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.

The third case is not well understood yet, but the current approach consists of fitting into main memory as many levels of T as possible (from the root downwards). The remaining levels are then collapsed into the deepest level (which does not fit) . An external parent array is made to point now from the deepest level into the last level residing in memory. A suitable index structure into the external parent array can then be used to filter the input edges (one level up) in one pass over the data. The rest of the filtering can then now proceed in main memory (as in case 1). The I/O performance depends strictly on the I/O efficiency of the access structure.

The increased availability of large RAMS makes it realistic to assume that the vertex set fits in main memory. For example, the essential information associated with 250 million vertices fits nicely in 2GB of RAM. In this case, in principle one can process "any" secondary storage multi-digraph with vertex set up to this size.

5 From Multi-digraphs to Surfaces

In this section, we formally introduce the notion of a *graph surface.* Consider a weighted, directed graph, $G = \langle V_1, V_2, E, w(E)\rangle$, where $V_1 \cup V_2$ form the set of vertices ($V_1 \cap V_2$ not necessarily empty), E is the set of edges in the graph from V_1 to V_2, and $w(E) \subset \mathcal{R}$ is a scalar function representing the weight of each edge. In other words, each edge e in the graph can be represented as

$$e = (v_1, v_2) \in E, v_1 \in V_1, v_2 \in V_2 \tag{1}$$

For the sake of completeness, we extend the domain of the function $w(.)$ to $V_1 \times V_2$ by setting

$$w^*(e) = \begin{cases} w(e) & \text{if } e \in E \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Consider two arbitrary injective mappings, $p_1 : V_1 \rightarrow \mathcal{R}$ and $p_2 : V_2 \rightarrow \mathcal{R}$. For every element $e = (v_1, v_2) \in (V_1 \times V_2)$, these two functions associate the unique three-dimensional coordinate, $(p_1(v_1), p_2(v_2), w^*(e))$. We refer to it as the *geographical position* of edge e. The surface passing through all the geographical positions of edges of a graph is its *graph surface.*

We make a few observations about graph surfaces.

- Each vertex of a graph surface represents an edge of the underlying graph. This is different from other graphical representations like stick diagrams which only map information associated with vertices.

- The graph surface for a particular graph is not unique. It depends on the functions $p_1(.)$ and $p_2(.)$.

- Every graph surface is a terrain (or a planar height field). This follows directly from the fact that the functions $p_1(.)$ and $p_2(.)$ are one-to-one.

Graph surfaces provide an interesting metaphor to visualize the underlying information in extremely large data sets arising from real-world applications. In our experience, these data sets have highly skewed distributions and this skewness is directly observed by the height difference in the graph surface. 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. One such example of a graph surface is shown in Figure 1.

We had mentioned earlier that the functions $p_1(.)$ and $p_2(.)$ must be chosen so that they form an injective mapping. We provide one simple example. Let the set $V = \{v_1, v_2, \ldots, v_n\}$ and consider a random permutation Π of the set $\{1, 2, \ldots, n\}$. Then the function p can be defined as

$$p(v_i) = a\Pi(i) + b, \tag{3}$$

where a and b are randomly chosen coefficients ($\neq 0$) which determine a translated coordinate system and the spacing between the vertices. It is immediate that $p(.)$ is an injective mapping.

Justification for the polygonal terrain metaphor: Currently, most information visualization systems use two-dimensional plots or three-dimensional stick figures to represent the data. A question naturally arises: why do we need to form an interpolatory or approximating surface between discrete data points? Points on the surface between data points have no obvious connection with the underlying data. This is a valid observation. However, when we are visualizing data sets of size two to three orders of magnitude (say around 250 million records) more than the screen resolution (typically about one million pixels), the distinction between discrete and continuous worlds vanish. Further, most of the current graphics hardware is fine-tuned to render polygons much faster than discrete line segments. If we require real-time interactivity in our application, it behooves us to use surface geometry.

6 Triangulation and Simplification Algorithm

In this section, we describe the algorithm we adopted to generate the triangulated surface of a height field. A height field is a set of height samples over a planar domain. Terrain data, a common type of height field, is used in many applications, including flight simulators, ground vehicle simulators, information visualization and in computer graphics for entertainment. In all of these applications, an efficient structure for representing and displaying the height field is desirable.

