SANDIA REPORT

SAND 2007-6307
Unlimited Release
Printed October 2007

Massive Graph Visualization: LDRD Final Report

Kenneth Moreland and Brian Wylie

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND 2007-6307
Unlimited Release
Printed October 2007

Massive Graph Visualization: LDRD Final Report

Dr. Kenneth D. Moreland, SMTS
Scalable Analytics & Visualization
P.O. Box 5800
Albuquerque, NM 87185-1323

Brian N. Wylie, PMTS
Scalable Analytics & Visualization
P.O. Box 5800
Albuquerque, NM 87185-1323

Abstract

Graphs are a vital way of organizing data with complex correlations. A good visualization of a graph can fundamentally change human understanding of the data. Consequently, there is a rich body of work on graph visualization. Although there are many techniques that are effective on small to medium sized graphs (tens of thousands of nodes), there is a void in the research for visualizing massive graphs containing millions of nodes. Sandia is one of the few entities in the world that has the means and motivation to handle data on such a massive scale. For example, homeland security generates graphs from prolific media sources such as television, telephone, and the Internet. The purpose of this project is to provide the groundwork for visualizing such massive graphs. The research provides for two major feature gaps: a parallel, interactive visualization framework and scalable algorithms to make the framework usable to a practical application. Both the frameworks and algorithms are designed to run on distributed parallel computers, which are already available at Sandia. Some features are integrated into the ThreatView™ application and future work will integrate further parallel algorithms.

Acknowledgment

The Massive Graph Visualization LDRD team would like to acknowledge the significant support provided by Nabeel Rahal, our LDRD program manager. Without Nabeel's fanatical support, our work may never have left the ground. We would like to acknowledge those who provided both direct and indirect technical development: Timothy Shead and Jeffrey Baumes. We also acknowledge Bruce Hendrickson, Jonathan Berry, and Patricia Crossno for their technical guidance.

Contents

List of Figures

Executive Summary

The purpose of the project is to develop techniques that enable understanding of large databases of connected data. Such databases naturally form mathematical structures called graphs. Graphs are a vital way of organizing data with complex correlations. A good visualization of a graph can fundamentally change human understanding of the data. Consequently, there is a rich body of work on graph visualization.

Although there are many techniques that are effective on small to medium sized graphs (tens of thousands of nodes), there is a void in the research for visualizing massive graphs containing millions of nodes. Sandia is one of the few entities in the world that has the means and motivation to handle data on such a massive scale. For example, homeland security generates graphs from prolific media sources such as television, telephone, and the Internet.

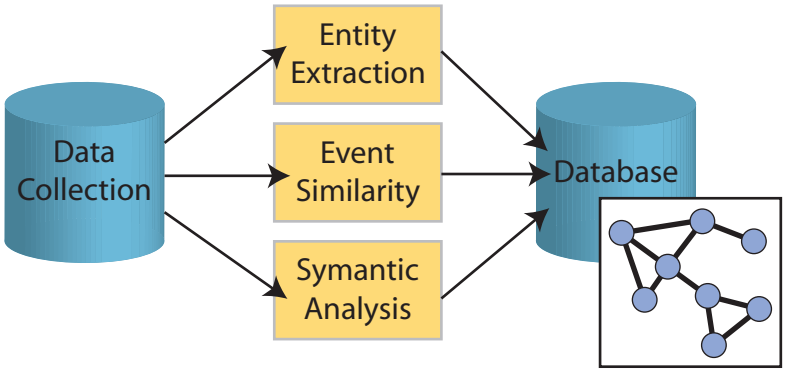


Figure 1. Graph extraction from raw data.

As demonstrated in Figure 1, the graphs with which homeland security must work originate with raw data collected from a variety of sources. Data sources of the modern world are far too prolific to sift through "by hand." Instead, the data is first conditioned by any number of automated processes such as entity extraction, event similarity calculation, and semantic analysis. The result is a set of entities that are stored in a database in such a way that relationships are easily retrieved or extracted. These relations between entities in the database fundamentally form a graph. The large size of the input data collection yields a similarly large graph database; millions of nodes are not uncommon.

The goal of this project is to facilitate the visualization of such massive graphs. We have identified to major fronts of basic research that are required for practical visual

applications: a parallel, interactive visualization framework and scalable algorithms to make the framework usable to a practical application. On both fronts we work with the constraint that the algorithms must perform well on a distributed memory parallel machine because of their proven scalability and accessibility from within Sandia.

Framework

To build a framework on which to visualize massive graphs, we start with an already well developed code base. The field from which we draw from is **scientific visualization**. Scientific visualization concerns the visualization of meshes, simulation results, and real-world sensor readings, all of which have a definition in physical space. As a world leader in supercomputing for physical simulation, Sandia has also fostered excellence in scalable parallel scientific visualization.

The framework that Sandia uses to perform scientific visualization is embedded in **VTK**, the Visualization Toolkit. VTK provides a component architecture for visualization components that also provides for data-centric parallel processing. Built on top of VTK is **ParaView**, which provides both a client/server architecture for parallel interactive visualizations and an end user application for performing parallel visualizations.

Graph visualization falls under the classification of **information visualization**, which deals with abstract data types that have no necessary connection to physical space. ParaView, designed as a scientific visualization tool, is not equipped to handle this type of visualization. However, the distinction between scientific visualization and information visualization is in many ways artificial. Many problems in one domain are mirrored in the other.

To leverage many of the features of VTK, including scalable parallel processing, we started the **Titan** project. The Titan project organizes the work of several funding sources (this LDRD included) to complement VTK with information visualization capabilities.

The role of this project in Titan is the creation of scalable graph components. To accommodate parallel graph structures in a variety of settings, we built data sets that support edge partitioning, as defined by Yoo et al. [15]. These partitioning structures allow you to divide the graph data amongst distributed processes and identify where neighboring vertices are located, the bare minimum for supporting parallel graphs, In addition, edge partitioning also limits the amount of communication needed to get neighborhood information and to trace connections within graphs. This feature can drastically reduce the communication overhead in algorithms.

The ability to store large graphs in parallel is pointless unless one can load in data, so our project addressed this as well. Once data is processed it is most appropriate

to store the entity information in a database. Databases are a very well developed technology that provide for the storage, retrieval and search of large amounts of data. Although an active area of research, databases are currently not designed specifically for graph storage and retrieval. Most databases store tables of relational data (although some object-oriented databases exist), and retrievals from the database are assumed to be tables that are streamed to a single client.

To read in large graphs from databases, we addressed two important issues. First, we read data from the database in a parallel distributed manner. We do this by first instructing the database to save a query in an internal structure, and then we stream partitions of the table in each process. Second, we convert the tables into graphs by identifying vertices and pulling in edges as necessary.

Algorithms

With the framework in place to handle large, parallel graphs, we addressed the basic algorithms necessary for visualization. One critical class of such algorithms are the layout algorithms that impose spatial coordinates on the graph elements such that their placement suggests the relationships of the entities. The most common layout algorithms are **force directed**, which places vertices by minimizing the energy caused by attractive and repulsive forces defined by the graph structure. The time to solve this problem completely grows quadratically with respect to the number of graph nodes. Good approximations with better running times exist [7], but they are difficult to parallelize efficiently and still have a greater than linear running time.

We investigated alternative layout algorithms that have a running time that grows linearly with respect to the number of vertices and edges in the graph and that can easily be run in parallel. The first such algorithm, called simply **fast layout**, is a relatively simple extension to the traditional force directed algorithm. This algorithm follows the same basic flow except that the force field is computed on a finite mesh. This sampled field can be computed in linear time with respect to the vertices and can be combined in parallel with a simple reduce command.

The second layout algorithm we designed is the **G-Space** layout algorithm. This algorithm differs significantly from the traditional force-directed layout algorithm. The G-Space algorithm first picks two vertices in the graph and performs a breadth-first search on each to determine the geodesic distance of each vertex to these two. These two values form coordinates in a 2D geodesic space. The G-Space layout is then based on the plot of the vertices in geodesic space.

Other algorithms deal with rendering and viewing graphs. The problem with viewing large graphs, even those with a good layout, is that the image quickly becomes saturated. It can be the case that there are more elements being rendered than pixels on the screen. To address this problem, we can render the elements with the assumption

that each is smaller than a screen pixel. The fragments then accumulate to show collections of objects. Although done in the past, we found that the technique, although effective for vertices, was not subtable for edges. In response to this, we designed an algorithm that accumulates edges based on their position and orientation to get a better idea of when edges bundle together.

Another way to view a large graph is to use a landscape representative of the density of vertices. We first derive a sampled field based on the density of vertices, and then perform a carpet plot of that field. This process must be very fast, as it is expected to be updated as we interact with (pan and zoom) the graph. We perform this operation using a special fast splatting algorithm in combination with parallel compositing techniques.

Once a landscape view is available, it is important to be able to identify where vertices are. Given little *a-priori* knowledge, the most useful information you can give is to identify major clusters. You can do that by identifying the peaks in the vertex density field and then placing an identifying label of a representative vertex there. Again, the algorithm we created has to work very fast to maintain interactivity.

