Dynamic Fisheye Views: Combining Dynamic Queries and Mapping with Database Views

by

Emanuel G. Noik

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Department of Computer Science
University of Toronto

© Copyright by Emanuel G. Noik 1996

Abstract

Dynamic Fisheye Views:
Combining Dynamic Queries and Mapping with Database Views

Doctor of Philosophy, 1996
Emanuel G. Noik
Department of Computer Science
University of Toronto

Data visualization refers to the graphical presentation of information with the aim of providing the viewer with a qualitative understanding of the information contents. *Dynamic mapping* (DM) and *dynamic queries* (DQ) are two complementary visualization techniques that aid the understanding of multidimensional data. In order to be effective, both DQ and DM require suitable multidimensional data which sometimes needs to be derived from existing data. In addition, DQ cannot support similarity queries because they do not incorporate similarity functions. A set of techniques called *Fisheye views* (FEVs) use similarity functions, but suffer from specialized hard-coded notions of similarity, and static, limited presentation strategies.

In this thesis we describe a novel approach that simultaneously simplifies and generalizes the notion of FEV and extends the power of DQ. We propose a view definition language, Dye, based on an object algebra and demonstrate how this language can be used to define and to efficiently evaluate a class of derived attributes that capture notions of similarity. We then show that incorporating such views with DQ and DM:

- extends the power of DQ by allowing similarity queries to be posed by pointing and clicking, potentially converting cognitive tasks into perceptual tasks; and,

- simplifies and generalizes the notion of FEV; a broad variety of *dynamic FEVs* can be created by applying DQ and DM to similarity attributes.

As proof of concept, we have implemented an extensive graph visualization system (Graphite) that combines DQ, DM, flexible automatic graph layout, and an active database view component that implements Dye. In this thesis, we demonstrate how Graphite is used to pose similarity queries and to generate dynamic FEVs through several representative sample applications in which data is visualized using plots, flat and nested hierarchical graphs, and geographic and abstract networks.

Acknowledgements

During the past five years I have had the pleasure and good fortune to meet and work with many talented and supportive individuals.

First I would like to thank my office mates and colleagues, Dimitra Vista and Masum Hasan, for their friendship, help, and patience, especially during the months leading up to my defence. Gene Golovchinsky has likewise provided me with much needed feedback not to mention excellent company for the occasional social beer and pool game! Thanks also to Nipon Charoenkipkarn and Eric Harley who agreed to be "guinea-pigs" where my visualization tools were concerned, and who provided valuable feedback.

I also owe a very special thank-you to Andrew Csinger and Sue Rathie, who played the multiple roles of friends, business partners, proof-readers, and "shrinks" admirably. I wish them both many successes in the future and wherever our lives should take us I know that they will remain with me forever.

Next, I would like to thank Arthur Ryman; as my manager at IBM Canada he taught me many things, not the least of which was the value of a graduate education. Through the IBM Canada Centre for Advanced Studies, Arthur was also single-handedly responsible for bringing me together with Alberto Mendelzon, my graduate advisor.

I owe much to Alberto, who always backed me, both financially and morally, and allowed me unlimited freedom to explore and to grow. Although the path that I chose for myself may not have been the easiest and at times a rather lonely one, Alberto gave me the opportunity to challenge myself and to succeed, and for that I will always remain grateful.

Over the years I have also grown indebted to Mark Chignell, both professionally and personally. The relationships developed between the database group in Computer Science and Mark's human factors group in Industrial Engineering have been stimulating and engaging, and I am grateful to Mark and Alberto for their continued support.

In addition to Alberto and Mark, I would like to thank the other members of my thesis committee: Tony Bonner, Derek Corneil, and Eugene Fiume, who have been involved from day one and have greatly facilitated the entire process. Stephen Eick graciously agreed to read my thesis and to join the committee as an external examiner. Unbeknownst to him, the work performed by Stephen's group at Bell Laboratories (now Lucent Technologies) has greatly influenced my own work and the shape of this Ph.D. thesis.

I cannot fail to mention the members of the computer science staff. In particular, Martha Hendriks, Joan Allen, Inge Weber, and Kathy Yen have always greeted me with smiles and were always available to help me negotiate the administrative mazes. Similarly, the doors of computer systems administrators Sandra Smith and Ken Lalonde were always open to me.

Finally, my dear parents. I owe so much to them that if I tried to enumerate my many debts I am afraid that I

could easily produce a second dissertation (God forbid!). The negative aspects of being an only child were most prominent during my five year tenure, as I believe that even when separated by thousands of kilometres the three of us laboured together. As such, the successful completion of my Ph.D. is a joint accomplishment in addition to a personal one. Mom and dad, I couldn't have done it without you, and I am very happy that we are able to share the fruits of our labour together.

To reiterate my sentiments upon completing my Master's degree, I have to say that I have been very lucky, very lucky indeed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data visualization refers to the graphical presentation of information with the aim of providing the viewer with a qualitative understanding of the information contents.

One strategy to aid understanding of multidimensional data is to map data dimensions or *attributes* (*e.g.,* temperature, population, location) to graphic properties (*e.g.,* colour, size, position). *Dynamic mapping* (DM) is an interactive technique that permits mappings to be adjusted dynamically.

The *dynamic query* (DQ), a related interactive technique, applies the principles of direct manipulation to the searching of multidimensional data:

- visual presentation of query's components (*e.g.,* as sliders or buttons);
- visual presentation of results;
- rapid, incremental, and reversible control of queries;
- selection by pointing, not typing; and,
- immediate and continuous feedback.

User experiments have shown DQ to be superior to traditional query approaches for many searching tasks [3, 64, 66, 104, 121].

DQ and DM are complementary techniques and have been successfully integrated in a number of recent visualization (hereafter DQM) systems (*e.g.,* [2, 3, 8, 11, 12, 26, 27]). However, both techniques share a common limitation: their effectiveness is inseparably tied to the availability of suitable multidimensional data (to query and map).

Two types of approaches have been proposed to overcome this limitation: to extend the expressive power of DQ (which permits existing attributes to be combined in more powerful ways); and to create additional *derived* attributes. An example of the former is the filter/flow boolean query interface [104, 105] which permits users to combine attributes with boolean operators; an example of the latter is the Aggregate Manipulator [38] which permits users to interactively define a limited class of derived attributes.

A second limitation of existing DQ systems stems from the fact that they do not incorporate similarity functions. That is, while existing DQ systems are well-suited to tasks that involve identifying data points that satisfy selection criteria, they offer little or no help in solving tasks that involve identifying data points that are "similar" to one or more selected data points. The absence of similarity functions means that many conceptually simple searching tasks can require substantial cognitive effort. Examples of such tasks include:

- *emphasize all cities that belong to the same state as one or more selected cities*;
- *identify files that are as large as one or more selected files*;
- *find flight destinations that can be reached inexpensively or quickly from one or more departing cities*; and,
- *find movies that are similar to "Star Wars"* (*e.g.,* by incorporating some combination of plot, main actors, director, studio, year, style, etc.).

A set of techniques that do employ similarity functions[1] are a variety of *fisheye views* (FEVs). In FEVs, similarity functions are used to assign to each point in the visualization a number that reflects the user's *degree of interest* in that point relative to one or more user-selected (focal) points. Most FEV techniques can be divided into two categories: filtering FEVs that simplify large visualizations by hiding "uninteresting" points; and distorting FEVs that distort the sizes and positions of points in the visualization in an attempt to smoothly integrate local detailed views with global contextual views. As such, existing FEV techniques are subject to two key limitations: specialized hard-coded notions of similarity, and static and limited presentation strategies.

In this thesis we describe a novel approach that simultaneously simplifies and generalizes the notion of FEV and extends the power of DQ. In particular, we propose a view definition language, Dye, based on an object-oriented algebra with object-preserving semantics, and demonstrate how this language can be used to define and to efficiently evaluate derived attributes, including a class of dynamic attributes whose values are efficiently updated in response to user actions; *similarity attributes* are dynamic attributes that capture notions of similarity and are incrementally updated when users manipulate focal points.

Next, we show that incorporating such active views with DQ and DM:

- extends the power of DQ by allowing similarity queries to be posed by pointing and clicking, thereby potentially converting previously challenging cognitive tasks into simple perceptual tasks (*e.g.,* recognizing sets of red points); and,
- simplifies and generalizes the notion of FEV; with this approach filtering FEVs are created by applying DQ to similarity attributes; distorting FEVs are created by mapping similarity attributes to position and size properties with DM; and more general types of FEVs are created by mapping similarity attributes to other graphic properties (*e.g.,* the hue and saturation components of colour).

[1] The terms *priority* and *proximity* have been used synonymously; *distance* is the inverse of *proximity*. We prefer the more neutral *similarity* which is free of temporal or spatial connotations.

Finally, one desirable side-effect of our incremental view maintenance approach is the ability to efficiently update views in response to database updates (*i.e.,* insertions and deletions of data points). As such, this represents an indirect contribution as existing DQM systems do not permit data to be updated while it is being visualized (and queried).

As proof of concept, we have implemented an extensive data visualization system (Graphite) that combines DQ, DM, flexible automatic graph layout, and an active database view component that implements Dye. Graphite depicts multidimensional data using several common techniques, including plots [7, 14] (that show relationships among data dimensions), node-and-link diagrams or graphs [22, 44] (that show relationships among individual data points), and nested graphs or hygraphs [16] (that show hierarchical relationships among points by visual containment). While explicit Dye view definitions make it possible to encode a broad class of similarity functions, Graphite also allows users to define a restricted class of similarity attributes interactively. By contrast, both the filter/flow boolean query interface [104, 105] and the Aggregate Manipulator [38] are too weak to express similarity attributes.

The remainder of the dissertation is organized as follows.

In Chapter 2 we introduce the main features of the Graphite visualization system through a set of examples.

In Chapter 3 we describe related work in two distinct areas, systems that employ DQ and FEV techniques.

In Chapter 4, we describe our approach. In particular, we describe the Dye view definition language, outline the high-level architecture of an active DQM system that supports dynamic attributes, and then describe an incremental view maintenance strategy that permits active views to be maintained efficiently. We conclude Chapter 4 by showing how the proposed language can be used in the context of such active DQM systems to simulate existing fisheye view approaches.

In Chapter 5 we summarize five sample applications, demonstrating how each application can benefit from the incorporation of similarity queries and dynamic FEVs. In particular, we look at the following representative sample applications:

- transportation networks (airline flights data);
- abstract networks (World-Wide Web structure visualizations and (biological) genome structure graph visualizations);
- geographic information systems (U.S. cities); and,
- hierarchical data visualization (Unix file directories).

Chapter 6 covers lessons learned, summary of limitations, topics for future work, summary of contributions, and concluding remarks.