Our primary motivation is to render height field data rapidly and with high fidelity. Since almost all graphics hardware uses the polygon as the fundamental building block for object description, it seems natural to represent the terrain as a mesh of polygonal elements. The raw sample data can be trivially converted into polygons by placing edges between each pair of neighboring samples. However, for terrains of any significant size, rendering the full model is prohibitively expensive. For example, the 2,000,000 triangles in a 1000 × 1000 grid takes one to two seconds to render on current graphics workstations. More fundamentally, the detail of the full model is highly redundant when it is viewed from a distance. Further, many terrains have large, nearly planar regions which are well approximated by large polygons. Ideally, we would like to render models of arbitrary height fields with just enough detail for visual accuracy. To render a height field quickly, we can use multiresolution modeling, preprocessing it to construct approximations of the surface at various levels of detail [4, 13, 6, 5]. When rendering the height field, we can choose an approximation with an appropriate level of detail and use it in place of the original. In most general visualization and animation applications, the scene being simplified might be changing dynamically. Finding a simplification algorithm that is very fast is therefore quite important.

Problem Statement: We assume that a discrete two-dimensional set of point samples S of some underlying surface is provided. This is most naturally represented as a discrete function where $S(x, y) = z$. The final surface will be reconstructed from S by triangulating its points. We can think of obtaining the final surface using a reconstruction operator Γ which takes a discrete function defined over a continuous domain like S and maps it to continuous function over the same domain. If S' is some subset of input points, then $\Gamma(S')$ is the reconstructed surface, and $\Gamma(S'(x, y))$ is the value of the surface at point (x, y). Our goal is to find a subset S' of S which, when triangulated, approximates S as accurately as possible using as few points as possible, and to compute the triangulation as quickly as possible. We denote the number of input points in S be n. The number of points in the subset S' is m, and consequently the number of vertices in the triangulation is also m.

Approach: The principal class of algorithms that we discuss in this paper are refinement methods. Refinement methods are multi-pass algorithms that begin with an initial approximation and iteratively add new points as vertices in the triangulation. The process of refinement continues until some specific goal is achieved, usually

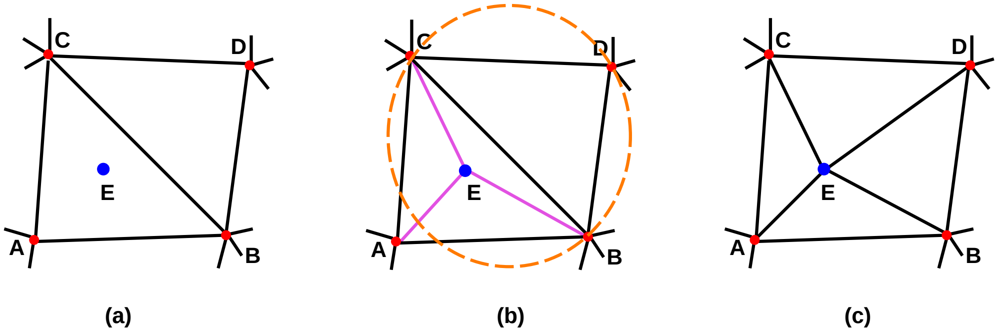

Figure 3: Incremental Delaunay triangulation (a) Point E is to be inserted inside triangle ABC. Edges EA, EB and EC are added. (b) Edge BC is a possible invalid edge and is checked using the circle test. Circumcircle of BCD contains E, so BC is replaced by ED (c) Now edges DB and DC are possibly invalid. [Garland95]

reaching a desired error threshold or exhausting a point budget. In order to choose which points to add to the approximation, refinement methods rank the available input points using some importance measure like local or global error estimates. We use local errors to perform this ranking primarily because it is cheap and in our experience, does not give significantly worse results than global error estimates.