Chapter 1

Introduction

The focus of this LDRD was to research the foundations for interactively analyzing massive graphs. This problem domain is of particular importance to homeland security where data accumulated from a variety of independent sources can quickly form these enormous graphs. Interactively visualizing these graphs is vital for detecting threats.

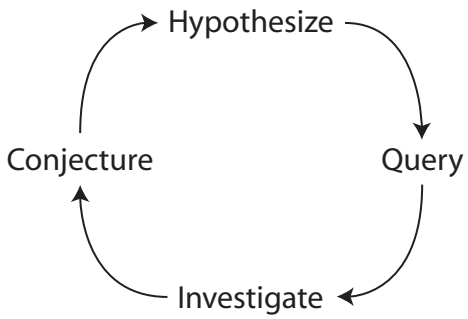


Figure 1.1. Threat Investigation Cycle.

Figure 1.1 shows the cyclical investigation of threats (a simplification of the **analytical reasoning process** [13]). An annalist makes a hypothesis and based on this hypothesis queries the data for relevant information. The annalist then investigates and explores this data. Based on this investigation, conjectures are drawn and the hypothesis is modified in accordance to the new information. The cycle is repeated with more queries to draw more data about the new information.

Clearly this investigative process relies heavily on the ability to not only manage this data but to also sift through it quickly. Several products have been created to this end [4, 9, 14], but to date they all are serial applications. They can handle data of only a limited scale and are not equipped to deal with graphs on the order of millions of vertices. Making the leap to a parallel framework for massive amounts of data requires the foundational research this LDRD project provides.

To build scalable graph visualization applications, we first need a framework on which to build them. This framework must have the ability to read, store, and process graphs. Furthermore, this framework needs to work well on distributed-memory parallel computers, which are the most easily scaled and are already available from within Sandia. The parallel graph framework is discussed in Chapter 2.

A parallel framework is usable only if we have parallel algorithms to run on it. This LDRD project developed some basic parallel graph algorithms that are fundamental to the visualization of these large graphs. The algorithms are discussed in Chapter 3.

Related Projects

This LDRD project is not the only work at Sandia focused on parallel graph algorithms and large graph visualization. There are several other projects in this domain. Yet each project contributes to a different part of the solution, and the collective effort is meant to be collaborative rather than competitive.

Titan

The Titan project is a coordination of several other projects, this LDRD included, to merge scientific visualization and information visualization into the same toolkit: VTK. The purpose of Titan is to leverage the enormous body of work in scientific visualization and jumpstart Sandia's presence in the information visualization domain. VTK is mature open-source tool with a large user base that can contribute back to our information visualization tools.

This LDRD project provides the necessary tools to make Titan scale to large, parallel, distributed-memory computers.

ThreatView™

ThreatView™ is an end user information visualization tool paid for by the PATTON program under the National Ground Intelligence Center (NGIC). ThreatView™ leverages and contributes to Titan for a modular and extensible architecture.

ThreatView™ also leverages the ParaView server libraries for the ability to interactively run parallel algorithms in client-server mode. However, the funding does not include the parallel framework and algorithms also required for parallel processing. Instead, ThreatView™ expects to leverage contributions from other projects (i.e. this LDRD).

Multi-Threaded Graph Library

The Multi-Threaded Graph Library (MTGL) is a compendium of parallel graph algorithms. The algorithm design is focused on a special class of computer architecture called massive multithreading. This architecture has special hardware to enable each processor to efficiently handle many threads running memory-access intensive algorithms. For algorithms that require repetitive memory requests, which many graph algorithms do, this type of architecture can be more effective than much larger distributed memory machines [1].

When massive multithreading architectures are available, MTGL can give a significant performance boost. Thus, MTGL is being integrated into Titan. However, massive multithreading computers are rare and expensive. This LDRD designed algorithms for the more common distributed-memory machines. We worked with the knowledge that the architecture has limitations and that we have to perform with the constraints that we are given.

Parallel Boost Graph Library

The Parallel Boost Graph Library (PBGL), developed at Indiana University, is an extension of the popular Boost Graph Library (BGL) to form parallel processing of graphs, possibly on distributed-memory computers [6].

BGL is integrated into Titan and PBGL will soon follow. However, PBGL, a library strictly concerned with graph algorithms, is missing the framework and many algorithms required for interactive parallel visualization. This LDRD has not duplicated the work of PBGL, but rather fills gaps needed to interactively visualize massive graphs.

VxOrd

VxOrd is a graph layout algorithm that has been developed and tweaked over many years at Sandia. Most recently, the VxOrd algorithm has been used in conjunction with graph coarsening techniques to generate structure from very large real-world graphs.

The technique is an iterative serial process that takes many hours to complete on large graphs. The results can generate stunning visualizations such as those demonstrating the structure of science [2, 3]. However, the resulting visualization is also static. The layout algorithms designed for this LDRD project lie on the opposite end of the spectrum. They are designed to run as fast as possible even if they do not give the most optimal result. Such algorithms are vital for true interactivity. Users can apply progressively slower algorithms to further refine the layout as necessary.

Chapter 2

Parallel Graph Visualization Framework

As part of Titan, this project leverages VTK for its parallel graph visualization framework. The VTK framework, which has supported parallel scientific visualization for years, is most efficient when running in **data parallelism** mode. In this mode, the data is partitioned and divided amongst all of the processes.

To support data parallelism, our framework must include two things. We need data structures that can hold a distributed graph and provide mechanisms for retrieving connectivity information between processes. We also need the ability to load data into the framework. This means reading the data from files or querying from databases. To be scalable, the data input cannot bottleneck on any one process trying to read the data.

Data Structures

The Titan data structure for holding graphs uses an **adjacency list** for its representation. In this representation, edges are defined with a variable-sized array on each vertex that points to the vertices to which it connects. For graphs with average vertex degree much lower than the total number of vertices in the graph (which is by far the common case), this is a compact representation.

We used the work of Yoo et al. [15] to guide our implementation of parallel graphs. Yoo defines two ways to partition a graph. The first way, **vertex partitioning** or 1D partitioning, is a common but naïve partitioning. The basic idea is to divide the vertices of the graph amongst processes and then assign edges based on the process holding the originating vertex. This partitioning is convenient in that the outgoing connectivity for each vertex is held locally. However, the collection of vertices in each process can, and most often does, point to vertices in all other processes, thereby requiring much all-to-all communication for anything but the most trivial algorithms.

The second partitioning mechanism Yoo describes, **edge partitioning** or 2D partitioning, divides edges instead of vertices. Edge partitioning uses not a linear list of p

process but rather a 2D array of $r \cdot c = p$ processes. (The processes do not have to be in a physical array. It is simply an indexing scheme using pairs to identify processes.)

$$\left[\begin{array}{c|c|c|c} A^{(1)}_{1,1} & A^{(1)}_{1,2} & \cdots & A^{(1)}_{1,c} \\ \hline A^{(1)}_{2,1} & A^{(1)}_{2,2} & \cdots & A^{(1)}_{2,c} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A^{(1)}_{r,1} & A^{(1)}_{r,2} & \cdots & A^{(1)}_{r,c} \\ \hline & & \vdots & \\ & & \vdots & \\ & & \vdots & \\ \hline A^{(c)}_{1,1} & A^{(c)}_{1,2} & \cdots & A^{(c)}_{1,c} \\ \hline A^{(c)}_{2,1} & A^{(c)}_{2,2} & \cdots & A^{(c)}_{2,c} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A^{(c)}_{r,1} & A^{(c)}_{r,2} & \cdots & A^{(c)}_{r,c} \end{array}\right]$$

Figure 2.1. Edge partitioning layout from Yoo et al. [15].
The notation $A^{(*)}_{i,j}$ denotes a block owned by process (i, j).

We partition the edges by first looking at the complete $|V| \times |V|$ adjacency matrix. (Again, we do not have to actually store the edges in an adjacency matrix; this is just a conceptual model for dividing edges.) The columns of this array are grouped into c partitions and the rows are grouped into $r \cdot c$ partitions. These partitions are then assigned to processes as demonstrated in Figure 2.1. The result is a **partial adjacency matrix** stored on each process.

The advantage of edge partition over vertex partitioning is that any edge in the graph can be followed by moving along the row or column in the adjacency table. By the way adjacency table is divided amongst processes, the number of processes assigned to a row or column of the table is r or c, respectively. Assuming r and c are relatively equal, a process can follow all of the local edges by communicating with roughly a square root of the total processes.

Reading and Querying

We currently are only able to read in parallel graphs from a database. The reasons for this are two-fold. First, we currently have no file formats that can hold massive graphs and still be read from efficiently in parallel. Second, there is little reason to

read in a large graph without querying capability. From our use cases and personal experience, graph analysis is an iterative process of searching and studying data as described in Chapter 1. The ability to query the data is paramount to this process. Thus, it would never make much sense to load data from a flat file. Instead, an introductory step would be to load everything into a database for efficient access.

We assume that whatever database we are accessing implements the **Structured Query Language** (SQL) or something equivalent. This is a reasonable assumption as SQL is an industry adopted interface that is implemented all current relational databases. Other types of databases, for example object oriented databases, might not implement SQL per se, but will definitely have to implement the equivalent functionality to hope for adoption.