Appendix A contains a more detailed description of the Graphite implementation and Appendix B a collection of selected algorithms.

Chapter 2

A Tour of the Graphite Visualization System

Graphite is a system that we created to manipulate, visualize, and query multi-dimensional graphs (node-and-link diagrams) and hygraphs [16]. It combines structural visualization, which aids in understanding the topological structure of the graphs, and quantitative visualization of highly-dimensional data, which aids in exploring statistical relationships, trends, and exceptions in data.

The system incorporates a number of interactive techniques including DQ, DM, brushing (highlighting, labelling, linking), pan and zoom, and flexible automatic graph layout.

Furthermore, Graphite incorporates a novel database view definition language, Dye, which permits it to simulate a variety of existing fisheye view techniques, and to pose similarity queries by direct manipulation thereby significantly extending the functionality of the DQ approach.

2.1 Multidimensional Graphs

A *graph* $G = \langle V(G), A(G) \rangle$ consists of a set of *vertices* $V(G)$, and a set of *arcs* $A(G) \subseteq V(G) \times V(G)$. There are a number of nested graph formalisms (graphs in which vertices may be nested inside other vertices). A hygraph [16], is a triple $G = \langle V(G), A(G), B(G) \rangle$, where $B(G) \subseteq V(G) \times 2^{V(G)}$ is a set of *blobs*. A blob associates a (container) vertex with a set of vertices that it contains; we assume that the containment relation is hierarchical.

A graph *diagram* or *layout* is a visualization of the graph in which vertices are depicted as *nodes* and arcs as *links* (hence the common name *node-and-link* diagram). In hygraphs, if the blob containment relation is hierarchical, blobs are easily depicted by visual containment (*e.g.,*by nesting closed rectangular or polygonal regions). Graph layouts may be hand-drawn or may be generated automatically by a *graph layout algorithm* [22].

Multidimensional graphs are graphs that arise from multidimensional data – data in which each data point has multiple attributes (or dimensions). Data attributes can be of different data types; there have been a number of efforts at characterizing data for the purpose of automatic design of graphical and textual displays. Among these A Presentation Tool (APT) [67] and SAGE [90, 91] use knowledge-based approaches for synthesizing appropriate visualizations by combining common "business" visualizations such as bar charts and x-y charts, as does A Network Diagram Designer (ANDD) [70] for node-and-link diagrams. These approaches characterize data into at least three main types: *quantitative sets* in which elements are ordered numerically; *ordinal sets* in which elements ordered explicitly with no intermediate values (*e.g.,* a set of ratings {poor, fair, good, excellent}; and *nominal sets* in which elements are unordered (*e.g.,* a list of brand names).

In Graphite we focus primarily on the role of quantitative data in DQ, DM, and especially, in deriving similarity attributes used to generate dynamic FEVs. Thus the Graphite model of graphs is the hygraph in which each vertex, arc and blob may possess some finite number of quantitative attributes.

As an example, consider an airline flights database, which might contain two types of objects: a set of cities and a set of flights between the cities. Each city object has quantitative attributes: longitude and latitude, and each flight object has quantitative attributes: price and duration (of a one-way ticket/trip). This database can be visualized as a graph by depicting each city as a node and each flight as a link between the corresponding cities. The graph so constructed is multidimensional, since each node and link has additional quantitative attributes that could be used to pose dynamic queries, to bind to graphic properties, and to derive additional attributes. A sample airline flights database visualization is shown in Figure 5.1 on page 66.

2.2 Dynamic Queries

Since our specification of dynamic FEVs relies primarily on the use of quantitative attributes, the primary DQ component in Graphite is the frequency distribution histogram, which we've used in a similar manner to that in [8, 26, 27].

Each histogram represents the distribution of a single numeric attribute. The range of the attribute is divided into some (user-configurable) number of equal subranges. Each such subrange is represented by a histogram bar, the height of which is proportional to the number of objects for which the value of the corresponding attribute lies in the associated subrange. The elements associated with a given histogram bar are said to be *selected* by the bar.

Frequency distribution histograms support a number of brushing operations: [7]

- *highlighting:* points contained in the brush are coloured differently from other points to make them stand out;
- *masking:* points contained in the brush are not displayed;
- *linking:* a special type of highlighting operation in which points that are highlighted in one display can be automatically highlighted in other display(s); and,
- *moving average:* display the average value of points currently contained by a brush.

Most DQ systems implement a form of masking, so DQ can be viewed as a restricted form of brushing in which the brush does not have a physical boundary and whose properties are manipulated by DQ components such as sliders, radio buttons, or, as done in our system, frequency distribution histograms.

Highlighting is performed by moving the mouse cursor over individual histogram bars. The effect of this action is to automatically highlight all selected elements (nodes or links) and to display the number of elements in the selected histogram bar and their average value (moving average), as shown in Figure 2.1.

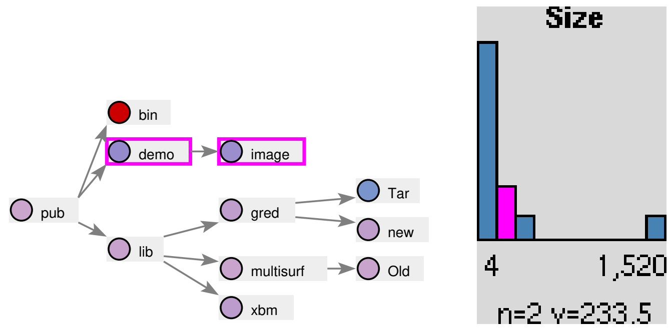


Figure 2.1: Highlighting with histograms. Placing the mouse cursor over a histogram bar causes the associated elements to be immediately highlighted; the number of highlighted elements and their average value is displayed below the histogram.

Masking is performed by pointing at a histogram bar and clicking. The effect of this action is to mask all selected elements, as shown in Figure 2.2. Notice that the histogram masking mechanism is more powerful than that of a simple slider. A slider can be used to mask elements with values less than the threshold chosen with the slider (simple range query). Even with a two-ended slider [2], the user can only mask two contiguous subranges of a given attribute (range query with two inequalities). With a histogram, however, it is possible to select any of 2^n masking combinations, assuming that the histogram has n non-empty histogram bars.

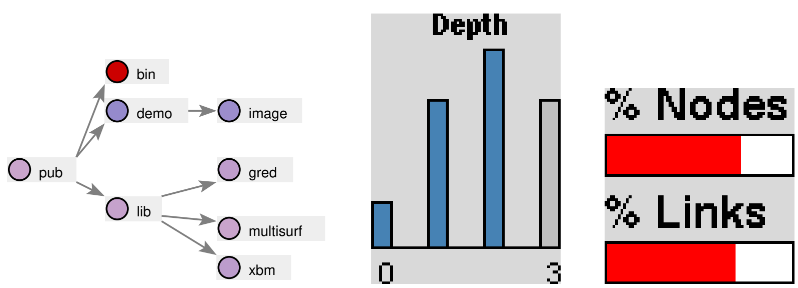


Figure 2.2: Masking with histograms. Pointing and clicking on a histogram bar causes the associated elements to be masked; clicking once more causes the masked elements to be redisplayed. Two horizontal meters show the proportion of the visible nodes and links (the size of the result of the dynamic query).

Linking is performed by pointing at a node icon or a link. The effect of this action is to immediately highlight all associated histogram bars, to display the number of objects associated with each highlighted bar, and to display the exact attribute value associated with the selected element. This is shown in Figure 2.3.

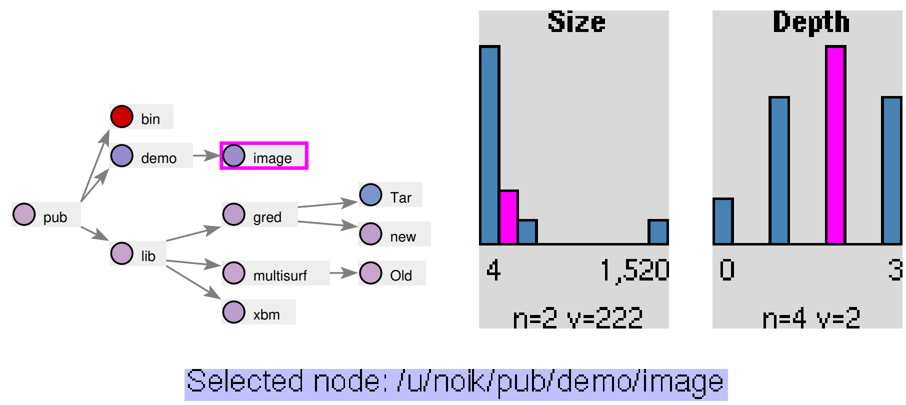


Figure 2.3: Linking and labelling with histograms. Placing the mouse cursor over a node icon or link causes the associated bars of every histogram to be highlighted and the selected object's attribute values to be displayed. The selected object's label is displayed in full on the graph, and its type and identifier are displayed in a message box. In this example node *image* was selected.

2.3 Dynamic Mapping

In *mapping*, data attributes are mapped onto graphic properties [2, 7, 8, 14, 48, 84]. For example, the temperature of a data point is often mapped onto colour using a heat scale, the rate or volume of flow through a network is often depicted by link thickness, and the population of a city is often conveyed by node size. *Dynamic mapping* refers to the ability to assign and modify mappings at run-time rather than fixing mappings *a priori*.

In designing DM systems, it can be important to incorporate a variety of results relating to human perceptual abilities, such as those reported by Cleveland [13] and Tufte [111]. Several approaches for using colour and *colour containers* in DM environments are proposed by Bardon [5]. Csinger [17] provides a nice overview of the psychology of visualization.

In Graphite, DM bindings are assigned by first selecting a node or link graphic property from the node attribute or link attribute menu, as shown in Figure 2.4, and then selecting the data attribute to be mapped onto the previously selected graphic property from a list of available attributes as shown in Figure 2.5.