6.1 Triangulation Algorithm

The basic algorithm to generate the triangulation of the height field is called a *greedy insertion* strategy. Many variations of this algorithm have been explored over the years [11, 10, 14, 7]. We have decided to use the Delaunay triangulation approach proposed by Garland et. al. [12]. The main reason for this choice is its ease of implementation and excellent performance on most of our test cases.

We begin with some basic functions that query the Delaunay mesh and perform incremental Delaunay triangulation. We build an initial approximation of two triangles using the corner points of S. Then we repeatedly scan the unused points to find the one with the largest error and add it to the current triangulation. The insertion procedure locates the triangle containing the given point, splits the triangle into three, and then recursively checks each of the outer edges of these triangles, flipping them if necessary to maintain a Delaunay triangulation (see Figure 3).

Each step of the insertion procedure affects only a local area of the triangulation. This implies that new errors have to be computed only for those unused points that lie within this local area. This local update significantly speeds up the computation at every insertion step (from linear time to almost constant time for sufficiently random distribution of points).

Further, a naive implementation of selecting the unused point with maximum error has linear time complexity. By introducing a slightly sophisticated data structure like a priority queue, we can reduce the selection to a constant time operation and all updates in the queue (due to insertion step) take logarithmic time.

These improvements in the algorithm reduce the worst case time complexity from $O(m^2n)$ to $O(mn)$, and the expected time complexity reduces from $O(mn)$ to $O((m+n)\log m)$. The memory usage for this algorithm is proportional to $(m+n)$. A detailed analysis of the time and space complexity can be found in [12].

7 Visual Navigation of Graph Surfaces

We now describe our scheme to visualize very large data sets in the form of digraphs. In this context, *large* refers to data sets that do not fit into main memory. Our system consists of two main components: *the computational engine* and *the graphical engine*. Given a large graph as input, the computational engine uses the approach described in the previous sections to cluster subgraphs together in a recursive fashion and generates a hierarchy of weighted multi-digraphs. The graphs in each node 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 (specially if there is human interaction involved). In our metaphor, we have decided to *amortize the visual content* in every scene by constructing the graph hierarchy. Further, the reduced size of individual data sets at every node also provides the necessary *real-time feedback* in such an exploratory setting. As the user traverses deeper into the hierarchy, the scene displayed becomes more and more detailed in a restricted portion of the data set.

The graphical engine has two primary functions - generating graph surfaces from individual nodes in $H(G,T)$ using the triangulation algorithm described in the previous section, and displaying them with appropriate visual cues and labeled text to provide the user with intuitive understanding along with complete navigational control.

7.1 Implementation

In this section we briefly describe some of the issues that arose when implementing the graphical engine. The system was implemented in C++ and uses the OpenGL standard library for the rendering part. Currently, the system uses a mouse/keyboard input interface, but we are exploring the use of joysticks and gestures to navigate the environment.

We had to make a few decisions on questions regarding the display of graph surfaces.

