ZAME: Interactive Large-Scale Graph Visualization

Niklas Elmqvist[*] INRIA
Thanh-Nghi Do[†] INRIA
Howard Goodell[‡] INRIA
Nathalie Henry[§] INRIA, Univ. Paris-Sud & Univ. of Sydney
Jean-Daniel Fekete[¶] INRIA

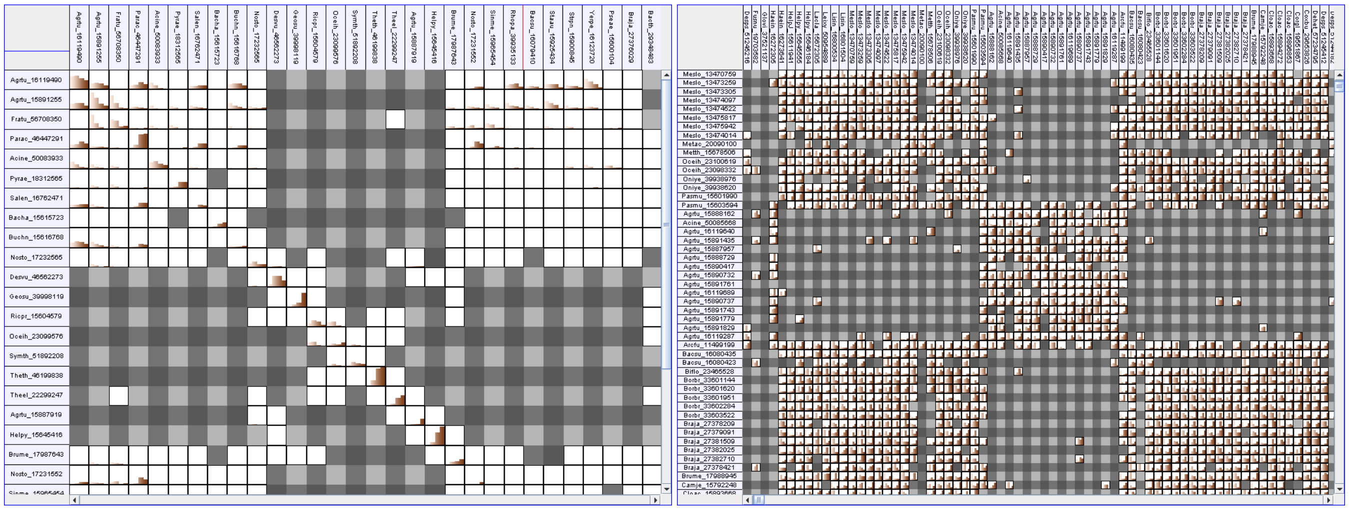

Figure 1: A protein-protein interaction dataset (100,000 nodes and 1,000,000 edges) visualized using ZAME at two different levels of zoom.

Abstract

We present the Zoomable Adjacency Matrix Explorer (ZAME), a visualization tool for exploring graphs at a scale of millions of nodes and edges. ZAME is based on an adjacency matrix graph representation aggregated at multiple scales. It allows analysts to explore a graph at many levels, zooming and panning with interactive performance from an overview to the most detailed views. Several components work together in the ZAME tool to make this possible. Efficient matrix ordering algorithms group related elements. Individual data cases are aggregated into higher-order meta-representations. Aggregates are arranged into a pyramid hierarchy that allows for on-demand paging to GPU shader programs to support smooth multiscale browsing. Using ZAME, we are able to explore the entire French Wikipedia—over 500,000 articles and 6,000,000 links—with interactive performance on standard consumer-level computer hardware.

Index Terms: H.5.1 [Information Systems]: Multimedia Information Systems—Animations; H.5.2 [Information Systems]: User Interfaces; I.3 [Computer Methodologies]: Computer Graphics

[*]e-mail: elm@lri.fr
[†]e-mail: dtnghi@lri.fr
[‡]e-mail: howie.goodell@gmail.com
[§]e-mail: nathalie.henry@lri.fr
[¶]e-mail: jean-daniel.fekete@inria.fr

1 Introduction

Protein-protein biological interactions; the collected articles of the Wikipedia project; contributors to the Linux and other Open Source systems; organization charts for large corporations; the World Wide Web. All of these are examples of large, dense, and highly connected graphs. As these structures become increasingly available for analysis in digital form, there is a corresponding increasing demand on tools for actually performing the analysis. While the scientists and analysts who concern themselves with this work often make use of various statistical tools for this purpose, there is a strong case for employing visualization to graphically show both the structure and the details of these datasets.

However, where visualization in the past has mostly concerned itself with graphs of thousands of nodes, the kind of complex graphs discussed above typically consist of millions of vertices and edges. There exist no general visualization tools that can interactively handle data sets of this magnitude. In fact, on most desktop computers, there are simply not enough pixels to go around to be able to display all the nodes of these graphs.

In this article, we present the Zoomable Adjacency Matrix Explorer (ZAME), the first general tool that permits interactive visual navigation of this kind of large-scale graphs (Figure 1). ZAME is based on a multiscale adjacency matrix representation of the visualized graph. It combines three main components:

- a fast and automatic reordering mechanism to find a good layout for the nodes in the matrix visualization;
- a rich array of data aggregations and their visual representations; and
- GPU-accelerated rendering with programmable shaders to deliver interactive framerates.

The remainder of this article is organized as follows: we first present a survey of related work. This is followed by a description of ZAME's features and design. We conclude with a discussion of its current implementation and some performance results.

2 Related Work

The most crucial attribute of a graph visualization is its *readability*, its effectiveness at conveying the information required by the user tasks [10, 18]. For example, Ghoniem et al. proposed a taxonomy of graph tasks and evaluated the effectiveness of node-link and adjacency matrix graph representations to support them in [16]. They found that graph visualization readability depends strongly on the graph's size (number of nodes and edges) and density (average edges per node). Different visualizations have better readability for different graph sizes and densities, as well as for different tasks.