2.4 Pan and Zoom

The Graphite graph editor provides facilities for panning (scrolling) and position-based zooming of graph layouts. The latter implies that zooming does not affect node sizes – only the relative positions of nodes are scaled. This has the effect of spreading closely-positioned or overlapping nodes apart when zooming in; if, on the other hand, node sizes were also scaled, overlapping or tightly packed groups of nodes would remain thus independently of zoom factor, thereby defeating the goal of zooming. Note that the Graphite pan-and-zoom features essentially reproduce the functionality of the Starfield display [2] (a 2-D scatterplot with pan-and-zoom feature), although our current implementation cannot generate smooth continuous zooming. Our pan-and-zoom feature does, however, support

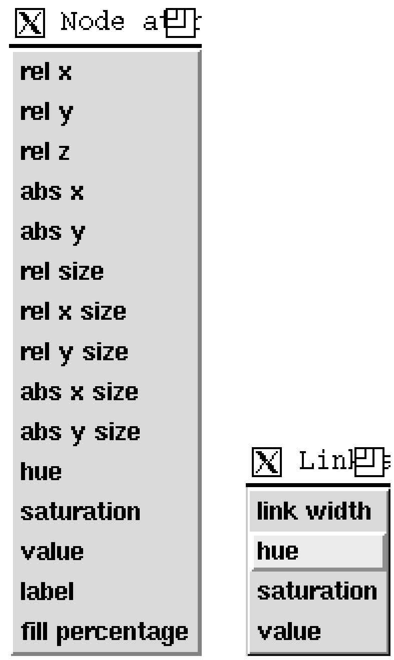


Figure 2.4: Selecting graphic properties for dynamic mapping. A quantitative vertex or arc attribute can be mapped onto any of the above graphic properties; node properties are shown on the left, link properties on the right.

Figure 2.5: Selecting data attributes for dynamic mapping. Once a graphic property has been selected, the user is presented with a list of available vertex or arc attributes; *reset* can be selected to remove a previously established binding; a list of available vertex attributes in the airline flights database is shown on the left, arc attributes on the right (notice that most of the available attributes are derived).

links among nodes whereas the Starfield display was limited to visualizing flat databases (as two-dimensional scatterplots).

Figure 5.18 on page 82 shows a geographic information system visualization while Figure 5.22 on page 86 shows a zoomed view centered on the Los Angeles area.

2.5 Regular Expression Matching

Graphite permits regular (text) expression matching over the identifiers and labels of vertices and arcs. The results of matching can be assigned to a derived attribute which in turn can participate in dynamic queries just as any other attribute. Unlike ordinary attributes, however, such derived attributes are updated dynamically every time the user modifies the regular expression string. Regular expressions are submitted using the dialog box shown in Figure 2.6, which allows the user to choose that matching should be performed on the identifiers or labels of either the nodes or links in the graph. A related technique to select alphanumeric strings is the Alphaslider [1].

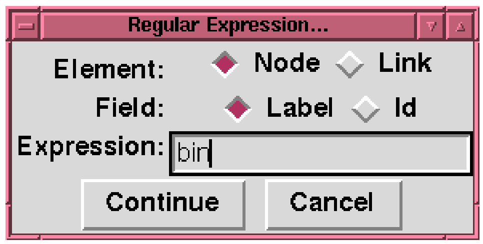


Figure 2.6: Regular expression matching. This dialog box allows users to specify a regular expression search on either the identifiers or labels of either the nodes or links in the graph.

2.6 Similarity Attributes

The novel feature in Graphite is its support for dynamically evaluated similarity attributes – a special class of attributes which can be used to pose similarity queries with DQ and to generate fisheye views with DM.

Similarity attributes are derived in one of two ways: through explicit views defined with our proposed Dye view definition language (described in Chapter 4), or through an interactive dialog box that enables users to define notions of similarity by manipulating a set of sliders (described in the following Section). Although explicit view definitions permit users to define potentially ineffective or nonsensical similarity functions, we intentionally chose not to restrict the Dye language so as not to sacrifice expressiveness. A family of "reasonable" similarity functions can be obtained with our interactive dialog box (next Section); an even more limited class of similarity functions can be obtained by restricting this technique to instances of the Minkowski metric [95].

Given one or more similarity attributes, users can interactively select subsets of nodes or links to be focal points by pointing and clicking. Each click that adds or removes a focal point causes the values of all similarity attributes to be dynamically and incrementally re-evaluated.

Following the airline flights visualization shown in Figure 5.1 on page 66, Figure 5.2 on page 67 shows the result of using DQ to pose a similarity query of the form: *show me all cities that are similar to Toronto and London*, where similarity is inversely proportional to the duration of the quickest trip between two cities, as captured by the *TripDuration* similarity attribute. Being a similarity attribute, the values of *TripDuration* are recomputed every time the user selects or deselects a focal point (city). Nevertheless, notice that any similarity attribute can participate in dynamic queries just as any other base or derived attribute.

Returning to the original visualization of Figure 5.1, notice that the *TripDuration* similarity attribute has been dynamically mapped onto the node percentage fill graphic property causing nodes that correspond to cities that are temporally far from the current set of focal points to appear filled, and those that are temporally close to the set of focal points to appear empty. This is an example of an adorned fisheye view – a fisheye view in which graphic objects are emphasized through non-geometric graphic properties such as colour, thickness, and fill percentage. As before, as the user selects or deselects focal points, the *TripDuration* similarity attribute is automatically recomputed. In addition, since this attribute is dynamically mapped onto the node percentage fill property, the display is also automatically updated every time a focal point is selected or deselected, causing the fill property of nodes to reflect the current similarity attribute values.

Figure 5.25 on page 88 shows a more common fisheye view type – an orthogonal distorting fisheye view – which is taken from a Geographic Information Systems application and is generated by mapping similarity attributes onto the node *relative position* and *relative size* graphic properties.

2.7 User-Configurable Similarity Attributes

Although users can define relatively arbitrary notions of similarity by explicitly specifying them in a Dye view definition, Graphite also possesses a novel feature that allows users to manipulate a more limited form of similarity function interactively. This notion of similarity is based on computing sums of absolute normalized differences in attribute values, and is particularly useful in data exploration tasks where it is difficult to determine appropriate notions of similarity *a priori*.

The user-defined similarity function that is used to define the similarity between data point p and focal point f has the following general form:

$$mag(f) \cdot \left(\sum_{i \in I} E_i \right)^e$$

and

$$E_i = \begin{cases} s_i \cdot (1 - |f_i - p_i|)^{e_i} & \text{if } s_i \geq 0 \\ |s_i| \cdot (|f_i - p_i|)^{e_i} & \text{otherwise} \end{cases}$$

if attribute i is quantitative, or

$$E_i = \begin{cases} s_i & \text{if } f_i = p_i \\ 0 & \text{otherwise} \end{cases}$$

if attribute i is ordinal or nominal, where:

- $mag(f) \in [-1, 1]$ is a user-supplied focal point magnification (relevance) value of focal point f;
- I is a (sub)set of attributes;
- $s_i \in [-1, 1]$ are scale factors;
- $e, e_i \geq 0$ are exponent values; and,
- f_i and p_i are the normalized (scaled to the interval $[0, 1]$) values of attribute i of the focal point and data point, respectively.

Notice that the general form of our similarity function for quantitative attributes is very similar to the more restrictive Minkowski metric [95]:

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

When $p = 2$, the Minkowski metric gives rise to the Euclidean distance in n-dimensional space, and when $p = 1$ it is the "city-block" distance.

Graphite realizes user-configurable similarity functions by dialog boxes containing a number of sliders as shown in Figure 5.19 on page 83, in which the user can express notions of similarity in terms of five node attributes. Note how the combination of settings shown in the example, (sum of difference in *Longitude* squared and difference in *Latitude* squared raised to the power 0.5) gives rise to the familiar Euclidean distance expression; the remaining attributes are ignored by setting their scale factors to 0.

The three buttons labelled *None*, *All*, and *Reset* are used to deselect all attributes, select all attributes equally, and restore previous settings, respectively. Furthermore, the slider text and trough areas are coloured in a manner intended to give users feedback as to the role of each attribute in the similarity function: scale factor sliders are coloured gray when set to 0, increasing saturation of green when set to positive values, and increasing saturation of red when set to negative values; exponent sliders are similarly colour coded.

To sum up, this feature permits users to define their own notions of similarity (or *fuzzy searching* criteria) interactively, by expressing the relative importance of individual attributes (through the use of scale factors) and tolerated degrees of *fuzziness* (through the use of the exponent values). This feature is missing in existing DQ systems and was identified as a desirable extension to future systems [2]. Although our initial implementation of the user-defined similarity dialog box has limited expressiveness, a much broader class of similarity attributes can be expressed through explicit view definitions, as discussed in Chapter 4.

In order to fully appreciate the power of similarity-extended DQ, consider one last example given in Figure 5.20 on page 84 in which the notion of similarity is that of belonging to the same state as any selected focal point; in the example, this (binary-valued) similarity attribute is mapped onto the node colour saturation property that visually emphasizes cities that belong to the same state as one of the selected focal points. Thus in this example, the task of determining which cities are similar to the selected cities is the perceptual task of identifying saturated points of colour.

Consider how such a task might have been accomplished without similarity attributes. One strategy to accomplish the above is to mask all cities that were found in a state other than a focal point; to do this the user would have to perform the following cognitive task:

1. brush the node of interest and determine in which state it was located by linking (by reading the value shown in the State histogram);

2. mask all histogram bars in the State histogram; and,

3. unmask the histogram bar corresponding to the state determined in step 1.

There are at least three significant problems with this approach:

1. it is *labour-intensive:* the user would have to repeat the aforementioned procedure for every node of interest;

2. it *sacrifices context:* masking all dissimilar nodes may not be desirable since it sacrifices context;

3. it is *error-prone:* there is a possibility that the user could obtain the wrong state number in step 1 above or unmask the wrong histogram bar in step 2.

Steps 2 and 3 above also imply that the State histogram would have to have more than 50 individual bars, which would make it difficult to manipulate individual histograms bars accurately. To be fair, the histogram is a poor way to represent nominal data; however, even if we were to use scrolling lists or a better alphanumeric selection widget such as the the AlphaSlider [1], DQ would still be inadequate at solving this task, as the user would still have to perform the above three actions repeatedly albeit with fewer errors. This is because both DQ and DM are attribute-centered rather than object-centered.

However, by introducing the notion of focal points and similarity attributes into a DQM system, a user can perform the above task by a single click on the node of interest. Consequently, this added functionality has the effect of converting the previously described cognitive task into a perceptual task – one of detecting in the visualization a smaller group of nodes whose colour is highly saturated among a larger group of nodes whose colour is less saturated. Furthermore, notice that we chose a very simple notion of similarity – that of equality in a single attribute; consider the cognitive challenge involved in a more complex task, one in which similarity is a function of multiple attributes, or worse yet, tasks in which similarity is a function of non-trivial graph distances in large graphs. In these scenarios, the cognitive task is not only challenging, often it may be practically impossible.

2.8 Shortest Paths

Another useful feature in Graphite is the ability to compute all pairs shortest paths by summing weights due to both the arcs and the vertices along a path; weights are specified in a Dye view definition and can be constants, or numeric expressions specified in terms of vertex and (adjacency or inclusion) arc attributes.

Shortest path information can then be used in two ways.