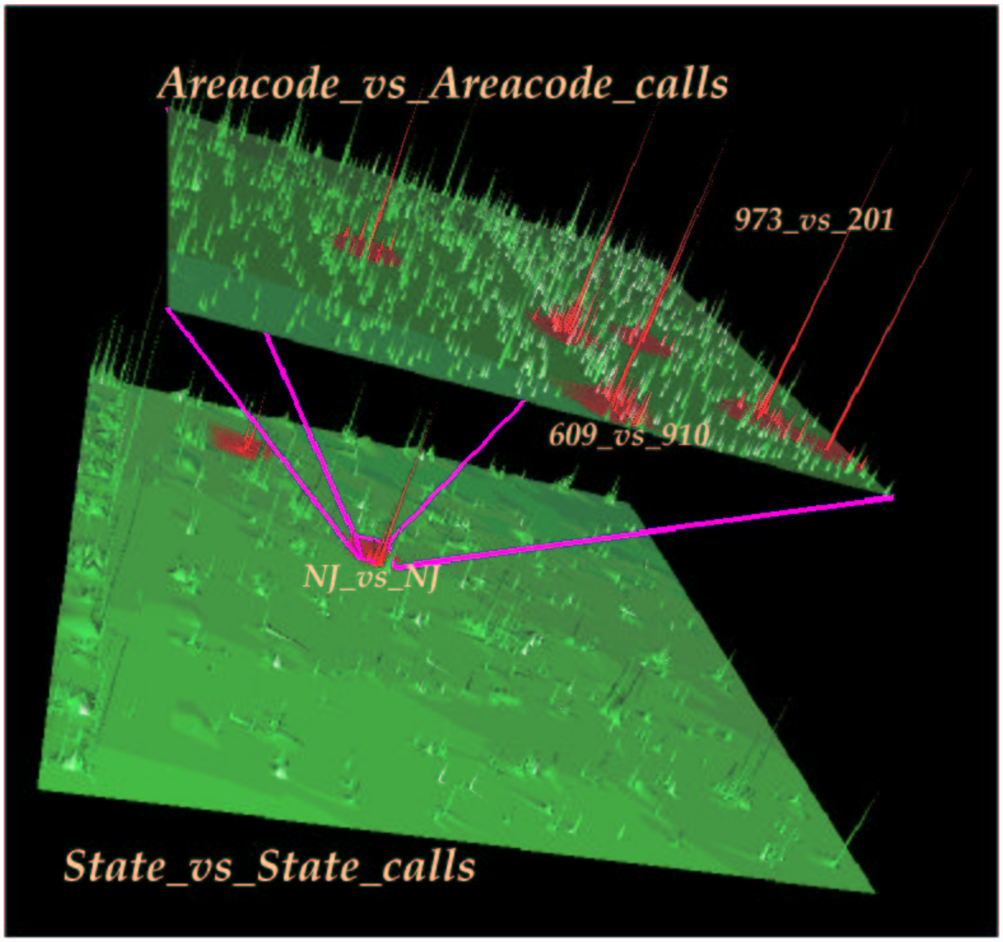

Figure 4: A snapshot from the display during a trial run of the system on phone call data. The size of this data is close to 250 million records and the hierarchy generated is 10 levels deep.

- How to 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?

In our display, we maintain context in two ways. We use one window which displays the hierarchy abstractly in the form of a tree and highlights nodes that the user has already visited or is currently visiting in different colors. This provides users with information about how deep they are in the hierarchy. In the main window, we augment the graph surface of the current node with the graph surface at the root of the hierarchy and show the region at the root that has been expanded to the current detail (see Figure 4). Further, the engine tracks the mouse activity of the user and also displays textual information about the closest vertex (edge of the original graph) in a separate window.

Potentially interesting regions in a node (i.e. *hotspots*) are highlighted in a different color to catch the user's attention. An obvious limitation of the current approach is that the computational engine pre-determines what *is* and *is not* interesting from a data mining point of view. However, given that most of the computation takes place out-of-core, dynamic hierarchy generation cannot be achieved without sacrificing interactivity in the visual feedback.

Finally, the problem of textually labeling large data sets is a well studied problem in information visualization and graph drawing . The approach we have taken is fairly simple. At each node, we divide the set of labels into two parts - static and dynamic. Static labels are displayed at all times. They are usually a very small fraction of the entire label set. Dynamic labels are displayed on the screen only when the user is interested (currently uses mouse tracking; some techniques like retina tracking are being studied by other researchers). The combination of static and dynamic labels elegantly manages to avoid excessive clutter in the display.

8 Applications and Open Questions

Currently, graph surfaces are being used experimentally for 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 300 million vertices. The aim is to process and visualize these type of multi-digraphs at a rate of a million edges per second.

Internet data is another prime example of a hierarchically labeled multi-digraph that fits quite naturally the graph surfaces metaphor. Each *i-slice* represents traffic among the aggregate elements that lie at the i^{th} level of the hierarchy. The navigation operations can be enhanced to perform a variety of statistical computations in an incremental manner. These in turn can be used to animate the traffic behavior through time. When the vertices of the multi-digraph have an underlying geographic location they can be mapped into a linear order (using for example, Peano-Hilbert curves) in order to maintain some degree of geographic proximity. In this way, the obtained surface maintains certain degree of correlation with the underlying geography.

We want to mention in closing the following mathematical questions that surfaced along this investigation.