In this document we describe reading from relational databases (i.e. from tables). Relational databases are currently the most mature databases, and this is the implementation we have so far. However, we are also considering, but have not implemented, reading from object oriented databases.

A precondition for reading a graph from a database is to have ready a vertex table and an edge table. This does not preclude building graphs from database queries; it simply means the queries must first be directed to this pair of tables. If necessary, the vertex table can be derived from the edge table by performing a query on the latter that lists all unique vertex entries.

The parallel graph reader first performs a row count on the vertex table. This provides us with the size of the full adjacency table for the graph (which we are not actually building but simply considering the conceptual model). Knowing the dimensions of the adjacency table, we can decide on the partition of the vertices and edges (described in the previous section on data structures). Each process then reads in the vertex data relevant to its local partition using a query on the vertex table with limits.

The information returned from the vertex table query includes the identifiers for the vertices that correspond to the entries in the edge table. We use this information to retrieve the appropriate edges from the edge table. To do this, each process builds rather elaborate queries on the edge table using values clauses that specify the vertices that should be pulled from the edge table. The returned table of edges provides the necessary information for building the partial adjacency lists required for the parallel graph.

Chapter 3

Massive Graph Visualization Algorithms

There is much research, both inside and outside of Sandia, dedicated to the processing of large graphs. In this LDRD we focused specifically on algorithms that provide mechanisms for visualizing large graphs.

Layout

Most graphs have no relationship to physical space. We may think of them as balls connected by sticks floating in space, but we cannot represent them this way until we have chosen a **layout**, the physical location of the graph elements.

Technically, any graph layout is a valid one, but we consider some layouts better than others. The criteria for a good layout can vary based on the problem domain, but the most universal criterion is that, on average, vertices that are connected by an edge are closer than vertices that are not connected. Based on this criterion, we can develop general purpose layout algorithms that perform well on a wide variety of graphs.

Fast Layout

Perhaps the oldest and most widely used class of layout algorithms are **force-directed layout**. In this method of layout, forces are applied to vertices so that they are pushed in the direction of its neighbors.

Figure 3.1 shows the typical forces applied to a graph. Vertices that are connected by edges have attractive or spring forces that increase with the distance between the vertices. Unconnected vertices repel each other, and the repulsion becomes stronger as the vertices become closer. The repulsive force is sometimes applied to all vertex pairs for consistency and to help prevent co-located vertices.

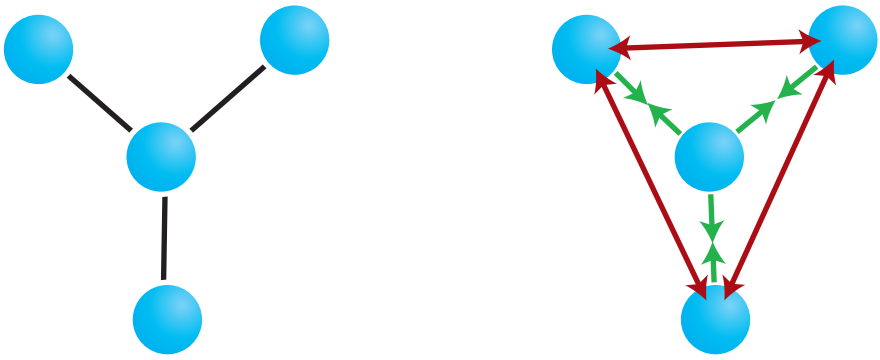


Figure 3.1. A simple graph and the forces applied to it whose minimization yields an appropriate layout.

Splatting Vector Fields

The major problem with force-directed layout is that, typically, every vertex pair forms a force on each vertex. Thus, completely resolving the accumulative forces on all vertices requires considering all vertex pairs. The number of vertex pairs grows quadratically with the number of vertices. A parallel algorithm also has to contend with the communication overhead of dealing with all vertex pairs. A good implementation will prune the overhead by ignoring insignificant forces such as repulsive forces of vertices far away [7]. However, such optimizations can quickly complicate the algorithm and are difficult to parallelize.

The **fast layout** algorithm starts with the observation that if all vertices repel each other (and connected vertices have an overriding attractive force), then we can model the repulsive forces as a vector field that all the vertices share. Rather than compute the repulsive forces individually for each vertex, the fast layout algorithm samples the vector field and computes it everywhere.

To compute the vector field, we employ a technique called **splatting**. To splat the vector field, we first create an image of a single vertex's impact on the vector field. This image is called a **kernel**. The kernel for the repulsive forces from a vertex point away from the vertex with decreasing magnitude, as demonstrated in Figure 3.2.

The vector field can then be determined by taking each vertex, finding the area of the field the vertex is located in, and "splatting" the kernel at that spot by adding the kernel to that region of the vector field, as shown in Figure 3.3. The vector field is formed when all of the splats accumulate.

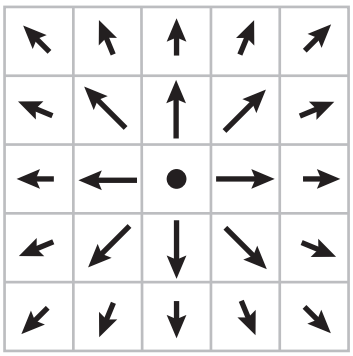


Figure 3.2. Repulsive force kernel.

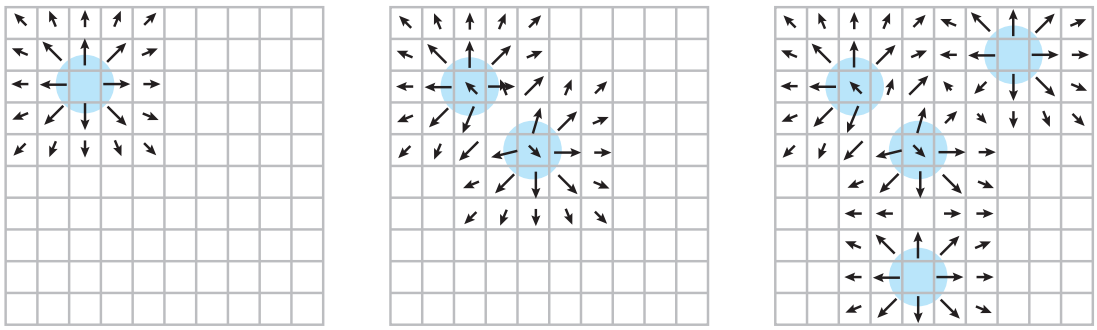


Figure 3.3. Splatting kernels to get a vector field.

Fast Splatting

A big overhead for the splatting technique is applying the kernel for each vertex. The larger the kernel, the larger the overhead. We can significantly reduce this overhead by using what we call the **fast splatting** technique. In this algorithm we perform the splatting in two phases. In the first phase we perform a splat like before, except that we use a simple impulse kernel. The kernel is a single pixel with a value of 1 in it. So the end result is simply a count of the number of vertices that lie within a single sample of the grid.

The next step is to convolve the impulses with the kernel we really want to splat. Because the convolution of a kernel with an impulse is the kernel centered at the impulse and because you can factor convolution, the end result is the same as if we had splatted the kernel directly.

For splatting large numbers of vertices, the fast splatting is significantly faster than

the original algorithm. Each vertex adds only one addition to the algorithm. The convolution of vector field and kernel can be time consuming, but it is a fixed overhead; it does not grow with the number of vertices that you are splatting. Unless you are splatting very few vertices, the time to convolve is smaller than the time to splat the kernel independently for each vertex.

Using a splatting technique, the vector field for the repulsive forces can be determined in time linear with respect to the number of vertices. The attractive forces are determined by following edges. Thus the overall running time of the fast layout algorithm can be characterized by $O(|V| + |E|)$ where $|V|$ and $|E|$ are the number of vertices and edges in the graph, respectively.

Parallelizing Fast Layout

Although we have not yet had a chance to implement it, making the fast layout algorithm parallel is straightforward. The repulsive vector field can be determined by first independently computing the field on each process with its local partition of the graph. The partial vector fields can then be combined by adding them together. This can be done with a simple reduce operation available with MPI.

Once the vector field is in place, the algorithm needs only to follow edges and find adjacent vertices for each vertex. We can do this with limited communication using the edge partitioning described in Chapter 2.

G-Space

Unlike the fast layout algorithm, the G-Space algorithm significantly diverges from the force-directed layout approach. However, the G-Space graph layout approach appears to be suitable for analyzing graph structures on multiple scales under various use cases; it can be used to visualize extremely large graphs representing a database of relationships in its entirety, or it can be used to show the results of a targeted point-to-point query (e.g. Kevin Bacon query). The first phase of our approach was originally inspired by the point-to-point case so well begin our explanation there.