Several recent efforts visualize large graphs with aggregated representations that present node-link and/or matrix graphs at multiple levels of aggregation. The following sections describe several important ones.

2.1 Node-Link Visualization of Large Networks

Node-link diagrams can effectively visualize networks of about one million vertices if they are relatively sparse. Hachul and Jünger [13] compared six large-scale graph drawing algorithms for 29 example graphs, some synthetic and some real, of which the largest had 143,437 vertices and 409,593 edges. Of the six algorithms, only three scaled well: HDE [14], FM^3 [12] and, to some extent, GRIP [8]. However, the densities of the sample graphs were small, typically less than 4.0. When the density or size grows, dimensionality reduction is needed to maintain readability.

Hierarchical aggregation allows larger graphs to be visualized and navigated, assuming that there is an algorithm for finding suitable aggregations at each level in a reasonable time. An aggregation is suitable if each aggregated level can be visualized effectively as a node-link diagram, and if navigation between levels has sufficient visual continuity for users to maintain their mental map of the whole network and avoid getting lost. No automated strategy published to date can select an appropriate algorithm for an arbitrary network, but there are many successful aggregation algorithms for specific categories of graphs. For example, Auber et al. [3] present effective algorithms for aggregating and visualizing the important class of networks known as small-world networks [21] (which includes the Internet and many social networks), whose characteristics are power-law degree distribution, high clustering coefficient, and small diameter. Systems such as Tulip offer multiple clustering algorithms and are designed to permit smooth navigation on large aggregated networks [2].

Gansner et al. propose another method involving a topological fisheye that is usable when a correct 2D layout can be computed on a large graph [9]. After the network is laid out, it is topologically simplified to reduce the level of detail at coarser resolutions. The fisheye selects a focus node and displays the full detailed network around it, showing the remainder of the network at increasingly coarse resolutions for nodes farther away from the focus. This technique preserves users' mental maps of the whole graph while magnifying (distorting) the network around the focus point. However, effectively laying out an arbitrary large, dense graph remains an open problem for node-link diagrams. All the methods require a good global initial layout, which can be very expensive to compute.

Wattenberg [20] describes a method for aggregating networks according to attributes on their vertices. The aggregation is *only* computed according to the attribute values, much like pivot tables in spreadsheet calculators or data cubes in OLAP databases. This approach works best when the values are categorical or numerical with a low cardinality. The article only refers to categorical attributes on the vertices. When no categorical attribute is suitable for computing the pivots, as with the Wikipedia hypertext network discussed in this paper, this approach is not effective—the whole network would be displayed as a single point. However, this can be avoided by transforming the raw data into derived categorical values that can be analyzed using Wattenberg's method.

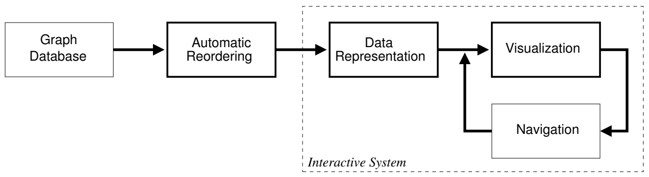

Figure 2: Overview of the components of ZAME.

Overall, the node-link representation has two major weaknesses [10]: (i) it copes poorly with dense networks, and (ii) without a good layout, it requires aggregation methods to reduce the density enough to be readable. Because these methods are very dataset-dependent, current node-link visualization systems leave the choice of aggregation and layout to the users, who therefore need considerable knowledge and experience to get good results.

2.2 Matrix Visualization of Large Networks

Several recent articles have used the alternative adjacency matrix representation to visualize large networks. Abello and van Ham demonstrated [1] the effectiveness of the matrix representation coupled with a hierarchical aggregation mechanism to visualize and navigate in networks too large to fit in main memory. Their approach is based on the computation of a hierarchy on the network displayed as a tree in the rows and columns of the aggregated matrix representation. The aggregation is a hierarchical clustering in which items are grouped but not ordered. It is computed according to memory constraints as well as semantic ones. The users operate on the tree to navigate and understand the portion of the network they are currently viewing.

Navigation in this approach is constrained by the hierarchical aggregation: users navigate in a tree that has been computed in advance. The main challenge is to find an aggregation algorithm that is both fast and that produces a hierarchy meaningful to the user. Unfortunately, this choice is typically dataset-dependent. Without a meaningful hierarchy, users navigate in clusters containing unrelated entries and cannot make sense of what they see.

Mueller et al. describe and compare 8 graph-algorithmic methods for ordering the vertices of a graph [17]. They are either quadratic or reported to be not very good, except the "Sloan" algorithm which has a complexity of $O(\log(m)|E|)$ where m is the maximum vertex degree in the graph.

Henry and Fekete in [15] proposed several methods based on reordering the rows and columns of the matrix as opposed to just clustering similar nodes. They describe two methods based on approximate Traveling Salesman Problem (TSP) solutions, which are computed on the similarity of connection patterns and not on the network itself. One method uses a TSP solver directly; the other initially computes a hierarchical clustering and then reorders the leaves using a constrained TSP. Both algorithms yield orderings that reveal clusters, outliers and interesting visual patterns based on the roles of vertices and edges. Because the matrix is ordered, not just clustered, both navigation (panning) between and within clusters of similar items may reveal useful structure. Unfortunately, both reordering methods are at best quadratic since they need to compute the full distance matrix between all the vertices. Therefore, it is difficult to scale them to hundreds of thousands or millions of vertices.

3 The Zoomable Adjacency Matrix Explorer

ZAME is a graph visualization system based on a multiscale adjacency matrix representation of a graph. Attributes associated with the vertices and edges of the graph can be mapped to visual attributes of the visualization, such as color, transparency, labels, border width, size, etc., using configurable schemes to aggregate each attribute appropriately at higher levels. ZAME integrates all the views of a large graph, from the most general overview down to the details, and provides navigation techniques that operate at all levels. Therefore, it can be used for tasks ranging from understanding the graph's overall structure, to exploring the distribution of results to a content-based search query at multiple levels of aggregation, and to performing analysis tasks such as finding cliques or the most central vertices. Additional relevant graph tasks can be found in [16].