First, shortest path distances can appear in the definitions of similarity attributes which allows notions of similarity to be expressed in terms of path distances in a graph, rather than simply expressions involving data attributes as is the case in flat databases. For example, the *TripDuration* similarity attribute which we encountered in Section 2.6, was defined in terms of shortest paths obtained by summing the duration attribute of arcs along each path.

Second, in addition to focal points, Graphite also maintains a set of path sources and sinks that the user can select and deselect by pointing and clicking. The Dye view definition language provides a construct that can be used to maintain a second type of dynamic attribute: one that keeps track of how many shortest paths between the current set of sources and the current set of sinks each vertex or arc takes part in. Since sources and sinks can be selected interactively, attributes defined in terms of this construct are also dynamic in the same sense as similarity attributes – each time a source or sink is selected or deselected, the corresponding attribute values are recomputed, potentially causing some DM bindings to be recomputed, some nodes and links to be masked and others to be displayed.

In Figure 5.1 on page 66 Honolulu is a path source and Reykjavik and Bergen are path sinks, and the *TripFlights* dynamic arc attribute is a 3-bit bitmask: bit 1 is set if the arc is a part of at least one most-direct route, bit 2 is set if it is a part of at least one cheapest route, and bit 3 is set if the arc is a part of at least one quickest route; in the given example, this attribute has been mapped onto link width. Figure 5.4 on page 69 shows how the *TripCities* and *TripFlights* attributes are used in dynamic queries to mask all cities and flights that do not appear on any quickest route originating in Honolulu and arriving in Reykjavik or Bergen.

2.9 Automatic Layout Facility

Graphite permits users to interactively select and apply a number of configurable layout algorithms as shown in Figure 2.7; individual layout algorithms are described in Section A.3. Graph layout is a challenging research area on its own – an extensive overview of the field can be found in an ever-evolving annotated bibliography [22].

The combination of masking with dynamic queries and parametric layout algorithms provides a form of composite layout capability. In addition, the Graphite view definition language, Dye, described in Chapter 4, may be just the missing component in Henry's *Interactive Graph Layout* approach [49, 50], in which selected subgraphs could be drawn independently and the results combined to form fairly intricate composite layouts. The limitation of Henry's approach as it was presented, was the lack of automatic subgraph selection methods – subgraphs were primarily selected interactively. The Dye view definition facility can be used to select subgraphs and to automatically assign layout algorithms to various containers in the graph container hierarchy (see Section A.3).

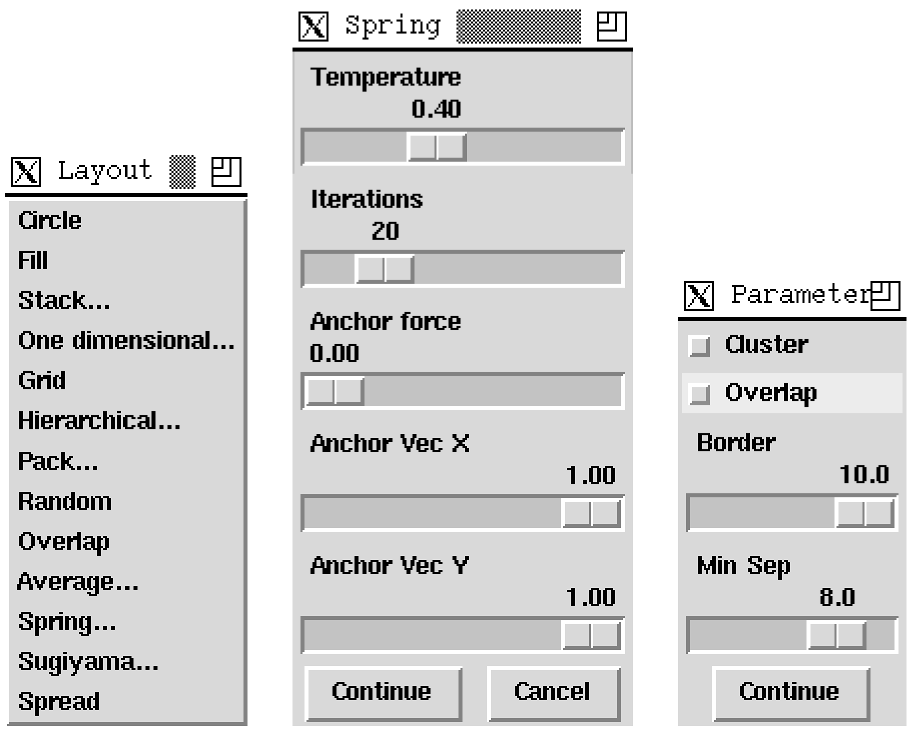


Figure 2.7: Selecting graph layout algorithms. When a layout algorithm is selected from the menu shown on the left, the user is presented with a dialog box that is used to configure various algorithm-specific parameters, as shown in the centre; pressing *Continue* causes the current graph to be redrawn and redisplayed. A set of generic graph layout parameters can be manipulated through an additional dialog box shown on the right.

Chapter 3

Related Work

3.1 Dynamic Queries

3.1.1 Definition

The dynamic query approach [105] applies the principles of direct manipulation [102] to database searching:

- visual presentation of the query's components (*e.g.,* as sliders, radio buttons, check boxes, and frequency distribution histograms);
- visual presentation of results (*e.g.,* as two dimensional scatter plots);
- rapid (100ms), incremental (outputs are inputs), and reversible control of the query;
- selection by pointing, not typing; and,
- immediate and continuous feedback.

This approach stands in stark contrast to traditional database query approaches that typically employ form-based interfaces or textual query languages like SQL [19], and in which output is formatted as a tabular list of tuples containing alphanumeric fields. The DQ approach offers many advantages for both novice and expert users, including reduced learning curves, fewer errors, increased levels of understanding, and, not to be ignored, user satisfaction. In particular, visual presentation of results can exploit the power of the human perceptual system in identifying patterns and exceptions, and discovering relationships in data, as in, for example, the types of tasks involved in air traffic control or in network management.

In general, each query component (slider, button, histogram) acts as a filter, controlling the number of visible data points. For example, dragging a slider can cause points to be added to the display (if enlarging the selected range), or it may cause points to be removed from the display (if narrowing the selected range). For this reason, such query components have also been called *dynamic query filters* [2]. The filtering effects of the components are typically combined using AND logic (set intersection).

Data Type	Component
Numeric	single or double-ended sliders [2, 3, 38, 84, 121]
	frequency distribution histograms [26, 27]
Alphanumeric	scrolling lists [38]
	Alpha slider [1]
	regular expression matching [84]
Nominal/Categorical	checkboxes, buttons [3, 38]
	scrolling lists [38]
Aggregate	Aggregate Manipulator [38]

Table 3.1: Some common dynamic query components.

A number of DQ component types have appeared in existing DQ systems and several common examples are summarized in Table 3.1. Note that some components, such as frequency distribution histograms can be used to both highlight and mask data points. For this reason, masking is often called *dynamic filtering* or *interactive filtering*. Also, note that a dynamic query which combines the effects of multiple components can be thought of as an n-ary brush [7]: if each component is a single brush, an n-ary brush is typically obtained by intersecting the individual brushes.

Formally, the result R' of a dynamic query on relation R is:

$$R' \quad \leftarrow \quad \bigcap R_i$$

where R_i is the subset of relation R selected by query component i; since $R_i \leftarrow \sigma_{F_i} R$

$$\begin{aligned} R' \quad &\leftarrow \quad (\sigma_{F_1} R) \cap (\sigma_{F_2} R) \cap \ldots \cap (\sigma_{F_n} R) \\ &\leftarrow \quad \sigma_{(F_1 \wedge F_2 \wedge \ldots \wedge F_n)} R \end{aligned}$$

in relational algebra [15, 114] notation, or:

```
SELECT R.* FROM R WHERE Condition1 AND Condition2 AND ... AND ConditionN
```

in SQL notation [19]. The exact form of each F_i depends on the type of DQ component associated with attribute i. For double-ended sliders, for example, F_i is of the form:

$$R.A_i \geq SliderMin \wedge R.A_i \leq SliderMax$$

while for frequency distribution histograms with n bars, F_i is of the form:

$$(R.A_i \geq Min_1 \wedge R.A_i \leq Max_1 \wedge On_1) \vee \ldots \vee (R.A_i \geq Min_n \wedge R.A_i \leq Max_n \wedge On_n)$$

where Min_i and Max_i are the extrema of each histogram bar, and On_i is true if histogram bar i is on (not masked). However, since each value of attribute i is contained in the range of exactly one histogram bar, a more direct way to specify the same condition is to simply check if this bar is turned on:

$$On_{((R.A_i - R.A_{min})/(R.A_{max} - R.Amin)\cdot(n-1)+1)}.$$

In what follows we review a number of existing systems that employ DQ, and summarize our findings.

3.1.2 Non-graph-oriented Systems

Homefinder

The Dynamic Homefinder was one of the earliest applications to use the dynamic query approach [121]. This system allowed novice users to explore a real-estate database that contained a number of attributes: number of bedrooms, price, type (house, apartment, condominium), features (fireplace, garage, central air conditioning, whether the property is new or resale), etc. Users used sliders to select ranges over quantitative attributes, and checkboxes to select nominal attributes. Homes were displayed as points of light on a 2-D scatterplot, the geographic position of the home being mapped onto the x and y axes. In our opinion, a somewhat understated feature of the system was the provision of two graphical marks on the display that could be interactively positioned by the user; the system computed the distance from any data point to each of the markers and made these quantities available for interactive filtering with two sliders.

Periodic Table of Elements

This educational program allowed users to interact with a periodic table of elements using dynamic queries [3]. The periodic table of elements data included 6 attributes: atomic number, atomic mass, electronegativity, atomic and ionic radius, and ionization energy. The dynamic query system allowed users to interactively highlight elements that satisfied the query which was specified through a set of sliders. The authors carried out user experiments to compare this interface with two alternatives: a graphical interface with form fill-in instead of sliders, and a textual interface with form fill-in. Performance results were measured in terms of the time required to answer questions and the number of correct answers. Results showed that the dynamic query interface was as good or superior in all five tasks.

Health Statistics

This application allows statisticians to study cancer rates throughout the United States [85]. The database contains annual statistics for each state sampled between 1950 and 1970; each data point consists of several attributes including cancer rate, per capita income, college education, and smoking habits. The user is presented with a geographic map containing state borders, and can obtain individual statistics by pointing and clicking. In addition to masking, as users manipulate sliders corresponding to time or demographic variables, rapid colour changes help to identify trends.

Filmfinder

The Filmfinder incorporates dynamic queries with *Starfield Displays* (scatterplots with pan-and-zoom capability) as an aid to users in searching a large film database [2]. The inclusion of starfield displays prompted more careful treatment of tight coupling and display invariants (*i.e.,* due to pan-and-zoom) in the context of *visual information seeking*.