Low Dimensional Embedding (LDE)

The G-Space starts with a technique we call Low Dimensional Embedding (LDE). Clearly, LDE is a play on the term HDE (High Dimensional Embedding) and we adopted it in homage of the work done by Harel and Koren [8]. When we started our work, we were not yet familiar with HDE and were only interested in two dimensional embeddings. Our work was motivated by the following use case: given a database

containing a large social network, a user wishes to see the **meta-relationship** (shortest path, types of relationships along the shortest path, identified critical links, etc.) between two individuals. Thus the LDE process is essentially HDE with two pivot points.

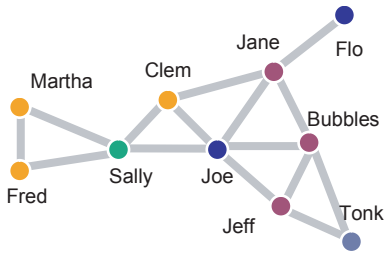


Figure 3.4. Example email network.

To demonstrate the LDE, we will use the simple graph shown in Figure 3.4. Assume that the user wishes to display the meta-relationship between vertices "Martha" and "Tonk."

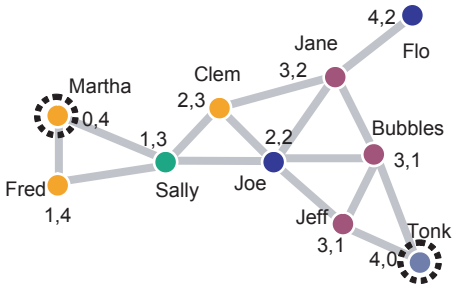


Figure 3.5. Giving coordinates to the graph vertices based on their shortest path distance from pivot points.

The LDE process selects those two vertices as pivot points and conducts a breadth-first search (BFS) from each. Each BFS tags the vertices it visits with the length of the shortest path the the start vertex. Every vertex now has an associated two-tuple containing its shortest-path distance from each of the two pivot points as shown in Figure 3.5.

Using these tuples we simply map each vertex into two dimensional geometric space as shown in Figure 3.6. You can see from Figures 3.5 and 3.6 that the "Jeff" and "Bubbles" vertices both have distance coordinates of 3,1 from the pivot points (a many-to-one mapping in the embedding process). Where HDE resolves these many-to-one mappings using higher dimensions and principle components analysis (PCA), LDE accepts the many-to-one mappings and resolves them using a different technique

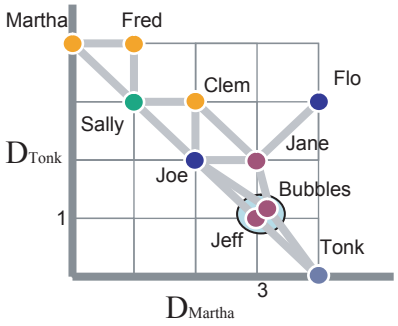


Figure 3.6. Direct mapping of graph distance (length of the unweighted shortest path) as coordinates into two dimensional space.

(which we address later). The advantage to this approach is that there is no need to run 50 or 100 breadth-first searches (BFS), embed the graph into a 100 dimensional space, do a PCA, compute projections with maximal variance and then project down to two dimensions.

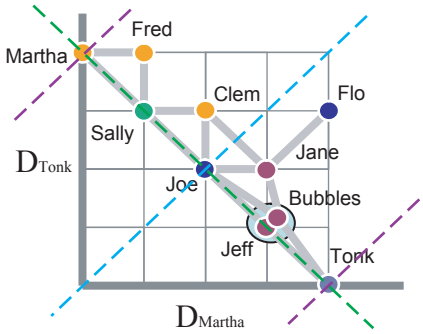


Figure 3.7. Low dimensional embedding feature lines.

The simplicity of the low dimensional embedding exposes many interesting geometric properties within the layout as shown in Figure 3.7. The embedding based on graph-theoretical distance ensures that the shortest path between the pivot points is guaranteed to be along the dashed green line. Longer paths form "arcs" into the positive quadrant. The light blue line represents shortest path equidistance between the pivot points. "Flo" has a shorter path to "Tonk" than to "Martha" simply based on this geometric property. The distance from a vertex to a pivot vertex can never be greater than its distance to the other pivot vertex plus the shortest path between the pivots. Thus, the two darker purple lines mark the "boundaries" of the layout.

Rotating the graph 135 degrees, as demonstrated in Figure 3.8, provides a more

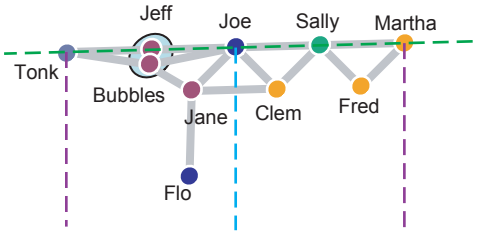


Figure 3.8. LDE plot rotated 135 degrees.

natural orientation. The shortest path will be along the top and equidistance is the vertical line in the center.

Generalizing the LDE Process

The previous example uses a point-to-point query where the two pivot points for the LDE were specified by the user. As mentioned in the HDE work, pivot points can be automatically computed. In our case, we run a total of three BFS searches. The first BFS begins at a random vertex within the graph, finds a **pseudo-peripheral vertex**, a vertex that has a long shortest path but is not guaranteed to be the longest. The pseudo-peripheral vertex is chosen simply as one of the vertices furthest from the initial random pick. The pseudo-peripheral vertex becomes one of the pivot points and is passed as the starting-point for the second BFS. The second BFS provides vertices with the first component of their graph distance tuple, and identifies a second pseudo-peripheral vertex to use as the second pivot point.. This vertex becomes the starting-point for the third BFS, which provides each vertex with the second component of the distance tuple. The running time of this phase is $O(|V| + |E|)$ where $|V|$ and $|E|$ are the number of vertices and edges in the graph, respectively.

In our testing the algorithm appeared fairly insensitive to the particular choice of peripheral vertices; initially we had been more formal about the pivot choices. In fact, the normal procedure for finding pseudo-peripheral vertices is to conduct a series of BFS passes until the passes give convergence on the two pivot points. Figure 3.9 demonstrates the layout resulting from the generalized LDE process with automatic pivot calculation.

Resolving the Many-to-One Mappings

Inspired by the terrific edge bundling technique of Holten [10], we adopted the term **vertex bundling** to describe how vertices can be packed together based on their

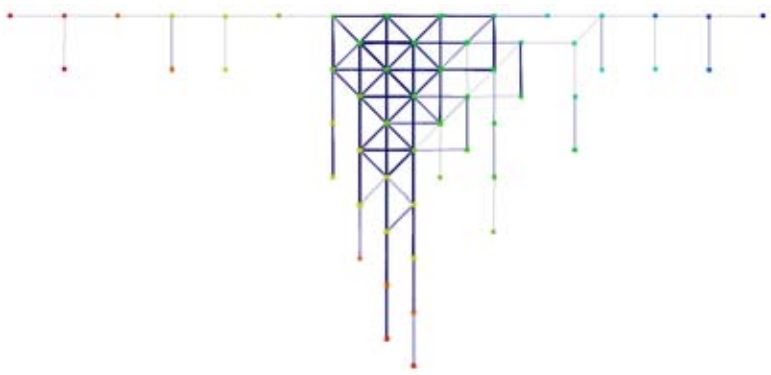
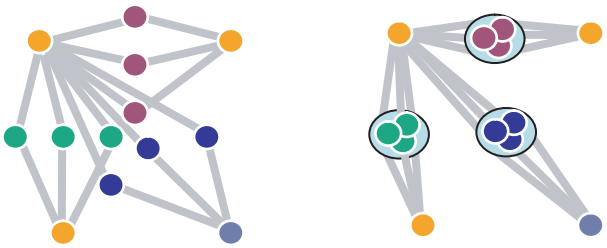


Figure 3.9. Automatic LDE layout of the Enron email database (322k edges, 75k vertices) [11].

connectivity attributes. Vertices have edge obligations to other vertices; if you have the same obligations as another set of vertices, bundle yourself together and *minimize* the space you take up.



(a) Typical Force Directed Space Filling

(b) Space Minimizing Vertex Bundles

Figure 3.10. Simplifying graph layout, and clarifying topological relationships with the use of vertex bundles.

If you take the small graph in Figure 3.10, you see that traditionally the vertices are pushed apart to fill the available space. We suggest the opposite approach, grouping the vertices together into "semantic" bundles which, like the edge bundling technique, simplify the layout and bring clarity to the topological relationships within the graph.

On the large scale, vertex bundling can be used to resolve a significant portion of our many-to-one mapping problem. We now use the problematic distance bins as an ally to create a "scaffolding" of control points. Each vertex has an edge to one of more other vertices, at this point all vertices lie within a bin (control point), so we simply traverse the vertex list, determining which vertices have edges to which control points and bundle all vertices with edges to the same control points.