The main technical challenge of the system is managing the huge scale of such graphs—on the order of a million vertices for the protein-protein interaction or Wikipedia datasets—while delivering the real-time framerates necessary for smooth interaction. Furthermore, the graph must be laid out (the matrix nodes reordered) to group similar nodes so that the visualization becomes readable, i.e. in such a way that any high-level patterns emerge and conclusions can be drawn from the data.

To achieve these goals, our tool consists of three main components (see the thick boxes in Figure 2):

- a *hierarchical data structure* for storing the graph and its attributes in an aggregated format;
- an *automatic reordering* component for computing a usable order for the matrix to support visual analysis; and
- an *accelerated rendering mechanism* for efficiently displaying and caching a massive graph dataset.

We also provide a set of *navigation techniques* for exploring the graph. The following sections describe these components in detail.

3.1 Multiscale Data Aggregation

To support multiscale graph exploration with interactive response, we designed an index structure tailored to its visualization abstraction, the three-dimensional binary pyramid of detail levels shown in Figure 3. Users pan across the surface of one detail level; they zoom up and down between detail levels (along perspective lines meeting at the tip of the pyramid, so features match between zoom levels). Detail level zero of this abstraction, the bottom level of the pyramid, is the adjacency matrix of the raw data with the nodes arranged (according to the reordering permutation) on the rows and columns at the edge and the edges between them indicated at the row/column intersections on the level's surface. Every detail level above the base has half the length and width (number of nodes) and a quarter the intersection squares (possible edges) as the level below it. Therefore, each node at a higher, more summary detail level represents four nodes at the level below it (except the last node on a level, in the case where the lower level had an odd number of nodes), and each intersection square indicating a possible edge at this level represents four possible edges at the level below it.

Combining nodes and edges for higher detail levels require that we define how to aggregate the data and its corresponding visual representation. See Sections 3.3 and 3.5 for these definitions.

3.1.1 Pyramid Index Structure

The original network at detail level 0 is stored as a standard graph in the InfoVis Toolkit (IVTK) [7]. The specialized index structure is maintained in zoomable equivalents specialized for the pyramid index structure.

A standard IVTK graph is implemented by a pair of tables, one for its vertices and another for its edges. Each row represents one

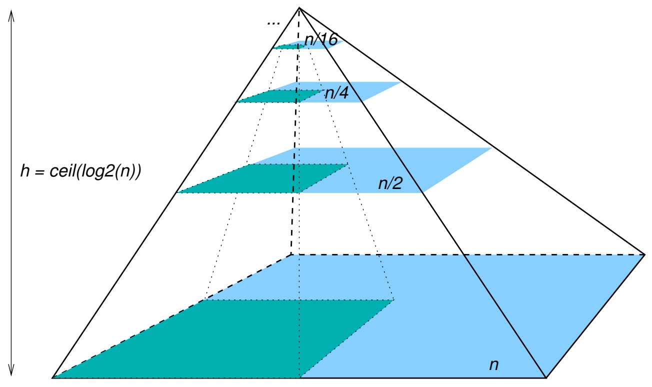

Figure 3: Conceptual structure of the aggregated graph pyramid.

vertex or edge. It contains attributes in internal columns that maintain the graph topology. For each vertex, there is a first and last edge for each of two linked edge lists for its outgoing and incoming edges. Each of these lists is maintained in a pair of columns in the edge table: the "next edge" and "previous edge" for the outgoing and incoming edge lists. To complete the topology, two more columns of the edge table store its first and second vertex (*source* and *sink* edge vertices for directed graphs). The doubly-linked edge lists are an optimization for fast edge removal, but the back-link can be omitted to reduce memory consumption if necessary. Thus, the IVTK needs to store four numbers for each vertex, and either six or four numbers for each edge. Its current implementation uses CERN COLT [6] primitive integer arrays that have very little overhead; so in a 32-bit Java implementation today, the memory consumed is 16 bytes for each vertex and 16-24 bytes for each edge.

A zoomable graph has the same basic structure. However, instead of single-layer vertex and edge tables, it uses specialized zoomable tables with multilevel indices. Also, it maintains some invariants that accelerate important operations. Below is a description of the basic operations:

`getRelatedTable()` returns the original table that the zoomable table aggregates;

`getItemLevel(int item)` returns the aggregation level for a specified item, 0 for the original level and $\lceil \log_2(|V|) \rceil$ for the highest level with only one element ($|V|$ being the number of vertices of the graph);

`getSubItems(int item)` returns a list of items in the next lower aggregation level (two vertices or up to four edges);

`getSuperItem(int item)` returns the corresponding item at the next higher aggregation level; and

`iterator(int level)` returns an iterator over all the elements of a specified aggregation level.

The zoomable vertex table refers to the original graph's vertex table (its related table). Each aggregated vertex at level 0 refers to one vertex in the related table, except that their order is changed by a "permutation" structure that implements the reordering described in 3.2. The numbering of aggregated vertices at level 1 begins immediately after those at level 0, 2 after 1, and so forth (except that an odd number at any level is rounded up by one). Each pair of vertices at level n is aggregated by a single vertex at level $n+1$. So, given the number of a vertex at any level, this simple numbering scheme makes it straightforward to compute corresponding vertices at levels above and below it. It is also straightforward to calculate the size of the index: it is bounded by the series $1/2 + 1/4 + \ldots < 1$; so all the index levels at most double the size of the table. Like vertices at level 0, an aggregated vertex has pointers (edge numbers) to its lists of out- and in-edges, but these refer to aggregated edges.