Also noteworthy, is the fact that the authors made the observation that some form of *fuzzy searching* capability was desirable in the Filmfinder scenario, but they did not elaborate on the nature or implementation of such a mechanism. As we show, using database views to derive similarity attributes captures a broad class of such fuzzy queries.

Aggregate Manipulator

The aggregate manipulator is a user interface mechanism that allows users to create and decompose aggregates (groupings of data) and was designed to complement the dynamic query technique [38]. Although not presented as such, in essence the aggregate manipulator permits users to define limited *views* on the underlying data. Although the expressive power of the view definition mechanism was not formally defined, the aggregate manipulator can be used to create the following types of derived attributes:

1. *User-defined* or *pre-defined groupings:* selected interactively by the user, or *a priori*, such as grouping years into quarters, and quarters into months.

2. *Element frequency divisions:* where each division has the same number of elements (*e.g.,* 20 divisions on a distribution that contains 1000 points would result in 50 points per division).

3. *Set interval divisions:* where each division has the same interval length (*e.g.,* 20 divisions on a distribution of 1000 points would result in each division having the interval $[20 \cdot i, 20 \cdot (i + 1)]$ for $i = 0, 1, 2, \ldots, 49$).

4. System-supplied statistical methods: which for quantitative data types include the familiar aggregate operations of sum, average, and count.

The first method relies on user-selection and therefore is not a candidate for view definition.

The second method, corresponds to computing statistical quantiles (percentiles) [58], and can be potentially difficult to realize in a database setting; first, it requires that the distribution be sorted on the attribute in question (which is relatively common), and second, it requires the ability to obtain the position of each point in the sorted view (which is relatively rare). Furthermore, it can be potentially expensive to maintain a sorted view if graph updates are permitted.

The third and fourth methods can be easily realized through the use of range queries and aggregate queries respectively.

To sum up, the aggregate manipulator permits the definition of a view on a single relation involving simple range queries over that same relation or aggregate queries over subsets of that same relation.

The authors claim that DQ with the aggregate manipulator is better than DQ alone; what they are really arguing, albeit in an overly narrow way, is that database views are helpful in that they permit the creation of abstract views of the underlying database. The contribution of this work thus is two-fold: to show that views can enhance dynamic query systems, and to propose a user interface technique that permits users to define a limited class of views.

3.1.3 Graph-oriented Systems

Dynamic Graphics for Network Visualization

This is one of the earliest papers that discussed the use of direct manipulation techniques including dynamic mapping and filtering for visualizing network (graph) data [8]. The dynamic mapping strategies mentioned include: symbol type, size (both uniform and non-uniform), and colour for nodes, and thickness, direction (wedge-shaped line segments), proportional length (don't draw line segments all the way to each node), and colour for links. The authors make the important observation that relative node sizes and link thicknesses generally cannot be assigned *a priori*, but must be configured at run time through a set of dynamic mapping widgets – typically by a slider to control relative sizes or thicknesses. In addition, the paper describes how brushing statistics, in the form of colour-coded "sliders", can be used to link statistics to nodes or links and to mask nodes or links by dragging the lower or upper thresholds of each slider.

Perhaps an even more significant observation, is that of the relationship of the nodes to the links: namely that node attributes can be aggregate expressions involving the attributes of incident links, and furthermore, that paths through the network can be important. However, the paper does not discuss how such attributes or paths might be maintained by a highly interactive dynamic query system. Instead, the authors apparently assume that graph data is static and all attributes are precomputed.

Browsing Unix Directories

Three alternative user interfaces to support dynamic queries and (text-based) visualization were compared in user experiments [66]. ASTER displayed every directory entry on the screen and placed an asterisk (*) in front of each line that matched the selection criteria; COLOR was similar, but used different colours (blue for selected, yellow for deselected) to identify entries selected by the query; and finally, EXPAND displayed only matching entries.

User studies were carried out with eighteen expert participants performing 10 tasks. Results varied by task, but it is very interesting to note that the one task in which all three interfaces were all (equally) least successful was answering the question: *How many files are there that are younger in age than "umcp_tai"?* This is very interesting for two reasons. First, it was the only task that involved the comparison of entries to a selected entry; and second, it was the task that took the longest to complete with all three user interfaces.

As we have already stated, it is precisely this type of similarity query (find points that are similar to the selected point(s)) that our proposed approach makes possible in a dynamic query setting.

HierNet

HierNet is a tool for visualizing and analyzing large hierarchical networks [27], and has been used to understand electronic mail interaction patterns (who talks with whom, who sends how much email), and module subsystem hierarchies arising in software engineering. The system can evaluate dynamic queries with the use of frequency distribution histograms, and in addition, permits users to interactively smooth the histograms using a *kernel density smooth*. HierNet also supports dynamic mapping of data attributes to presentation properties such as node and link colour, node size and shape, and link thickness. Smoothing histograms thus has an immediate effect on the visualization by causing mapped presentation properties to be updated. Another interactive technique takes advantage of the hierarchical nature of the data by allowing users to interactively expand and collapse subtrees.

Seenet and NicheWorks

Recent work on data visualization at Bell Labs [26] included two systems for visualizing graphs: Seenet, a system for displaying geographic networks – graphs in which node placement is derived from domain-data (*i.e.,* geographic position); and, NicheWorks, a system for visualizing abstract networks – graphs that arise from applications in which there is no predetermined natural way to position nodes. Both systems permit users to filter data by interacting with frequency distribution histograms linked to data attributes, and to dynamically map data attributes onto graphic properties such as node and link colour, node size (width and height could be controlled independently), and link thickness. As in [8] node attributes could be aggregations over incident links, but here too, such attributes apparently are precomputed.

Webviz

Webviz is a tool for visualizing and analyzing World-Wide Web access logs [84]. Webviz provides very rudimentary query capability, namely the capability to mask nodes by regular expression matching. Strictly speaking, this form of query is not direct-manipulation (and therefore not DQ), as it involves a relatively complex user task – filling in a textual regular expression. Webviz also supports a limited form of dynamic mapping: the frequency and recency of document accesses can be mapped onto node border width, node border colour, link colour, and link thickness.

Visual Information Management for Network Configuration

This prototype network configuration management system employs colour-coded treemaps [56, 112] combined with dynamic queries that permit user-controlled filtering and animation of network configuration data [65]. The size and colour of treemaps was controlled by the user via dynamic mapping, and treemap slices could be grayed-out through dynamic queries. A conventional node-link tree browser was also used to show hardware containment hierarchies. The nodes in the tree browser could be colour-coded to represent configuration status and work is in progress to allow both attributes-based and topology-based dynamic querying. Note that our Graphite system already supports both of the above styles.

Selective Dynamic Manipulation

Selective dynamic manipulation (SDM) is a set of interactive techniques for 2-D and 3-D visualizations [11, 12]. *Selective* is meant to convey user control in selecting sets of objects, *dynamic* indicates real-time interactivity, and *manipulation* suggests that users can move objects and transform their appearance. Selection in SDM can be performed in two ways: through dynamic queries involving slider dynamic query components, and by clicking on individual objects. The novel feature of this approach is the relatively fine-grained ability to manipulate subsets of selected graphic objects. This has both advantages and disadvantages: SDM allows selected objects to be transformed independently of the other objects, but it requires users to describe transformations manually; dynamic mapping, on the other hand, is generally used to apply a transformation to all objects simultaneously, but does so automatically.

Note that in systems like HierNet [27] described above, dynamic mapping is user configurable to the extent that users can control the degree of transformation when mapping to certain graphical properties such as node size (by using a slider), and the shape of the transformation through histogram smoothing.

PDQ Tree Browser

The *Pruning with Dynamic Queries* (PDQ) tree browser [64] presents trees in two tightly-coupled views: a detailed view and an overview. The tool allows users to filter nodes at each level of the tree, and prunes subtrees of unselected nodes.

The authors make an important distinction between “flat” data sets consisting of independent data points (in which there are no relationships between data points), and “non-flat” data sets in which data points are related, as in hierarchical data sets.

System	*Year*	*Mapping*			*Views*			*Inputs*		
		None	*Static*	*Dynamic*	*None*	*Predefined*	*User-defined*	*Flat*	*Hierarchical*	*Graph*
Network Visualization [8]	90			✓		✓				✓
Homefinder [121]	92			✓		✓		✓		
Periodic Table of Elements [3]	92			✓	✓			✓		
ASTER/COLOR/EXPAND [66]	92		✓		✓				✓	
Health Statistics [85]	93		✓		✓			✓		
HierNet [27]	93			✓		✓				✓
Seenet, NicheWorks [26]	93			✓		✓				✓
Filter/flow proposal [104, 105]	93	✓					✓	✓		
Filmfinder [2]	94			✓	✓			✓		
Aggregate Manipulator [38]	94		✓				✓	✓		
Webviz [84]	94			✓	✓					✓
Network Configuration [65]	94			✓		✓			✓	
SDM [11, 12]	95			✓	✓					✓
PDQ Tree Browser [64]	95			✓		✓			✓	

Table 3.2: Summary of existing Dynamic Query system features.

3.1.4 Limitations

As evidenced by Table 3.2, the main limitation found in existing DQ systems is the lack of integrated flexible view definition facilities. Several instances of predefined views have appeared in the literature [8, 64, 65]. Generally, it is assumed that attribute values in a predefined view are precomputed – an assumption that tends to simplify the task (and hence the architecture) of the visualization system. In particular, the visualization system does not have to evaluate or maintain views. However, we argue that precomputed views are too restrictive.

In general, just as it is desirable to have the facility to define abstract views in traditional database settings, so it can be equally desirable to incorporate view definition facilities in DQ systems, for this provides a mechanism by which user-defined attributes can be computed and maintained.

More specifically, an integrated view maintenance facility is a must in order to maintain derived attributes whose values change in response to user actions (*i.e.,* dynamic attributes), as exemplified by our proposed use of similarity attributes.

The Aggregate Manipulator [38] contains a relatively limited view definition facility, and the proposed filter/flow boolean query interface [104, 105] would similarly require a relatively simple view component; in particular, both approaches are limited to flat databases and are too weak to express similarity queries.

3.2 Fisheye Views

3.2.1 Definition

The generalized FEV formalism [34] is based on the workings of the fisheye or very wide angle lens used in photography, which magnifies the image at multiple levels – greatest near the focus and least in the periphery. By balancing local detail and global context, FEVs can simultaneously display information at multiple levels of abstraction.