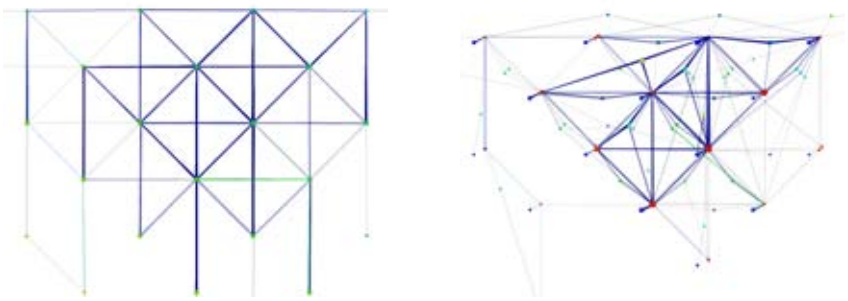


Figure 3.11. On the left an LDE layout of a subset of the Enron database (1374 Vertices, 2241 Edges). On the right the same layout after the vertex bundling pass. Vertex bundling significantly mitigates the many-to-one mappings and helps convey topological information.

After the graph has been through the LDE phase and looks like the layout shown on the left side of Figure 3.11, we now pass the graph through the vertex bundling process and in a manner similar to marching cubes, each vertex has its edges tested against a case table of control points to see which vertex bundle it will be a member of. The entire graph is processed from first vertex to last, and depending on which case is matched the vertex is offset some relative amount from its current position. The running time of this phase is $O(|V| + |E|)$.

Currently we call out 14 different cases (and 11 additional sub-cases) that are split out from the LDE bins as shown in Figure 3.12. The 25 defined cases are as follows. For each vertex the following are determined:

1. Edges only go to vertices in 1 bin: case 0.
2. Edges go to vertices of 2 bins: cases 1 - 11.
3. If you are case 1-11 and also have edges back to the main bin then you are placed very close to your bundle but biased toward the main bin (see inset of Figure 3.13): cases 12 - 22.
4. If you do not meet any of these 23 cases, but you have one or more edge connections within the same bin you stay in the bin: case 23.
5. If none of these apply then you are marked as an "unresolved vertex" and placed in the main bin.

The diagram in Figure 3.12, at first looks oddly biased towards the bottom left corner. When splitting out the vertex bundles an even distribution seems more effective, until

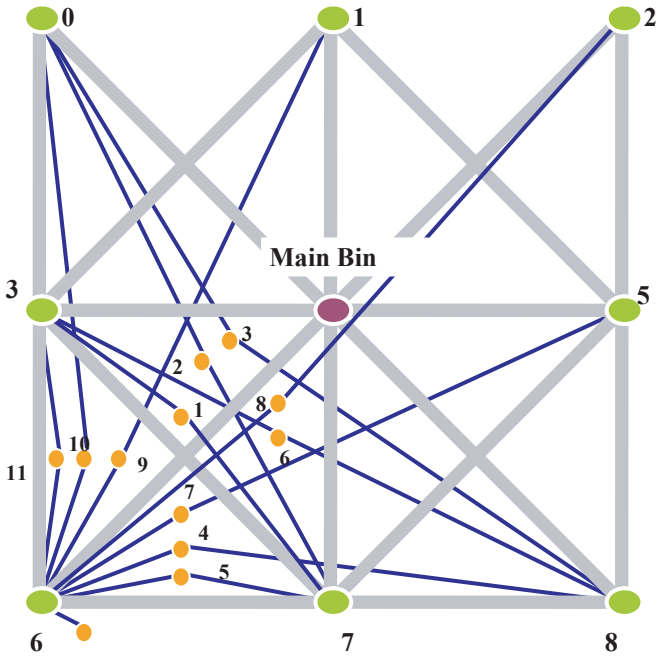


Figure 3.12. Vertex bundle Map. For each distance bin created by the LDE, the following vertex bundle map is applied.

the realization that these bundle maps are applied at every bin within the LDE layout. So the diagram will need to mesh well when placed next to, above, and below your neighbors who are also applying the diagram to their vertices.

Parallelizing the G-Space Layout

We have not yet had a chance to implement it, but performing a G-Space layout in parallel is straightforward and efficient. All the major pieces for doing this are given by Yoo et al. [15] and are already implemented in Titan.

The majority of the layout algorithm involves performing breadth-first searches. Yoo's algorithm is proven to be scalable and efficient on distributed memory machines. The vertex bundling and breakout requires only knowing the bin of the nearest neighbors. This information can be achieved with a limited amount of communication with the edge partitioning described in Chapter 2.

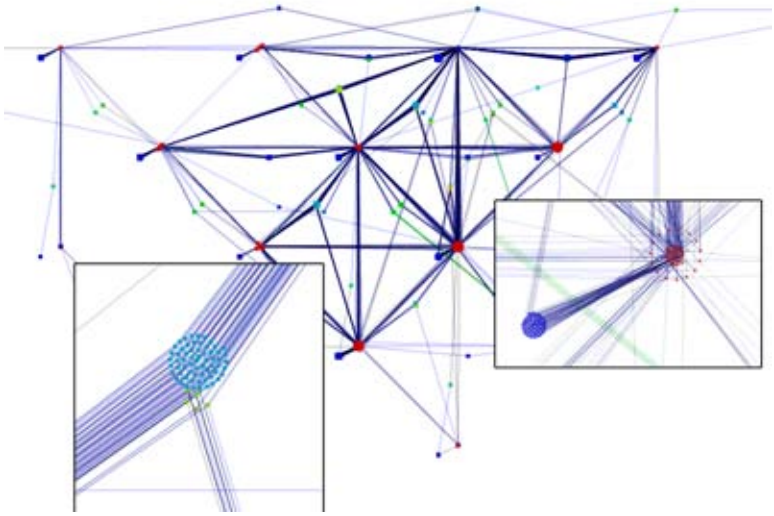


Figure 3.13. Enlarged image of vertex bundling breakout (left side of Figure 3.11 with insets showing layout detail.

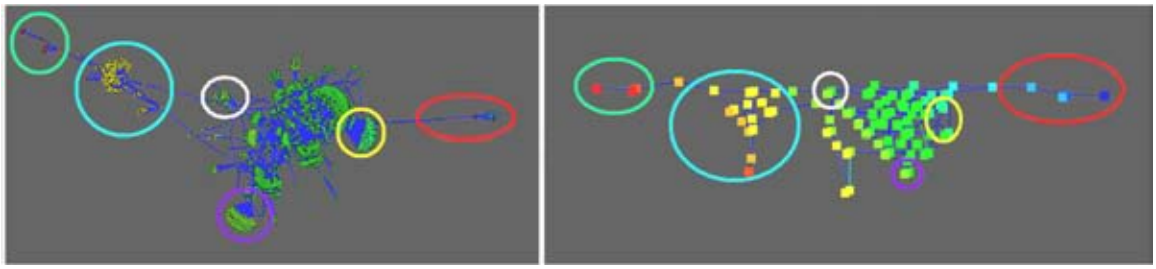


Figure 3.14. Correlation of the G-Space layout to a traditional force directed layout. Color matched circles represent similar sets of vertices between the two layouts. Data: subset of Enron database (1374 Vertices, 2241 Edges).

Results

The quality of the visual layout of G-Space is hard to quantify when comparing it to other layout techniques. The angular lines and extremely tight clusters give the appearance that the layout is showing a small graph, and so we wanted to give some visual registration with Figure 3.14. The color matched circles enclose the same vertex sets in both layouts. The dataset shown is a subset of the Enron email database (1374 Vertices, 2241 Edges). The remarkable correlation between the visual appearances of the layouts is partially because they were rotated and scaled to better show cluster similarity.

On a qualitative level, when exploring the two layouts side by side, with linked selection, the G-Space layout conveys better global graph structure and allows more detailed inspection of topological relationship.

Edge Accumulation

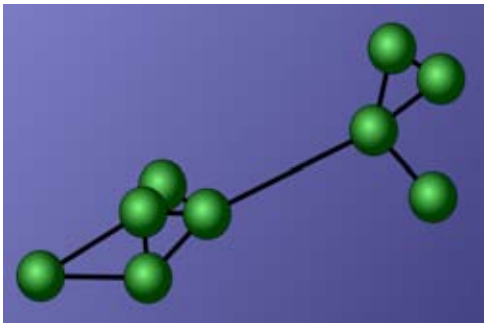


Figure 3.15. Ball and stick drawing of a simple graph.

An obvious but effective way to represent graphs in a visualization is to use a **ball and stick** drawing. This simply means that vertices are represented as balls and edges are represented as sticks connecting them as shown in Figure 3.15. We have also seen examples of this type of drawing elsewhere in this document.

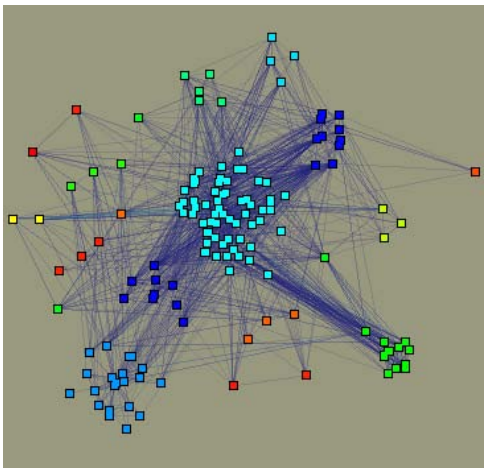


Figure 3.16. Ball and stick drawing of a larger graph.