Unfortunately, the zoomable edge table index is much more complicated to build and maintain, because the set of possible edges is not sparse. The number of possible edges between N nodes is on the order of N^2; so in 32-bit signed arithmetic, calculations based on a simple enumeration of possible edges analogous to those used for vertex indices would overflow for more than 2^{15} (32K) vertices. As with the vertex table, edges at level n follow edges at level $n-1$. However, because the corresponding edge numbers at each level cannot be calculated, they must be stored. Achieving fast access requires several optimizations. To compute the level of an edge, we perform a binary search in a vector containing the starting index of each level. At each level, the outgoing edges are stored in order by their first (source) vertex: the outgoing edges of vertex n follow the outgoing edges of vertex $n-1$. Similarly, all the outgoing edges for one source vertex are sorted in order of their second (sink) vertex. This arrangement makes it very fast to search for an edge given its vertices using two levels of binary search. It also allows us to omit the previous and next edge columns for aggregated edges.

Despite these memory optimizations, aggregated edge indices are still very costly in terms of memory, several times larger than the original data. We also need an extra column to store the "super edge" (that is, the corresponding edge in the level above). This appears wasteful since the "super vertex" corresponding to this endpoint could be calculated and the super edge found by a binary search in the edge list of the super vertex. For example, Wikipedia has around six million edges, so 24 Mb are required just to add this one column to the base level, and the total including all its index levels is around five times more. However, we still chose to store this information, because the basic operation of aggregating edge attributes requires going through each edge from level 0 up and accumulating the aggregated results on the super edge. Without direct access, the complexity of this operation would be $n \times \log(n)$ instead of n, and it would require tens of minutes instead of minutes.

Because the amount of information used for aggregated indices on a huge file such as Wikipedia exceeds the virtual memory capacity of a 32-bit Java Virtual Machine (JVM), we implemented a paging mechanism that allocates columns of memory in fairly large fixed-size chunks that are retrieved from disk on-demand. Fortunately, the memory layout of the zoomable aggregated graph is very well suited to paging. Most operations are performed in vertex- and edge-order on a specified level, so they tend to use consecutive indices likely to be allocated nearby on disk.

The total size of the aggregated edge table depends dramatically on the quality of the ordering. A good ordering groups both edges and non-edges; so multiple nearby edges and non-edges aggregate at each step and the size of successive index levels rapidly diminishes. The worst case is a "salt and pepper" pattern where edges are widely dispersed across the whole matrix. These do not aggregate significantly for many levels, resulting in an aggregated edge table that can be 4 to 8 times larger than the original edge table. Our current reordering methods, though imperfect, improve this to about a factor of 5.

3.2 Reordering

Of all the algorithms described in the literature on matrix reordering, few are sub-quadratic. We experimented with those based on linear dimension reduction such as Principal Component Analysis (PCA) or Correspondence Analysis (CA) and greedy TSP algorithms such as the Nearest-Neighbor TSP (NNTSP) heuristic.

3.2.1 High-Dimensional Embedding

PCA and Correspondence Analysis were used effectively by Chauchat and Risson [5] for reordering matrices. Their matrices were small enough to permit computing the eigenvectors directly, but this is obviously infeasible for hundreds of thousands or millions of points. Harel and Koren describe a modified PCA method called "High Dimensional Embedding" [14] that can efficiently lay out very large graphs by computing only k-dimensional vectors where k is typically 50. This method is designed for laying out node-link diagrams by using the two or three first components of the PCA for positioning of the vertices. We used it and improved it for reordering matrices.

The solution proposed by Harel and Koren consists of choosing a set of *pivot vertices* that the algorithm tries to place near the outer edges of the graph, and to use the graph distances to these pivots as the coordinates of each vertex. With 50 pivots, these coordinates are in 50 dimensions. PCA is computed on these dimensions and the eigenvectors are computed using a power-iteration that converges very quickly in real cases. The BFS algorithm is a simple breadth-first search computed when no edge weights exist; otherwise, it is replaced by a Dijkstra shortest path computation.

However, the original algorithm can choose pivots in a very ineffective way. Consider a large connected network with two very deep subtrees S_1 and S_2, where each leaf in one subtree is very distant from each leaf in the other subtree and even two leaves in the same subtree can be very far apart. HDE will pick a random pivot first, then take the farthest leaf from the pivot, say around S_1. The next vertex will then be a pivot around S_2. In turn, the next will be another vertex around S_1, and so forth, until several children of S_1 and S_2 are enumerated, producing a very biased distribution of pivots. This is not merely a theoretical problem; something similar actually occurs in Wikipedia. Because the taxonomies of the animal and plant kingdoms in biological species classifications are the deepest tree structures in Wikipedia, unmodified HDE merely enumerates some of the deepest leaves of these two classifications. Obviously, any major axes determined by PCA using this highly unbalanced pivot selection will not represent the rest of Wikipedia very well.

To avoid this bias, we modified HDE to penalize edges progressively according to the number of times they already participate in a path between existing pivots (Figure 4). This is done by permanently halving the edges involved in the shortest path selected for each pivot pair. This penalty encourages the algorithm to place new pivots in regions of the graph not traversed by paths between previous pivots, which hopefully represent important different features of its overall structure.

This algorithm avoids the pathological distribution of pivots, but it does not solve the fundamental question of how many pivots are required to adequately represent a large graph, which is still open.

3.2.2 Nearest-Neighbor TSP Approximation

Although the Traveling Salesman Problem is NP-complete, it has good approximations in many cases. When the edges are weighted and the weights are not too similar, the Nearest-Neighbor TSP (NNTSP) approximation algorithms can be effective [11]. An initial ordering can be computed in linear time by limiting the search distance. In our NNTSP algorithm, we limit the distance to 6 since the number of nodes examined grows by a factor of 12 (the average number of links per page in Wikipedia) for each iteration, and we felt that nodes more than 6 hops away were unlikely to be good matches.