When displaying large structures, the basic strategy uses a *degree of interest* (DOI) function to assign to each point in the structure a number, or *priority*, that quantifies the user's interest in that point given the current task. Priorities may be assigned by the user, or may be computed by an algorithm. The DOI function is typically decomposed into two components: *a priori importance* (API) which computes the global importance of any point in the structure, and *distance* (Dist) which computes the conceptual distance between any two points. If one point is selected as the current focus of interest, or *focal point* (FP), then in its simplest additive form, $DOI(p, f) = API(p) - Dist(p, f)$ is the user's degree of interest in point p given FP f; thus DOI increases with API and decreases with distance.

3.2.2 A Simple Taxonomy of Graph Views

There are many different variants of fisheye views – a fact that unfortunately has lead to much confusion. The following simple taxonomy is an attempt to shed some light on the matter.

A *normal view* is a layout in which all elements have identical priorities, or, equivalently, a view with no FPs. An *implicit FEV* arises from the effect of point perspective in 3-D, by which nearby points loom large and distant points appear small; implicit FEVs are usually static (*e.g.,* [30, 63, 89]), but may be dynamic if the 3-D layout is permitted to be altered (*e.g.,* by moving higher priority nodes to the foreground [88]). There are at least three types of *emphasized views* (EVs). A *filtered view* displays a subset of elements and suppresses the rest – this can be accomplished by zooming or by filtering according to elements' priorities (display elements with priorities above some threshold, suppress others). A *distorted view* emphasizes elements by distorting their sizes, shapes, and positions – either by a polar or orthogonal distorting FEV transformation (Dist is geometric), by a non-geometric FEV distortion (Dist is non-geometric), or by scaling at single (local) or multiple (global) levels. An *adorned view* emphasizes elements by varying other visual presentation variables such as colour, shading, line style and thickness, as well as audio [57], and motion (*e.g.,* "in-betweening" animation [88], vertical oscillations and small random movements [30], vibration or pulsing [100]). Figure 3.1 shows a normal view of this taxonomy, while Figures 3.2, 3.3, and 3.4 show examples of other views of the same taxonomy. Note that many algorithms described in the next Section generate hybrid views that exhibit the properties of two or more of these basic views.

For brevity, throughout the following review Section we refer to an algorithm that generates an emphasized view as EA (emphasis algorithm), and an algorithm that computes priorities as PA (priority algorithm). The review summarizes observations that we reported in [81].

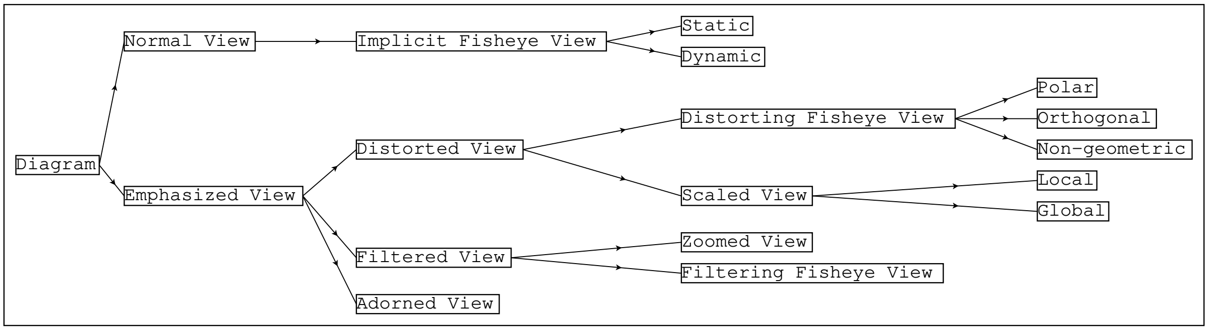


Figure 3.1: A simple taxonomy of graph views; note that existing categories could be further refined and more categories could be added.

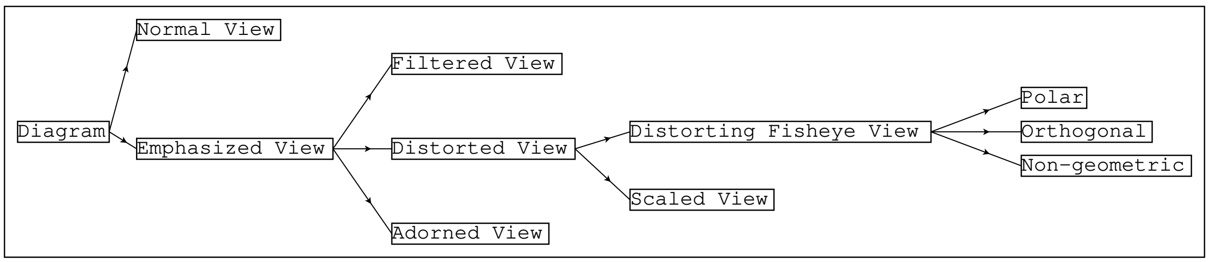


Figure 3.2: A first order filtering FEV with FP: *Distorting Fisheye View*.

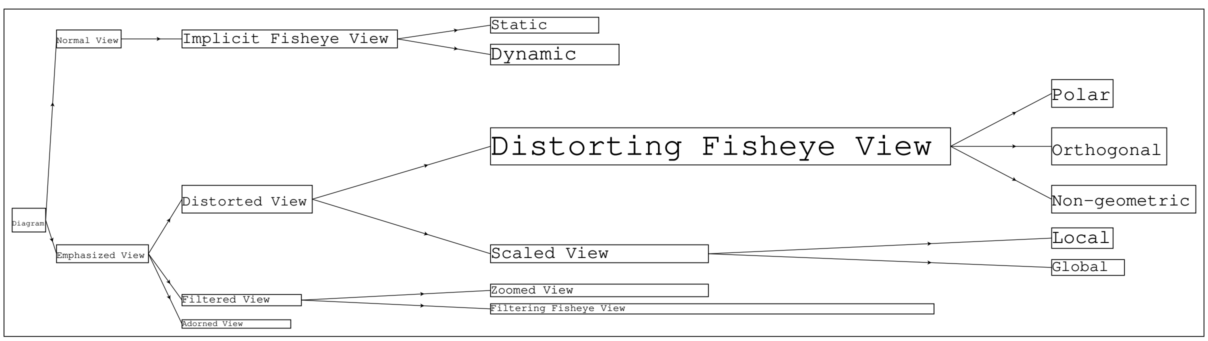


Figure 3.3: An orthogonal distorting FEV with FP: *Distorting Fisheye View*.

3.2.3 Non-graph-oriented Techniques

Generalized Fisheye Views

This formalism for reducing the complexity of highly detailed visualizations by displaying subsets of the most relevant details [34], generates filtering FEVs. As mentioned earlier, in a generalized FEV, DOI(p) = API(p) - Dist(p, f), given FP f.

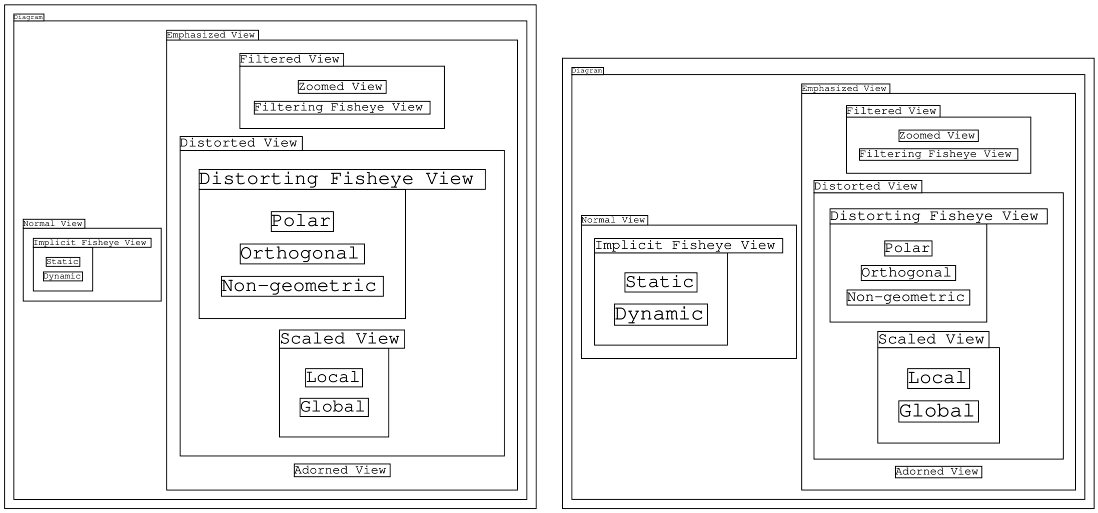


Figure 3.4: Non-geometric (layout-independent) distorting FEVs; left – FP: *Distorting Fisheye View*, and right – FPs: *Dynamic* and *Global*.

Aircraft Maintenance Diagrams

This EA, an extension of the generalized FEV strategy for presenting aircraft maintenance data [76], generates filtering FEVs. The overall goal of this work was to enhance the usefulness and effectiveness of computer-based maintenance and diagnostic tools for aircraft repair technicians. Although the visualizations in this domain were circuit schematics and mechanical parts diagrams, the underlying data model was a directed network in which vertices represented components and arcs represented physical and functional relationships between the components. The generalized FEV formalism was extended to capture multiple FPs in a directed network by rewriting Dist as the sum of shortest path distances from a vertex to the FPs. The resulting vertex priorities enabled the construction of nth order FEVs, similar to the generalized FEVs of trees [34].

The Perspective Wall

This EA for visualizing linear information [68] such as time, generates globally scaled views. It uses graphics hardware support to fold wide 2-D charts into 3-D visualizations consisting of a center panel for showing detail and two perspective panels for displaying context. The perspective view provides efficient space utilization for 2-D charts with wide aspect ratios, and generates a FEV effect: it emphasizes the neighbourhood of the detailed view by making it larger than the more distant parts of the contextual view. The trade-off between detail and context is controlled by manipulating the degree of folding, the width of the detail panel, and the angle of the field of view. The single FP is changed by scrolling the wall as one would a sheet in a player piano.

Treemaps

This EA for presenting hierarchical information [56, 57, 103, 112], generates hybrid locally scaled and adorned views. A hierarchy is drawn as a set of nested boxes in which each node is depicted as a rectangular region composed of the rectangular regions that represent its children. The main advantage of the method is its ability to visualize large hierarchies, which it achieves through its linear top-down space-filling layout algorithm. Treemaps can emphasize nodes in the hierarchy by a two step process. First, each node is assigned a weight based on the weights of its children in the hierarchy (the leaves are typically assigned a weight corresponding to some quantitative attribute). Next, the layout algorithm ensures that the total area in the drawing allocated for each node is proportional to its weight. In this manner, more important nodes are emphasized by being drawn larger than nodes of lesser importance.

Notice that while Treemaps provides the mechanism to generate visualizations that incorporate emphasis, it does not solve the problem of assigning meaningful weights to nodes given a set of FPs.