8.0.1 Matrix Smoothing Problems

For a given $m \times n$ matrix A with real non-negative entries, let $\mathcal{P}(A)$ denote the set of points $\{(\mathrm{i}, \mathrm{j}, A(i,j)\}$ where i and j index the rows and columns of A, respectively. One can then consider different surfaces that fit the set of points $\mathcal{P}(A)$. The surfaces that we are interested in do not have to interpolate the points. We just want a "good" approximation. We want to allow row and/or column permutations of A in order to find a *"better fitting"* surface. Different versions of the problem can be formulated if we permute columns and rows simultaneously or if we permute them independently. We formulate below a restricted version of the problem for the case of square matrices and suggest area as a reasonable surface fitting criteria. To make the problem more tractable we can fix a surface approximation method $\mathcal{F}$, and denote by $S_{\mathcal{F}}(A)$ the corresponding surface produced by $\mathcal{F}$ when applied to the data set A.

The Permuted Surface Fitting Problem

INSTANCE: A surface approximation method $\mathcal{F}$ and a square nonnegative real matrix A.

QUESTION: find a permutation matrix P such that $S_{\mathcal{F}}(PAP^T)$ has minimum surface area (over all possible permutations P).

8.0.2 Balanced Graph Partitioning Problems

Let γ and λ be real parameters such that $0 < \gamma, \lambda \leq 1$. An undirected graph $G = (V, E)$ is called λ-sparse if $|E(G)| \leq \lambda\binom{|V|}{2}$; G is called γ-dense if $|E(G)| \geq \gamma\binom{|V|}{2}$.

The approach taken in this paper depends on the existence of a predefined partition of the vertex set that satisfies certain desirable conditions. Finding such a partition is in our opinion one of the most fundamental problems in the exploration of massive multi-graphs. More formally, a partition $\Pi(V(G))$ is called (λ, γ)-*good* if $G[\Pi(V)]$ is λ-sparse, G_{V_i} is γ-dense for every subset V_i in the partition and if the ratio of the sizes of any two blocks in the partition is close to 1.

The Parameterized Graph Partition Problem

INSTANCE: A connected graph $G = (V, E)$ and two real parameters γ and λ, such that $0 < \gamma, \lambda \leq 1$.

QUESTION: Find a (λ, γ)-*good* partition of $V(G)$.

Are there I/O efficient approximation algorithms for finding (λ, γ)-good partitions of graphs whose vertex and edge sets do not fit in main memory?.

9 Conclusions

Graph surfaces constitute one step forward in the search for the ultimate solution to the problem of visualizing and computing with external memory multi-digraphs. The *graph surfaces* metaphor allows the integration of visualization and computation on massive data sets. They open the door to the use of matrix theoretical methods for the analysis of very large data collections. As a case in point, it is tantalizing to suggest that the spectra of one level of the hierarchy has a direct relation with the spectra at lower levels. From the computational point of view, a distributed memory implementation of *graph surfaces* is certainly a natural question to ponder.

Acknowledgements: We would like to thank Fan Chung and Sandra Sudarsky for key discussions during the gestation of the main ideas that led eventually to this work.

References

[1] J. Abello. A dynamic multigraph model for massive data sets. Technical report, AT&T Labs Research, Florham Park, NJ, December 1997.

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

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

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

[5] Mark de Berg and Katrin Dobrindt. On levels of detail in terrains. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C26–C27, 1995.

[6] L. De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Comput. Graph. Appl.*, 9(2):67–78, March 1989.

[7] N. Dyn, D. Levin, and S. Rippa. Algorithms for the construction of data dependent triangulations. In J. C. Mason and M. G. Cox, editors, *Algorithms for Approximation II*, pages 185–192. Chapman and Hall, London, 1990.

[8] 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.

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

[10] L. De Floriani, B. Falcidieno, and C. Pienovi. A Delaunay-based method for surface approximation. In *Eurographics '83*, pages 333–350. 1983.

[11] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. volume 13, pages 199–207, August 1979.

[12] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical report, CS Dept., Carnegie Mellon U., Sept. 1995.

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

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