Because hypertext links are not weighted, we computed various dissimilarity functions between the source and destination pages of the link to provide them. To date, we have tried three link-weighting or distance-calculation algorithms. In order of increasing quality and running time, they are dissimilarity of adjacency patterns, dis-

```
Function HighDimDraw($G(V = 1,\ldots,n,E),m$)
  % This function finds an $m$-dimensional layout of $G$:
  | % Initialize the penalized edge length $E$ to 1 for each edge
  | $\mathbf{E}[\mathbf{1},\ldots,.\mathbf{n}] \leftarrow \mathbf{1}$
  Choose node $p_1$ randomly from $V$
  $d[1,\ldots,n] \leftarrow \infty$
  for i = 1 to $m$ do
     $d_{p_{i*}} \leftarrow \mathrm{BFS}(G(V,E),p_i)$
     % Compute the $i^{\mathrm{th}}$ coordinate using BFS
     for every $j \in V$ do
        $X^i(j) \leftarrow d_{p_i} j$
        $d[j] \leftarrow \min\{d[j],X^i(j)\}$
     end for
     % Choose next pivot
     $d' \leftarrow 0$
     $p' \leftarrow$ undefined
     for j = 1 to $n$ do
        $d'' \leftarrow d[j]$
        if $j$ is not a pivot and $d'' > d'$ then
           | % Compute the penalized edge length
           | $\mathbf{d}'' = \sum_{\mathbf{e}\in\mathrm{path}(\mathbf{p_i},\mathbf{j})} \mathbf{E}[\mathbf{e}]$
           if $\mathbf{d}'' > \mathbf{d}'$ then
              % $p'$ is the farthest vertex so far
              $d' \leftarrow d''$
              $p' \leftarrow j$
           end if
        end if
        $p_{i+1} \leftarrow p'$
        for all | $\mathbf{e} \in \mathrm{path}(\mathbf{p_i},\mathbf{p_{i+1}})$ do
           | % Penalize all edges in shortest path from $p_i$ to $p_{i+1}$
           | $\mathbf{E}[\mathbf{e}] \leftarrow \frac{\mathbf{E}[\mathbf{e}]}{\mathbf{2}}$
        end for
     end for
     return $X^1,X^2,\ldots,X^m$
  end for
```

Figure 4: Computation of pivots in HDE and our modifications in boldface starting with a bar.

tance in the HDE pivot space and distance in a randomly-selected pivot space. (Dis)similarity of adjacency patterns is determined by merging the set of destination vertices between the edge sets of a pair of source vertices. Visually, an ordering that optimizes this measure will tend to maximize the number of groups of aligned squares on an adjacency matrix representation, which is good for perceptual analysis. This measure can be calculated efficiently—in time linear in the average number of vertices per node—so it was the first one we tried.

NNTSP using this distance measure does a reasonable job of aligning almost all the vertices of Wikipedia—about 97%, approximately the first 460,000 of the 474,594 pages in its largest connected component that we actually analyze (since graph distance cannot be computed for unconnected components). However, at the end of the NNTSP analysis, when relatively few unvisited vertices remain to be placed, the chance of finding pages with common links is small. The adjacency pattern measure is likely to be zero for all of the remaining pages, so we cannot discriminate between them and the search for a match becomes very long.

The second and third methods we tried differ from the first by the adjacency pattern dissimilarity metric. First, they find some number (currently 10) of HDE pivots, respectively either by Harel and Koren's method [14] or randomly. They compute the distances between every graph point and each pivot, which takes about 20 seconds per pivot for Wikipedia. Then, each graph distance is approximated by the Triangle Inequality: the true distance between vertices is less than the sum of the distances between the two vertices and any pivot. The shortest distance is used (corresponding to the closest pivot). Dissimilarity is computed on the distance-to-pivots coordinate space and nearest-neighbor is also computed in that space. These methods find better orderings for the graph than the first (which gives up and accepts a low-quality ordering for the last 3% of the points) at the expense of several minutes longer calculation time.

3.2.3 Results

Figure 5 compares the results of HDE versus NNTSP on a relatively small social network dataset. The HDE method (left side) provides the best overview of the graph at a high, heavily-aggregated level, with the majority of connected nodes grouped in the upper left corner. While the local ordering is poor compared to NNTSP, the calculation is much faster, requiring just a fraction of second instead of 30 seconds required by the standard TSP solver that produced the figure on the right.

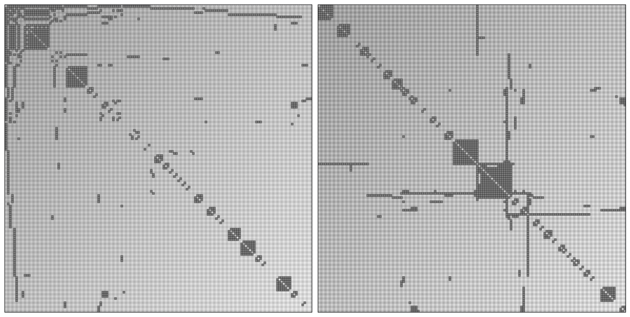

Figure 5: Results of the HDE (left) and NNTSP (right) reordering algorithms applied to a social network.

In contrast, the NNTSP algorithm orders the graph reasonably well globally, but dramatically better locally. As stated above, the global order is not as good as HDE's. This is reflected by a poorer compression factor of the indices, leaving more "salt and pepper" (i.e. unclustered) edges. However, NNTSP's ordering is better at local scale. Basically, it finds many large groups of closely-related nodes, groups them in order, and orders them well. Therefore, it is the default reordering method in ZAME.

3.3 Aggregating Data Attributes

To make views of data at a higher level useful, attributes of the original vertices (such as article names, creation date, or number of edits) and edges (such as link weights) must be combined to provide rapid access to details, including the original data values. This requires considerable flexibility in order to accommodate a range of data types and semantics as well as user intentions. There is a new trend in the classification community to use so-called "symbolic data analysis" for richer aggregation [4]. We summarize the principle and explain how we map symbolic data to visualizations.