StretchTools

This EA [98] used to manipulate 2-D screen space, generates locally scaled views. The user interface is based on the metaphor of stretching a rubber sheet by using *handles* and *clamps*. A handle, when placed on the screen and pulled causes the screen (and the graphical objects displayed on the screen) to expand on one side of the handle and contract on the other. Clamps impose constraints on the movement of handles by joining handles with one another.

Although the EA demonstrates the potential effectiveness of the rubber sheet metaphor, the approach has several undesirable properties. First, the technique does not provide a way to enlarge a single region without affecting an entire row or column of regions. Second, because local scaling is used, the regions near a FP are not enlarged automatically, but remain as small as remote regions; to generate a FEV effect, more handles must be explicitly added to scale diagrams at multiple levels. Third, orthogonal stretching causes large discontinuities at the region boundaries failing to smoothly integrate individual regions into a coherent layout.

On the positive side, since the EA can be used to stretch a screen that contains arbitrary graphical objects, it follows that the approach can be used to manipulate drawings of graphs – both flat and nested.[1]

Morphing

This EA [99] is based on image transformation technology and generates globally scaled views. It was proposed to overcome the difficulties associated with StretchTools [98], and allows users to specify polygonal FP regions; global scaling is used to enlarge the interior of the selected regions while simultaneously adjusting the remainder of the layout so that the magnified areas are smoothly integrated with the demagnified ones.

[1] In fact, the authors show an example of stretching screens generated by the prototype graph browser which was used to display graphical FEVs [96, 97].

3.2.4 Graph-oriented Techniques

Generalized FEVs of Trees

This EA for simplifying the display of trees by suppressing less relevant nodes [34], generates filtering FEVs. A generalized FEV of a tree is obtained by providing specific instantiations of the API and Dist functions. Specifically, Dist is the path distance between two vertices, API is the distance from the root of tree, and DOI(v) = - (Dist(v, f) + Dist($v, root$)). This treats the internal nodes as intrinsically more important than the leaves of the tree. An nth order FEV is obtained by only displaying points with DOI(v) $\geq -($Dist$(v, root) + 2 \cdot n)$.

SemNet

Three EAs for the 3-D networks of SemNet were proposed: clustering (filtering FEV), 3-D point perspective (static implicit FEV), and sampling density (filtering FEV) [30]. In clustering, recursive subdivision of the 3-D graph volume yielded an 8-ary tree with SemNet nodes partitioned among its leaves; tree distance (an approximation of Euclidean distance) was used in the DOI function. Although not implemented, the third EA was motivated by the workings of the human retina which samples an image densely at its center of focus and in successively less detail for points further away. All three EAs were defined in terms of a single FP.

Topographic Networks

This EA for displaying topographic networks [51, 52], generates hybrid filtering fisheye and non-geometric distorted fisheye views. The FEV was formulated in terms of a single FP; the DOI function was used to both, determine the amount of detail to be displayed (*e.g.,* omit node labels), and to distort the positions and sizes of nodes.

Cone Trees

Cone trees display hierarchical information by using 3-D graphics and interactive animation [89]. By limiting the domain of inputs to hierarchies, Cone Tree visualizations were not exposed to many of the problems that complicated the design of SemNet [30] and increased cognitive load. Cone tree visualizations support two types of FEVs: distortion from 3-D point perspective (static implicit FEV), and filtering through interactive *gardening* operations (filtered view). Additionally, the Cone Tree search facility automatically rotates cones to bring the target node to the front of the display, thereby relieving the user from an otherwise difficult manual searching task. Gardening consists of three types of operations: in *pruning*, the descendants of a selected node are hidden; *growing* restores hidden nodes; the *prune others* operation prunes the siblings of the selected node leaving only the selected substructure visible. The last operation, in particular, allows a user to interactively select FPs, and can be used in conjunction with the search facility to simplify a large, complex hierarchy. Ironically, the highly interactive nature of the technique makes it less appropriate for exploring very large hierarchies – a more automatic solution would be preferred (*i.e.,* it is unrealistic to expect users to work hard to simplify visualizations of large hierarchies).

Compound Digraph Display Methods

Three multi-viewpoint perspective (MVP) EAs [75, 108] have been applied to drawings of compound digraphs [109]: the fisheye EA uses a polar transformation to map each point in the original drawing to the perimeter of a circle (polar distorting FEV); the orthogonal fisheye EA uses independent Cartesian transformations to map each point to the perimeter of a square (orthogonal distorting FEV) meaning that lines that were parallel to the x or y axis in the normal view remain parallel to these axes in the distorted view; in the biform EA, FP areas are magnified uniformly and others are de-magnified uniformly (locally scaled view).

A related publication [24] describes three properties that should be preserved when a drawing is transformed:

1. *Orthogonal ordering:* preserve the vertical and horizontal ordering of points (*e.g.,* if p is north-east of q in the normal view then p' must be north-east of q' in the distorted view).

2. *Topology:* the inside of a closed continuous curve must be mapped to the inside of a closed continuous curve (*i.e.,* if there is no overlap in the normal view, there must be no overlap in the distorted view).

3. *Clusters:* objects that are proximate in the normal view must also be proximate in the distorted view.

Document Associative Networks

This EA for simplifying the task of information retrieval [32], generates hybrid filtering fisheye, globally scaled, and adorned views. The approach is centered around a common visually displayed network structure: an associative network. The user interface visualizes networks using the generalized FEV formalism in which a node's API is its degree (number of incident arcs) and Dist is the sum of arc weights along a shortest path. A node is displayed only if its DOI exceeds a threshold. Inexplicably, however, only the API rather than the DOI is used to determine the size of each displayed node. In addition, shading is also used to further emphasize interesting (larger) nodes. The FP (node) is drawn as large as the largest node in the normal view.

The shape of links is also used to convey information about the structure of the network: links are widest at the FP and narrow as they connect nodes that are more distant from the FP.

Interactive Graph Layout

This approach comprises interactive or algorithmic subgraph selection, layout algorithm assignment, and sublayout composition [50], and generates hybrid locally scaled and adorned views. The nodes in a selected subgraph can be enlarged to emphasize the subgraph. For example, to emphasize a path, the nodes on the path can be positioned by a row or column layout algorithm, their bounding boxes can be magnified, and an alternative font (type and size) can be used to render their textual labels. Since interactive layout algorithm assignment is required to specify multiple local scaling distortions, it is unclear whether the technique is appropriate for generating EVs of large graphs.

Graphical FEVs of Graphs

This technique [96, 97] generates hybrid filtering fisheye and distorting fisheye (polar and orthogonal) views. "Graphs" implies that the technique can be applied to non-hierarchical graphs, while "graphical" implies that the technique furnishes a graphical interpretation of FEVs which integrates layout considerations into the fisheye formalism: the position, size, and level of detail of displayed nodes are computed based on client PAs.

A graphical FEV is obtained by magnifying the nodes of greater interest and correspondingly demagnifying nodes of lesser importance, and recomputing the positions of all nodes and link bend points. Although the client PAs can be modified to ignore the geometry of the normal view, the technique has been designed to function primarily as a distortion of existing layouts. For example, the position of a node in the FEV is a function of its position and the position of the FP in the normal view (*i.e.,* Dist is the Euclidean distance between nodes in the normal view).

Although graphical FEVs can be implemented efficiently using dedicated graphics hardware, the technique is limited to non-nested graphs with a single FP.

Abridgment

This EA, used in the D-ABDUCTOR system [74] for viewing and manipulating compound digraphs [109], generates hybrid filtering fisheye and non-geometric distorting fisheye views. Priorities are computed as a linear combination of three quantities: structural importance (vertex nesting depth); semantic importance (API); and focal importance (proximity to the set of FPs). Priorities can be used in two ways: in a hybrid drawing, nodes with priorities less than a lower threshold are suppressed, those between the lower and upper thresholds are reduced, while those with priorities greater than the upper threshold retain their original sizes; in a proportional drawing, nodes' sizes are directly proportional to their priorities. Animation can be used to reduce abrupt changes caused by a change in the set of FPs.

Variable Zoom

This EA [100] generates hybrid filtering fisheye and globally scaled views of hierarchically nested graphs. The EA is similar to the Biform Display Method [75], and is limited to drawings in which the projections of nodes on the x and y axes do not overlap. The contents of each node are recursively rescaled in a top-down fashion. Within each node, selected nodes are magnified while the rest are demagnified; the EA computes scale factors that preserve a node's size while resizing and repositioning its contained nodes. A balance factor is used to control the ratio of detail to context at each level in the hierarchy, meaning that all zoomed nodes at a given level are magnified equally. Variable Zoom, like StretchTools [98], was designed for interactive rather than automatic operation – the user interacts to selectively magnify and demagnify portions of the display. Unlike StretchTools, however, which permits arbitrary distortions of the screen, the distortions obtained by this EA are more restrictive due to its use of the balance factor as described above. Furthermore, without explicit notions of DOI and decomposition into API and Dist, these and other techniques [50, 68, 75, 89] are less appropriate for automatic generation of EVs.

Fractal Views of Trees

This EA exploits fractal self-similarity to aid the visualization of large trees in 3-D [63], and generates hybrid filtered, static implicit fisheye, and globally scaled views. A node's priority is its *fractal value* which has the following general form:

$$\begin{aligned} FV(focus) &= 1 \\ FV(child(x)) &= (C{N_x}^{-\frac{1}{D}}) \times FV(x) \end{aligned}$$

where N_x is the branching factor of node x, D is a fractal dimension, and $0 < C \leq 1$.

The fractal filtering mechanism has one nice property that the generalized FEV of trees [34] does not: it guarantees that as the FP changes, the number of nodes whose fractal value exceeds a specified threshold will remain nearly constant regardless of branching factor. Thus if only nodes whose fractal values exceed the threshold are displayed, both the visual complexity of the resulting layouts and the corresponding system response times will remain relatively constant as the FP is changed. In addition to this *fractal pruning*, each visible node is resized so that its size is proportional to its fractal value.

PLUM

This system supports the visualization of abstract data in 3-D [88], and generates hybrid implicit fisheye (static and dynamic) and adorned views. Although it can generate a variety of static and dynamic visualizations, PLUM has been aimed primarily at displaying information about software structures and is best at visualizing graphs, as it provides a variety of 3-D graph layout algorithms. PLUM supports visual emphasis in two ways.

First, a client application can control a common set of graphical object properties including stylistic properties, sizing information, and a priority setting. Stylistic properties such as colour, font, fill and line styles can be controlled explicitly, as can the sizes of objects. The priority setting provides a general means for specifying that an object is important and should be emphasized in the display – different graphical objects may respond to the priority setting in different ways.