As the graph drawn becomes bigger, the size of the balls and sticks must become smaller to make them distinct. This is typical of any "zoom out" operation. With

even a moderate sized graph, the balls and sticks become the width of a pixel as shown in Figure 3.16.

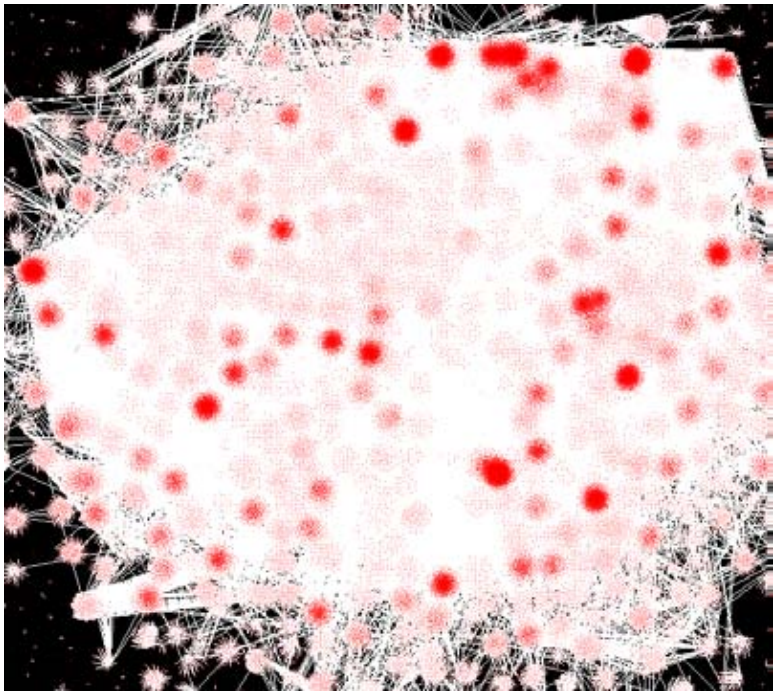


Figure 3.17. A completely saturated graph drawing.

As the graph we draw becomes even larger, the display quickly becomes saturated. In Figure 3.17 we see a drawing of a graph built from the Enron email database [11] containing 75,539 vertices and 322,638 edges. This images is completely saturated with edge data. We have reached the limit of our display and the amount of information we get from edges is reduced to zero.

A simple way to get around this problem is to throw away the edges from the drawing and draw only the vertices. This is often called a **galaxy** drawing so named because the plot of the vertices resembles the plotting of stars in a map of a galaxy. In this type of drawing we have to glean edge information from the positions of the vertices, which, if the layout is done properly, will show high areas of connectivity.

Pixel Blending

All is not lost. There exist **blending** techniques is computer graphics that make it possible to represent effects from features that are smaller than a pixel. The basic idea for the technique is to add an **alpha** parameter to the color that defines the fraction of the pixel covered. Elements smaller than a pixel will have an alpha parameter less

than one. Porter and Duff [12] provide an algebra for combining colors with the alpha parameter.

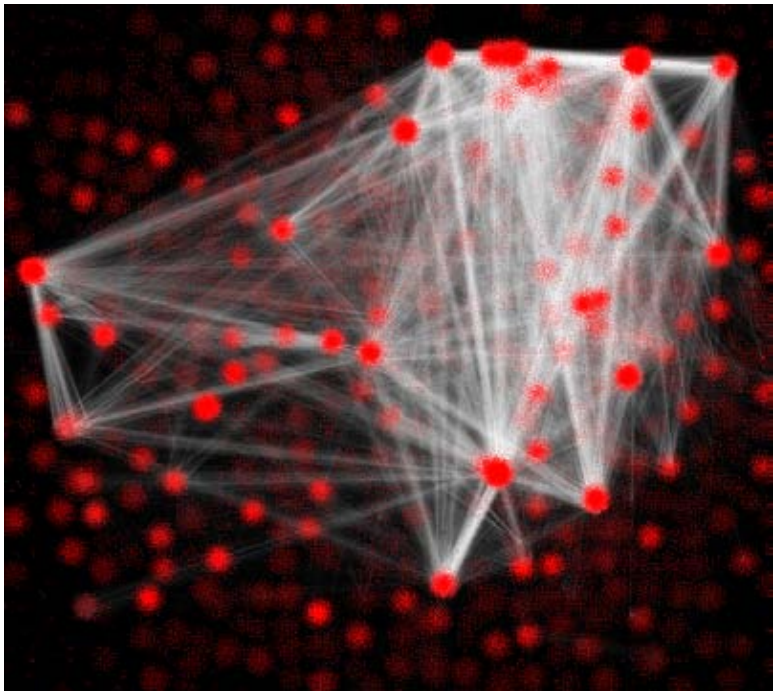


Figure 3.18. The same graph as Figure 3.17 drawn with sub-pixel elements.

So we can effectively lower the width of the elements below the width of the image pixels by reducing the alpha parameter. Massive graphs that require very small alpha values can cause precision problems when accumulating the colors, but circumnavigating this problem results in an image like the one in Figure 3.18.

Employing the blending technique clearly makes the edge information more visible. We can make out areas of high versus low density of edges, and if we look very carefully we can make out groups of edges that connect clusters far away. The image is not perfect, however. There is still quite a bit of clutter in the graph. Lots of dissimilar edges are accumulating together to form unimportant clumps in the graph. Furthermore, bundles of edges are difficult to spot and trace.

Edge Similarity

The problem with the blending operation is that the per-pixel computation is not indicative of the similarity of nearby edges. As is demonstrated in Figure 3.19, a set of dissimilar edges can have as much or more contribution as a pair of edges that are location-wise nearly identical.

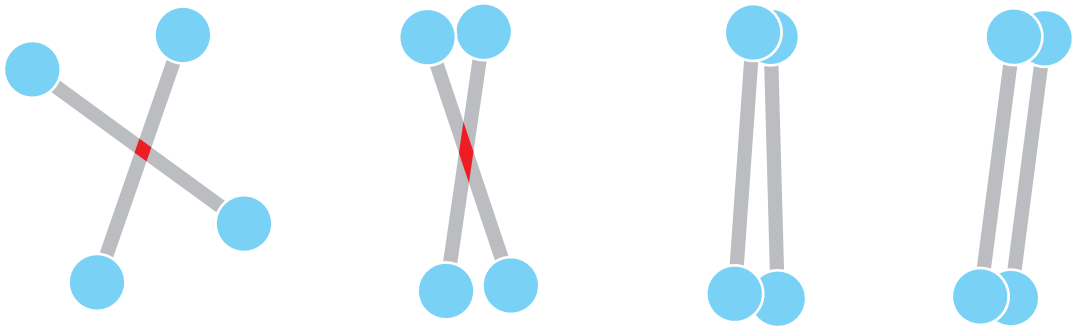


Figure 3.19. Blending contribution from edge pairs.

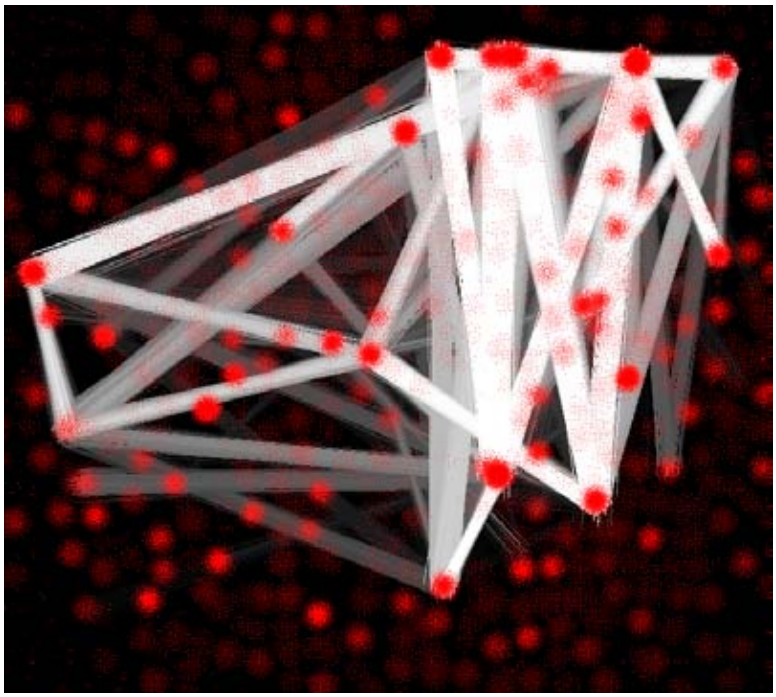


Figure 3.20. The same graph as Figures 3.17 and 3.18 drawn with edges weighted based on how similar they are to other edges.

We really want the contribution of the pair of edges to be based on the similarities of the entire line segment including location, length, and angle. When we do that bundles of edges stand out much more whereas edges with random orientations are completely hidden regardless of overlap as shown in Figure 3.20. The remainder of this section will describe the algorithm for determining all of the **edge frequencies**,

a weighted count for each edge of how many other edges have similar placement.