It is important to make it evident to the user when a particular cell is representing an aggregated—as opposed to an original—attribute. Section 3.5 describes some of the visual representations we employ for this purpose. Typically, if enough display space is available, a histogram can faithfully visualize the aggregated values for each item. If less space is available, a min/max range or Tukey diagram can be used. In the worst case, where only one or a few pixels are available, the aggregated value can be used to modulate the color shade of the whole cell.

3.3.1 Categorical Attributes

Categorical attributes—including Boolean values—have a cardinality: the number of categories. At the non-aggregated level, a categorical variable can hold one categorical value, such as a US state. When aggregating categorical values, we compute a distribution, i.e. the count of each item aggregated per category.

3.3.2 Numerical Attributes

Numerical attributes may be aggregated by various methods such as mean, maximum or minimum. They can also be transformed into categorical values by binning the values in intervals. Numerical data permits a wide range of analysis such as calculating standard deviations and other statistics. ZAME internally computes a discrete distribution for the aggregated values using a bin width computed according to [19]. We also keep track of the mean, extreme, and median values.

3.3.3 Nominal Attributes

Unlike even ordinal data, there is no inherent relationship between nominal attributes such as article names, authors, or subject titles. Unfortunately, nominal attributes are often vital to understanding what elements of the visual representation refer to. For example, article titles for the Wikipedia dataset are vital for understanding its structure.

Like numeric attributes, nominal attributes of specific datasets can be aggregated using special methods such as concatenation, finding common words, or sampling representative labels. Specifically, in ZAME we aggregate text by simply selecting the first label to represent the whole aggregate. A more general solution is obviously needed for real-world use.

3.4 Visualization

For the visualization component of our matrix navigation tool, we are given an elusive challenge: to efficiently render a matrix representation of a large-scale aggregated graph structure consisting of thousands if not millions of nodes and edges. The rendering needs to be efficient enough to afford interactive exploration of the graph with a minimum of latency.

We can immediately make an important observation: for matrices of this magnitude, it is the screen resolution that imposes a limitation on the amount of visible entities. In other words, there is no point in ever drawing entities that are smaller than a pixel at the current level of geometric zoom. In fact, the user will often want the entities to be a great deal larger than that at any given point in time. This works in our favor and significantly limits the depth we need to traverse into the aggregated graph structure to be able to render a single view of the matrix.

At the same time, we must recognize that accessing the aggregated graph structure may be a costly operation and one which is not guaranteed to finish in a timely manner. Clearly, to achieve real-time framerates, we must decouple the rendering and interaction from the data storage.

Our system solves this problem by utilizing a tile management component that is responsible for caching individual tiles of the matrix at different detail levels. Missing tiles are temporarily exchanged for coarser tiles of a lower detail level until the correct tile is loaded and available. The scheme even allows for predictive tile loading policies, which try to predict user navigation depending on history and preload tiles that may be needed in the future.

In the following text, we describe these aspects of tile management and predictive loading in more depth. We also explain our use of programmable shaders and textures for efficiently rendering matrix visualizations.

3.4.1 Basic Rendering

Instead of attempting to actively fill and stroke each cell representing an edge in our adjacency matrix, we use 2D textures for storing tiled parts of the matrix in video memory. Textures are well-suited for this purpose since they are regular array structures, just like matrix visualizations. Also, they are accessible to programmable vertex and fragment shaders running on the GPU (graphical processing unit) of modern 3D graphics cards. This allows us to avoid sending excessive geometry data to the GPU and instead render a few large triangles with procedural textures representing the matrix.

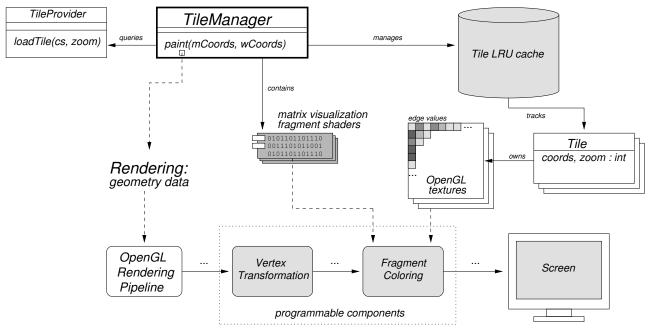

Figure 6: Matrix visualization rendering pipeline.

Our basic matrix rendering shader accesses the texture information for a given position and discards the fragment if no edge is present. If there is an edge there, the data stored in the texture is used to render a color or visual representation (see Section 3.5). Stroking is performed automatically by detecting whenever a pixel belongs to the outline of the edge—in this case, black is drawn instead of the color from the visual representation of the edge.

3.4.2 Tile Management

Adjacency matrices may represent millions of nodes on a side, and thus storing the full matrix in texture memory is impossible. Rather, we conceptually split the full matrix into tiles of a fixed size and focus on providing an efficient texture loading and caching mechanism for individual tiles depending on user navigation.

In our implementation, we preallocate a fixed pool of tiles of a given size. We use an LRU cache to keep track of which tiles are in use and which can be recycled. As the user pans and zooms through the matrix, previously cached tiles can be retrieved from memory and drawn efficiently without further cost. Tiles which are not in the cache must be fetched from the aggregate graph structure—this is done in a background thread that keeps recycling and building new tiles using the cache and the tile pool.

While an unknown tile is being loaded in the background thread, the tile manager uses an *imposter* tile. Typically, imposter tiles are found by stepping upwards in detail zoom levels until a coarser tile covering the area of the requested tile is eventually found in the cache. The imposter is thus an aggregation of higher zoom levels and therefore not a perfectly correct view of the tile, but it is sufficient until the real tile has finished loading.

3.4.3 Predictive Tile Loading

Beyond responding to direct requests from the rendering, our tile caching mechanism can also attempt to predictively load tiles based on an interaction history over time. For example, if the user is increasing the detail zoom level of the visualization, we may try to preload a number of lower-level tiles in anticipation of the user continuing this operation. Alternatively, if the user is panning in one direction, we may try to preload tiles in this direction to make the interaction smoother and more correct.

Our tile manager implementation supports an optional predictive tile loading policy that plugs into the background thread of the tile manager. Depending on the near history of requested tiles, the policy can choose to add additional tile requests to the command queue. Furthermore, the visualization itself can give hints to the policy on the user's interaction.

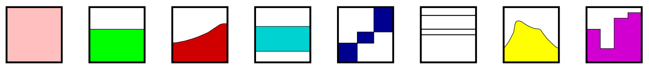

Figure 7: Eight different glyphs for aggregated edges (color shade, average, min/max histogram, min/max range, min/max tribox, Tukey box, smooth histogram, step histogram).

3.5 Aggregated Visual Representations

By employing programmable fragment shaders to render procedural textures representing matrix tiles, we get access to a whole new set of functionality at nearly no extra rendering cost. In our system, we use this capability to render visual representation glyphs for aggregated edges. As indicated in Section 3.3, we can use these to give the user an indication of the data that has been aggregated to form a particular edge.

Currently, we support the following such glyphs (Figure 7 gives examples for each of these):

- **Standard color shade:** Single color to show occupancy, or a two-color ramp scale to indicate the value.
- **Average:** Computed average value of aggregated edges shown as a "watermark" value in the cell.
- **Min/max (histogram):** Extreme values of aggregated edges shown as a smooth histogram.
- **Min/max (band):** Extreme values of aggregated edges shown as a band.
- **Min/max (tribox):** Extreme values of aggregated edges shown as a trio of boxes (the center box signifies the range).
- **Tukey box:** Average, minimum, and maximum values of aggregated edges shown as Tukey-style lines.
- **Histogram (smooth):** Four-sample histogram of aggregated edges shown as a smooth histogram.
- **Histogram (step):** Four-sample histogram of aggregated edges shown as a bar histogram.

Each glyph has been implemented as a separate fragment shader and can easily be exchanged. Furthermore, new representations can also be added. Depending on the availability and interpretation of the data contained in the tile textures, the user can therefore switch between any of these representations at will and with no performance cost.

Figure 8 shows a general overview of the fragment shaders used in our system. The texture representing the matrix tile is first accessed to see whether there is an edge to draw at all; if not, the fragment is discarded and nothing is drawn. The next step is to check whether the current fragment resides on the outer border of a cell, in which case the fragment is part of the stroke and the color black is produced as output. Finally, the last step depends on the actual visual representation chosen, and determines the color of the fragment depending on its position in the cell. The output color can either be the currently active OpenGL color for flat shading, or a ramp color scale indexed using the edge data.

3.6 Navigation

Navigation techniques for the ZAME system control both *geometric zoom* and *detail zoom*:

- **Geometric zoom** encodes the position and dimensions of the currently visible viewport on the visual substrate.

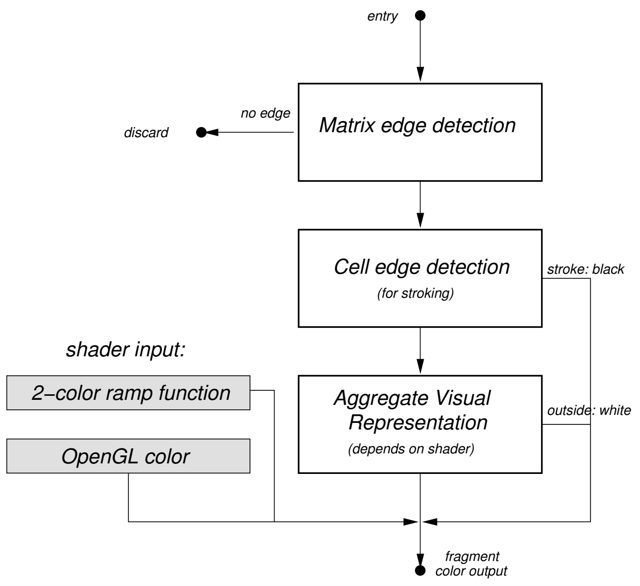

Figure 8: Schematic overview of the glyph fragment shader.

- **Detail zoom** describes the current level of detail of the adjacency matrix.

In other words, the viewport defined by the geometric zoom governs which part of the matrix is mapped to the physical window on the user's screen. This is a continuous measure. The detail zoom, on the other hand, governs how much detail is shown in the window, i.e. at which discrete level in the hierarchical pyramid structure we are drawing the matrix. Since the hierarchy has discrete aggregation levels, detail zoom is also a discrete measure.

ZAME provides all of the basic navigation and interaction techniques of a graph visualization tool. Users can pan around in the visualization by grabbing and dragging the visual canvas itself, or by manipulating the scrollbars.

4 Results

4.1 Implementation

Our implementation is built in Java using only standard libraries and toolkits. Rendering is performed using the JOGL 1.0.0 with OpenGL 2.0 and the OpenGL Shading Language (GLSL). The implementation is built on the InfoVis Toolkit [7] and will be made publicly available as an extension module to this software.

4.2 Performance Measurements

Performance measurements of the different phases of the ZAME system for several graph datasets are presented in Table 1. Figure 9 shows ZAME in use for the French Wikipedia dataset. The measurements were conducted on an Intel Core 2, 2.13 GHz computer with 2 GB of RAM and an NVIDIA GeForce FX 7800 graphics card with 128 MB of video memory. For the navigation, the visualization window was maximized at 1680×1200 resolution.