There are several ways to parameterize a line segment. We use the segment's centroid, length, and orientation angle as the parameters. The main reason for this parameterization is that the centroid gives an easy way to compare the spatial distance of two segments. In particular, for two edges to be considered "similar," their centroids must be nearby. Thus, when we have a segment with which we are looking for similar segments, we need only look in a small radius around the centroid. This provides a very effective and important pruning of the number of comparisons we need to perform.

Sweep Algorithm

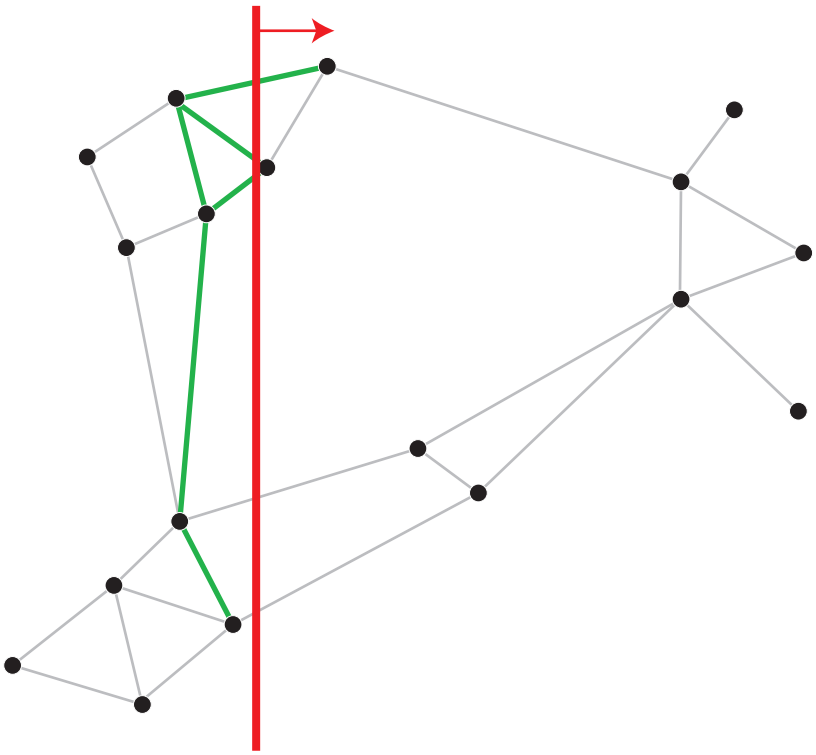


Figure 3.21. A sweep line on a graph. As the sweep line (red) passes over the graph, it maintains a list of "active" edges (green) that it uses to look for similar edges.

The fastest way we have found to determine the similarities of all of the edges (or rather, the line segments representing the edges) of a graph, is to use a **sweep line**. As the name implies, the algorithm will sweep a line across the graph as demonstrated in Figure 3.21. As the line sweeps, it will maintain an **active line segments list**

that contains candidates of segments similar to those near the sweep line.

As the sweep line encounters a new segment centroid, it first checks to see what other segments in the active list are similar to it. The "frequency" parameter of all these similar segments is updated. The new segment is then added to the active lines list. When the sweep line passes past the point where a segment in the active list may be considered similar to one centered on the sweep line, it is removed from the active lines list.

```
LineFrequency2D(Lines)
 1  Initialize Frequencies (either to the weights if using them or 1 if not)
 2  Initialize EventQueue to empty
 3  Initialize ActiveLines to empty
 4  for each line in Lines
 5        do Insert ADD event for line in EventQueue at line's centroid
 6  while EventQueue has events
          do
 7           Pop Event off of EventQueue
 8           if Event is ADD
                then
 9                    line1 ← line associated with Event
10                    for each line2 in ActiveLines that is "similar" to line1
11                          do Update Frequencies for line1 and line2
12                    Insert line1 in ActiveLines
13                    Insert REMOVE event for line1 in EventQueue
                        at distance where line1 cannot be similar.
                else ▷ Event is REMOVE
14                    Remove line associated with Event from ActiveLines
15  return Frequencies
```

Figure 3.22. A sweep algorithm for computing the line frequency in the plane.

The sweep algorithm is shown in Figure 3.22. Although conceptually the sweep line moves continuously over the graph, we are really only concerned when the line hits "events" at certain discrete positions. Thus, the algorithm controls the sweep plane with an event queue.

Although not demonstrated in the pseudocode of Figure 3.22, it is important that the *ActiveLines* list be implemented as a search tree or some other search structure that can be iterated over. This is important so that when finding all of the segments similar to the new one (line 10), we can iterate over only those segments with a

centroid near the new segment's centroid.

Another speed benefit can be obtained by implementing *EventQueue* as two separate search structures: one for the ADD events and one for the REMOVE events. The ADD event queue is filled at the beginning of the algorithm. It can be most efficiently implemented as a simple list of segments, sorted by the location of the centroid.

The REMOVE event queue will have events added and removed throughout the algorithm. This queue must be implemented as an ordered search structure. Inserting and removing events will be much faster when there is less items in the queue. Thus, not having the ADD events, for which there will be more of throughout the algorithm than the REMOVE events, will significantly save on the running time.

Drawing the Frequent Edges

Once the edge frequency is determined drawing an image like in Figure 3.20 is straightforward. When drawing the edges, it is easy to shade the edges based on their frequencies. Edges with high frequencies should stand out whereas edges with low frequencies should blend into the background. The range of colors and the function of mapping frequency to highlighting (linear vs. logarithmic) depends on the graph. A proper highlighting is up to user preference and experimentation.

When drawing the edges, you also want the most frequent edges "on top." One way to arrange this is to order the edges from lowest to highest and use the **painter's algorithm** to draw the lowest frequency edges first and cover them up with the higher frequency edges. When using graphics hardware, an easier (and usually quicker) way to place the most frequent edges on top is to use a 3D orthographic projection, assign a depth value to the edges based on the frequency, and then let the graphs hardware's internal **z-buffer** hidden surface removal place the most frequent edges on top.

Parallel Landscape View

A **landscape view**, demonstrated in Figure 3.23, is an alternative method for drawing graphs. In this method of graph visualization, a terrain is used as a metaphor for the density of the graph with peaks corresponding to dense vertex clusters and valleys corresponding to empty portions of the graph. As the user zooms in and out, the landscape changes to show more or less detail. Landscape view is currently used in several serial visualization tools [4, 14].

To make the landscape, we first compute a field based on the density of the vertices. To make this density field continuous, we allow the vertices to effect the density in some finite space around it that is determined by the current zoom level. We compute

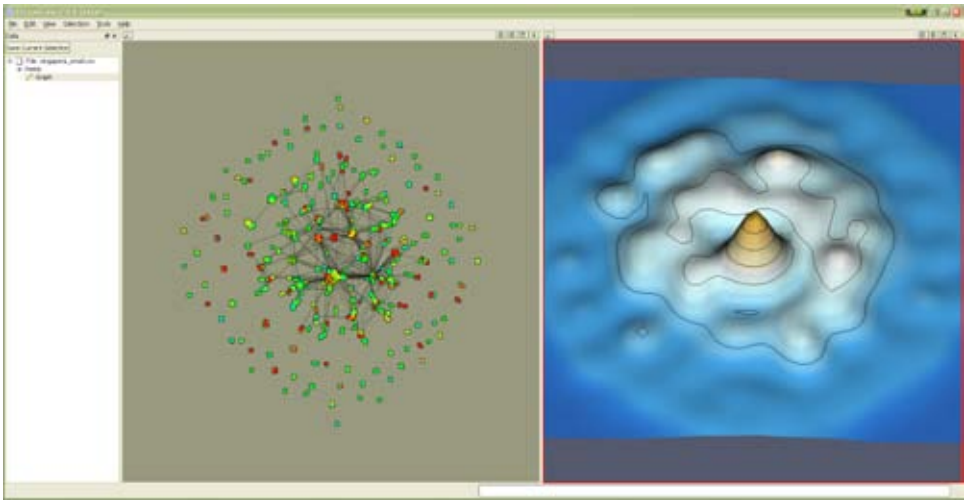


Figure 3.23. An example of a landscape graph view (right) with the corresponding ball and stick view on the left.

this density field on a sampled 2D grid. We use the same fast splatting method that is part of the fast layout algorithm (see the corresponding section starting on page 21).

With the 2D mesh containing the density field computed, we perturb the mesh in the Z direction proportional to the density value there. We also color each point based on the density value (and, equivocally, the height) for further cues on the density. This geometry is then sent to graphics hardware for the appropriate filling and shading. In the example in Figure 3.23, we have also added contour lines. Note that this all must happen in a fraction of a second as the landscape needs to be recalculated as the user changes the viewpoint.

Implementing landscape generation in parallel is straightforward. We assume that each process has some partition of the graph (which is the point). Each process independently computes the density field of its local partition of vertices (using the fast splatting method). As the density fields are computed, we need to make sure that any **ghost vertices**, vertices that are replicated on other processes, are removed. We do not want any vertices counted multiple times.