5 Conclusion and Future Work

This article has presented ZAME, our tool for interactively visualizing massive networks on the scale of millions of nodes and edges. The article describes the technical innovations we introduced:

- a fast reordering mechanism for computing a good layout;
- a set of data aggregations and their visual representations; and

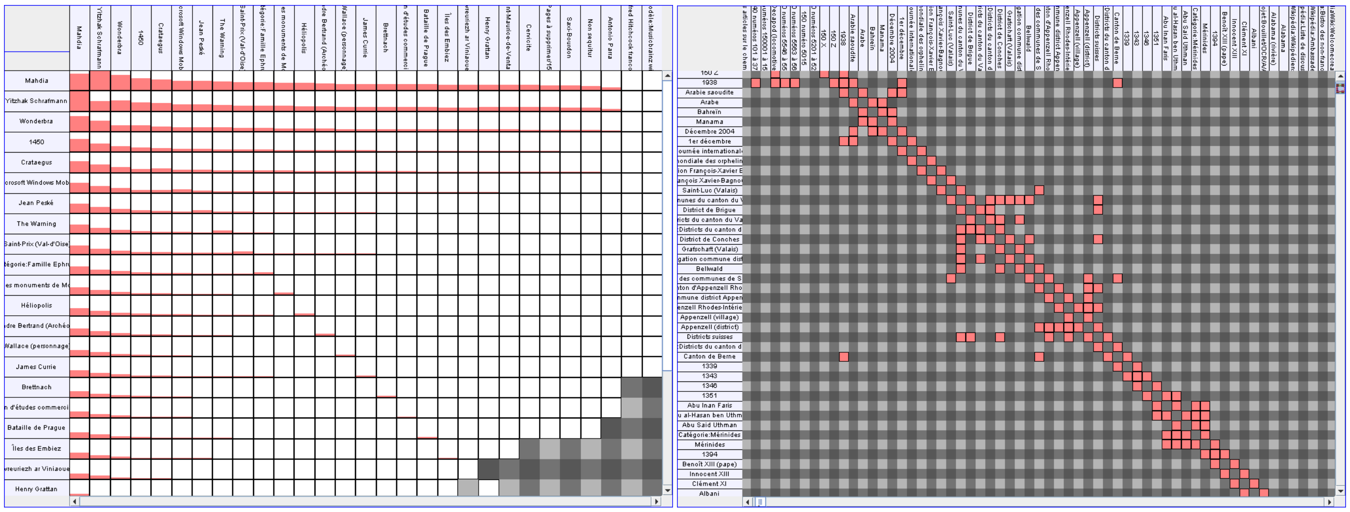

Figure 9: Overview (left image, with aggregation) and detail (right image, no aggregation) from visualizing the French Wikipedia using ZAME.

Dataset	Nodes	Edges	Load (secs)	Order (secs)
InfoVis04	1,000	1,000	10	30
Protein-protein	100,000	1,000,000	10	30
Wikipedia (French)	500,000	6,000,000	50	50

Table 1: Performance measurements for standard graph datasets. Rendering performance was interactive (>10 frames-per-second) for all three datasets.

- GPU-accelerated rendering with shader programs to deliver interactive framerates.

In the future, we intend to continue exploring the problem of multiscale navigation and how to provide powerful yet easy-to-use interaction techniques for this task. We will also investigate the use of additional matrix reordering algorithms, such as the "Sloan" algorithm described in [17]. Furthermore, we are interested in exploring the human aspects of detail zoom versus geometric zoom and suitable policies for coupling these together. We would also like to study the utility of the aggregate visual representations introduced in this work.

Acknowledgements

This work was supported in part by the French Autograph, Microbiogenomics, and SEVEN projects funded by ANR.

References

[1] J. Abello and F. van Ham. MatrixZoom: A visual interface to semi-external graphs. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 183–190, 2004.

[2] D. Auber. Tulip: A huge graph visualisation framework. In *Graph Drawing Software*, pages 105–126. Springer-Verlag, 2003.

[3] D. Auber, Y. Chiricota, F. Jourdan, and G. Melancon. Multiscale visualization of small world networks. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 75–81, 2003.

[4] L. Billard and E. Diday. *Symbolic Data Analysis: Conceptual Statistics and Data Mining*. Wiley, Jan. 2007.

[5] J. Chauchat and R. A. AMADO, a new method and a software integrating Jacques Bertin's Graphics and Multidimensional Data Analysis Methods. In *International Conference on Visualization of Categorical Data*, 1995.

[6] The Colt project. http://dsd.lbl.gov/ hoschek/colt/.

[7] J.-D. Fekete. The InfoVis Toolkit. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 167–174, 2004.

[8] P. Gajer and S. G. Kobourov. GRIP: Graph drawing with intelligent placement. In *Proceedings of the International Symposium on Graph Drawing*, pages 222–228, 2000.

[9] E. R. Gansner, Y. Koren, and S. C. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):457–468, 2005.

[10] M. Ghoniem, J.-D. Fekete, and P. Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[11] G. Gutin and A. Punnen. *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers, Dordrecht, 2002.

[12] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm (extended abstract). In *Proceedings of the International Symposium on Graph Drawing*, pages 285–295, 2004.

[13] S. Hachul and M. Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *Proceedings of the International Symposium on Graph Drawing*, pages 235–250, 2006.

[14] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. In *Proceedings of the International Symposium on Graph Drawing*, pages 207–219, 2002.

[15] N. Henry and J.-D. Fekete. MatrixExplorer: a dual-representation system to explore social networks. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of Visualization/Information Visualization 2006)*, 12(5):677–684, 2006.

[16] B. Lee, C. Plaisant, C. S. Parr, J.-D. Fekete, and N. Henry. Task taxonomy for graph visualization. In *Proceedings of BELIV'06*, pages 82–86, 2006.

[17] C. Mueller, B. Martin, and A. Lumsdaine. A comparison of vertex ordering algorithms for large graph visualization. In *Proceedings of Asia-Pacific Symposium on Visualization*, pages 141–148, 2007.

[18] C. Mueller, B. Martin, and A. Lumsdaine. Interpreting large visual similarity matrices. In *Proceedings of Asia-Pacific Symposium on Visualization*, pages 149–152, 2007.

[19] D. W. Scott. On optimal and data-based histograms. *Biometrika*, 66:605–610, 1979.

[20] M. Wattenberg. Visual exploration of multivariate graphs. In *Proceedings of the ACM CHI 2006 Conference on Human Factors in Computing Systems*, pages 811–819, 2006.

[21] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.