The partial density fields can be combined by simply adding corresponding mesh values together. This can be done with a simple reduce operation available with MPI. Once the density fields are combined, the resulting vector field, which is independent of problem size, can be further transformed in the serial manner described previously. The resulting geometry can also be sent from parallel server to client for local rendering and display.

Parallel Peak Identification and Labeling

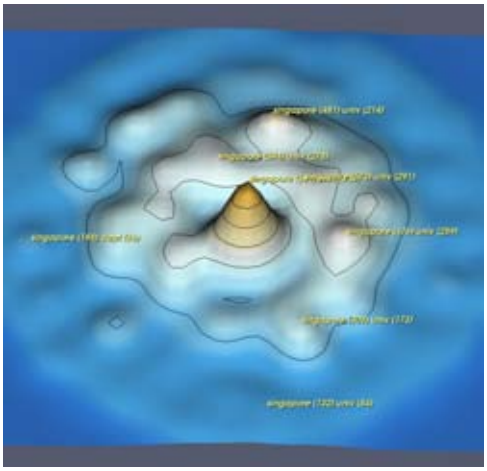


Figure 3.24. An example of peak labels.

The big advantage of the landscape view is that it provides the location and relative size of vertex clusters within a graph that automatically change scope as the user zooms in and out. This information is meaningless, however, without knowing what each peak represents. We can provide a lot of information about each peaks contents with **peak labeling**, where we place an identifying string at each peak of the landscape as shown in Figure 3.24.

Requiring the user to label the peaks is problematic as it is a time consuming task that would have to be repeated when the view changes. Instead, we would like the visualization to automatically identify the peaks in the landscape and provide a label based on the data of the vertices contributing to the peak.

Figure 3.25 shows the flow for the basic peak identification and labeling algorithm. There are three major functional units. **Peak identification** finds the location of all the peaks in the landscape. **Contributor identification** finds the vertices that most contribute to the peak. **Label extraction** uses the contributing vertices to determine a descriptive string. Like the landscape generation, these tasks must finish quickly as they happen every time the user changes the view.

Peak Identification

The peak identification is the simplest portion of the algorithm. A peak is simply a local maximum in the height field. To find the peaks, we simply need to find the landscape points with the highest value in a specified radius. We make the radius

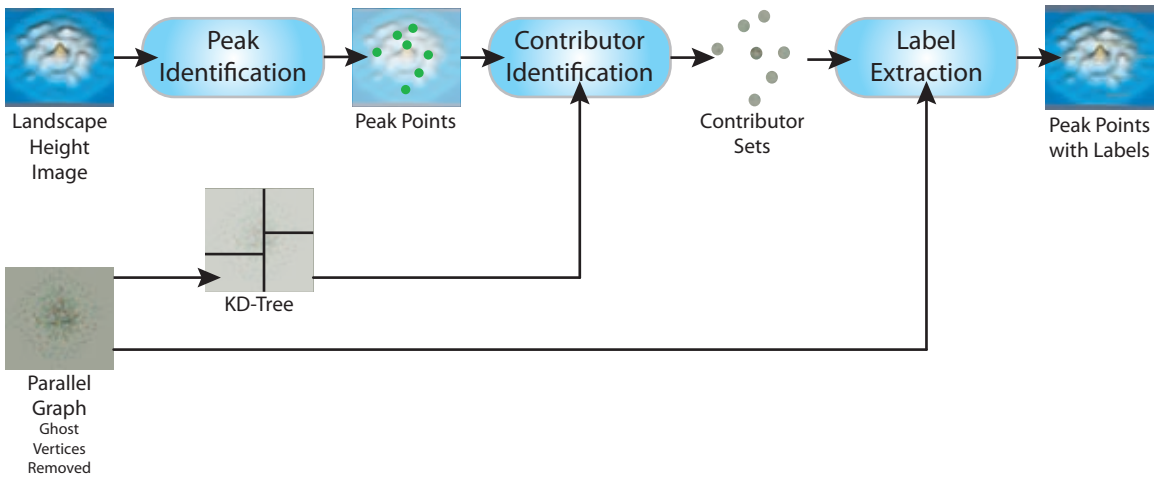


Figure 3.25. Flow for peak identification and labeling.

larger than its immediate neighbors to avoid many peaks in relatively flat areas. We find that a default radius of around 5% of the total landscape works well.

Remember from the previous section on the landscape generation that the landscape geometry created is independent of the graph size and is always small. Thus, the peak identification is performed on one process and the result is broadcast to the rest of the processes.

Contributor Identification

The contributor identification finds all of the vertices located within a certain distance from each peak. Each process, which holds a distinct partition of the graph, performs this operation independently.

To make sure that these vertices can be found quickly, they are placed in a **kd-tree** search structure [5] as a preprocessing step. We also speed up the search by looking for vertices in a square rather than a circle. Although this adds preference to vertices diagonally away, it has little influence on the final result (which is a heuristic measure anyway).

Label Extraction

The label extraction takes each set of vertices "nearby" a peak and finds the words most commonly used in the associated data. These words are then used to build the string for the label.

The parallel processing of the label extraction starts independently on all processes. Each process takes each vertex set and extracts all the words for the identifiers for those vertices. They then count the instances of the words and identify the most common sets of words.

Once independently extracted, these common words sets are aggregated on one of the processes where the order is reevaluated and the final labels are constructed from the most common words.

Chapter 4

Future Work

This LDRD project has successfully laid the groundwork for a scalable, parallel framework that handles massive graph visualization on distributed memory computers. We have demonstrated the ability to run an interactive parallel visualization session where we load and visualize a large graph. Furthermore, this work is integrated with Titan, which lowers the barrier to leveraging this work in other projects.

We have already begun to integrate some of this work in the ThreatView™ application. In fact, it was with this application that we were able to demonstrate the parallel graph visualization. As we move forward, we would like to integrate more of these techniques into the ThreatView™ application and the ParaView framework it sits on. This step requires the completion of the parallelism in some algorithms and the further integration of these parallel components.

Deployment also requires a substantial amount of code hardening. As simple as we try to make integrating components into software, nothing can replace the testing and debugging required to make components production ready. This hardening includes making ThreatView™ ready to function in client-server mode. Although we have demonstrated that running in client-server mode is possible, further testing needs to be done.

Finally, we want to use this work to springboard Sandia into a role of leadership for scalable analytics. We have laid the foundation and built the tools needed to perform scalable information visualization like no one else. From here we need to assert our worldwide presence with publications, self promotion, and community involvement.

References

[1] Jonathan Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IEEE Workshop on Multithreaded Architectures and Applications*, 2007.

[2] Kevin W. Boyack. Mapping knowledge domains: Characterizing pnas. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 101(Suppl. 1), 2004. DOI=10.1073/pnas.0307509100.

[3] Kevin W. Boyack, Richard Klavans, and Katy Börner. Mapping the backbone of science. *Scientometrics*, 64(3):351–374, 2005.

[4] Kevin W. Boyack, Brian N. Wylie, and George S. Davidson. Domain visualization using VxInsight® for science and technology management. *Journal of the American Society for Information Science and Technology*, 53(9):764–774.

[5] Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, second edition, 2000. ISBN 3-540-65620-0.

[6] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[7] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Proceedings of Graph Drawing (GD 2004)*, volume 3383, pages 285–295, 2005. DOI=10.1007/b105810.

[8] David Harel and Yehuda Koren. Graph drawing by high dimensional embedding. In *Proceedings Graph Drawing 2002 (GD'02)*, 2002.

[9] Beth Hetzler, Paul Whitney, Lou Martucci, and Jim Thomas. Multi-faceted insight through interoperable visual information analysis paradigms. In *Proceedings of IEEE Symposium on Information Visualization, InfoVis '98*, pages 137–144, October 1998.

[10] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), September/October 2006.

[11] Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *Conference on Email and Anti-Spam (CEAS) 2004 Proceedings*, July 2004.

[12] Thomas Porter and Tom Duff. Compositing digital images. In *Computer Graphics (ACM SIGGRAPH 84)*, volume 18, pages 253–259, July 1984.

[13] James J. Thomas and Kristin A. Cook, editors. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. IEEE Computer Society Press, 2005. ISBN 0-7695-2323-4.

[14] Jim Thomas, Kris Cook, Vern Crow, Beth Hetzler, Richard May, Dennis McQuerry, Renie McVeety, Nancy Miller, Grant Nakamura, Lucy Nowell, Paul Whitney, and Pak Chung Wong. Human computer interaction with global information spaces - beyond data mining. Technical report, Pacific Northwest National Laboratory, 1999.

[15] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catelyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005. DOI=10.1109/SC.2005.4.

Index

DISTRIBUTION:

4	MS	1323	Moreland,Kenneth, 1424
1	MS	1209	Rahal,Nabeel, 5925
1	MS	1323	Wylie,Brian, 1424
1	MS	1323	Rogers,David H.,1424
1	MS	1323	Shead,Timothy, 1424
1	MS	1323	Crossno,Patricia, 1424
1	MS	1323	Stanton,Eric, 1424
1	MS	9018	Central Technical Files, 8944 (electronic)
1	MS	0899	Technical Library, 9536 (electronic)
1	MS	0123	D.Chavez,LDRD Office, 1011