Second, PLUM's built-in animation support can be used to reflect changes in the display structure. For example, if an object's priority setting is linked to its z position, then increasing its priority could cause the object to be moved from the back of the 3-D display to the foreground. Animation can be used to make this transition smoother enabling the user to perceive the desired change with less cognitive effort.

Layout-independent FEVs

This EA generates hybrid filtering fisheye, non-geometric distorting fisheye, and adorned views [78, 79]. While most distorting FEV EAs distort normal views, this EA uses built-in and client PAs to guide the drawing of the EV. The EA supports nested graphs with multiple variable magnification strength FPs, and generates EVs by a two step process. First vertex priorities are computed. Next, the graph is drawn by a bottom-up layout algorithm that computes the size and shape of each node by first positioning its contained nodes.[2] An EV is generated by uniformly scaling the bounding box of each node in proportion to its priority – thus each node is resized *before* it is positioned by the layout algorithm. The priority of each arc is the average of the priorities of its endpoints, and is used to vary the thickness of link line segments in the drawing.

Although most EAs compute DOI values by using additive DOI functions, this EA uses the following multiplicative form:

$$DOI(v) \quad = \quad API(v) \cdot (p \cdot Wt(v) + (1 - p) \cdot Prox(v))$$

The functions API, Wt and $Prox$ return non-negative real values: $API(v)$ is v's API, $Wt(v)$ is a generic measure of v's proximity to the FPs, $Prox(v)$ is an application-specific measure of v's proximity to the FPs, and $0 \leq p \leq 1$ reflects the trade-off between generic and domain-specific notions of proximity. The generic measure of proximity is an approximation to a weighted nesting distance to the FPs. Both Wt and $Prox$ take into account the magnification strengths of the individual FPs.

Continuous Zoom

This EA generates hybrid filtering fisheye and globally scaled views [6], which permit users to view and navigate hierarchically nested graphs by smoothly expanding and shrinking nodes. The technique evolved from the Variable Zoom EA [100] and is part of the Intelligent Zoom interface to time-critical systems; unlike the Variable Zoom, however, this EA calculates priorities using four quantities: the node's API, alarm state, connectivity to other important nodes, and perceived user interest (determined by node "open" and "close" actions).

[2] This layout strategy permits the generation of composite layouts [50] by assigning a possibly unique layout algorithm to each vertex.

CATGraphs

This EA generates globally scaled views using the compressed arctangent function [60]:

$$f(x) \quad = \quad arctan\left(\frac{0.5}{r}(x - x_i)\right)$$

given a region of interest of radius r centered about the focal point x_i. This function is applied independently to nodes' center x and y coordinates. Multiple points of interest are handled by transforming each node once for every focal point and then computing the mathematical average of the transformed points. Note that the function maps values into the interval $(-\frac{\pi}{2}, +\frac{\pi}{2})$; in practice the layouts are scaled to fit within the dimensions of a computer screen or printed page.

Since the function is applied independently in x and y, given a focal point at (x_i, y_i) the width of nodes at $x = x_i$ will be the same as the width of the focal point; similarly, the height of nodes at $y = y_i$ will be the same as the height of the focal point. For this reason, when several focal points are selected, this technique creates *ghost* focal points at the intersections of these lines. The authors suggest using a polar transformation to overcome this limitation, however, this approach can lead to severe distortion of node shapes which may render the technique far less effective. The transformed coordinates (x_t, y_t) in the polar transformation are computed as follows:

$$\begin{aligned} d &= \sqrt{(x - x_i)^2 + (y - y_i)^2} \\ \theta &= arctan\left(\frac{y - y_i}{x - x_i}\right) \\ d_t &= arctan\left(\frac{0.5}{r}d\right) \\ x_t &= d_t cos\theta \\ y_t &= d_t sin\theta \end{aligned}$$

3.2.5 Limitations

The key features of existing fisheye view techniques are summarized in Table 3.3 along the following four dimensions:

1. *Emphasis Technique* – types of views used to generate emphasis (see Section 3.2.2): *Implicit* (i). *Filtered* (f). *Geometric Distortion* (g). *Non-geometric Distortion* (ng). *Adorned* (a).
2. *Priorities* – methods to obtain priorities:
   - *Supplied* (s)*:* (static) priorities are supplied by user or client PA.
   - *Built-in API* (A)*:* API is computed by a built-in PA.
   - *Built-in Dist* (D)*:* distances (proximities) are computed by a built-in PA.
3. *Number of Focal Points* – number of FPs explicitly supported by the EA: *None* (0); *Single* (1); *Multiple* $(*)$.

Approach	***Date***	***Emphasis Technique*** $i/f/g/ng/a$	***Priorities*** $s/a/d$	***# of FPs*** 0/1/$*$	***Inputs*** $s/h/g/n/*$
Non-graph-oriented:					
Generalized FEVs [34]	04/86	□■□□□	□□□	○◉○	■■■■■
Aircraft maintenance [76]	08/90	□■□□□	□■■	○○◉	■■■□□
Perspective Wall [68]	04/91	□□□■□	□□■	○◉○	■□□□□
Tree-Maps [56, 57, 103, 112]	10/91	□□□■■	■□□	◉○○	■■□□□
Stretch Tools [98]	09/92	□□■□□	□□□	○○◉	■■■■■
Morphing [99]	11/93	□□■□□	□□■	○○◉	■■■■■
Graph-oriented:					
Generalized FEVs of trees [34]	04/86	□■□□□	□■■	○◉○	■■□□□
SemNet (implicit) [30]	1988	■□□□□	□□□	○◉○	■■■■□
SemNet (clustering) [30]	1988	□■□□□	□■■	○◉○	■■■■□
Topographic networks [51, 52]	1989	□■□■□	□■■	○◉○	■■□□□
Cone trees (implicit/search) [89]	04/91	■□□□□	□□□	○◉○	■■□□□
Cone trees (gardening) [89]	04/91	□■□□□	□□□	○○◉	■■□□□
MVP display methods [75, 108]	09/91	□□■□□	□□■	○○◉	■■■■□
Associative networks [32]	10/91	□■□■■	□■■	○◉○	■■■□□
Interactive graph layout [50]	1991	□□□■□	■□□	◉○○	■■□□□
Graphical FEVs [96, 97]	05/92	□■■□□	□■■	○◉○	■■■□□
Abridgment [74]	05/93	□■□■□	■■■	○○◉	■■■■□
Variable Zoom [100]	05/93	□■■□□	□□■	○○◉	■■■■□
Fractal views of trees [63]	08/93	■■□■□	□□■	○◉○	■■□□□
PLUM [88]	08/93	■□□□■	■□□	◉○○	■■■■□
Layout-indep. FEVs [78, 79]	08/93	□□□■□	□□■	○○◉	■■■■□
Continuous Zoom [6]	05/94	□■■□□	■■■	○○◉	■■■■□
CATGraph [60]	11/94	□□■□□	□□□	○○◉	■■■■□

Table 3.3: A comparison of existing fisheye view techniques.

4. *Inputs* – class of inputs supported by the EA; classes are listed from the least to the most expressive – each class is fully contained in the next more expressive class:
   - *Sequences* (s)*:* one-dimensional data (*e.g.,* time line).
   - *Hierarchies* $(h) \supset s$*:* hierarchical graphs (*e.g.,* a tree or a forest of trees).
   - *Flat graphs* $(g) \supset h$*:* general graphs.
   - *Nested graphs* $(n) \supset g$*:* graphs in which vertices may be (hierarchically) nested.
   - *Beyond nested graphs* $(*) \supset n$*:* any graphical visualization.

The first limitation of existing fisheye view approaches is the lack of facilities for encoding priority algorithms. Most existing techniques employ fixed priority algorithms which are typically domain-specific and inflexible. One way to circumvent this problem is by assuming that the client (program or user) supplies the necessary priority values; among the existing techniques, however, this approach was limited to specifying API values. In order to create fisheye views, the client would need to supply the visualization system with n^2 pairwise distances, and while this can be done, such static approaches are limited as the user is not afforded the ability to define notions of similarity interactively. A flexible high-level specification method is a must in order to create visualization tools that can be easily customized to support a broad variety of applications. This was one of our primary design goals for the Graphite visualization system. Thus the first serious limitation of existing approaches is **specialized hard-coded priority algorithms**.

The second limitation of existing approaches is evidenced by the nearly empty *adorned* emphasis technique column. Non-geometric graphic properties such as (colour) hue and saturation can be very effective for creating emphasis in a visualization, as shown in [5, 111], for example. The table only tells part of the story, as only 2 of the 3 techniques that generate adorned views have dynamic mapping capability. The remaining techniques were primarily filtering-distorting hybrids. Thus the second serious limitation of existing approaches is **static and limited presentation strategies**.

Chapter 4

The Proposed Approach

In this chapter we describe our proposed technique for generating dynamic FEV visualizations by combining DM and DQ with active database views. The chapter is organized as follows. Section 1 outlines the basic philosophy behind our approach. Section 2 summarizes relevant background material relating to databases and database views. Section 3 details the event-action architectures of interactive data visualization systems and shows how the DM, DQ and active database view components are interrelated. Next we focus on the active view component which comprises our proposed view definition language, Dye (Section 4), and algorithms for generating database view update programs from Dye view definitions which are invoked in order to re-evaluate dynamic attribute values in response to various events (Section 5). Finally, Section 6 summarizes how Dye can be used to encode views that simulate a number of existing fisheye view techniques.

4.1 Motivation and Philosophy

Our approach is based on the following key observations.

1. The effectiveness of dynamic queries and dynamic mapping is directly tied to the availability of suitable multidimensional data. Masking with DQ is only useful if the data contains attributes that allow users to quickly identify relationships, groups, or exceptions in data (*i.e.,* to facilitate database searching and exploration tasks). The overall effectiveness of DM is similarly tied to the availability of a sufficient number of suitable quantitative attributes that could be dynamically bound to graphic properties.

2. Both DQ and DM are flexible domain-independent techniques, in that they are not tied to particular application domains, and can be combined in arbitrary ways. However, current DQ systems do not have sufficiently powerful view definition facilities to permit similarity queries.

3. Existing FEV techniques do capture the notion of similarity, but they are not domain-independent – most techniques involve the use of specialized hard-coded priority algorithms and employ static and limited

presentation strategies.

While the first observation suggests that it can be very important to have a mechanism to derive new attributes from existing ones, the second and third observations suggest that it may be desirable to combine the strengths of DQM interfaces and FEV techniques.

The crux of our proposed approach is the observation that a view definition language is precisely the "glue" that is needed to derive attributes and to integrate DQ and DM with FEVs.

Thus in our approach:

1. An appropriate view definition language is used to derive FEV degree of interest functions (as similarity attributes), thereby providing the means to encode flexible priority algorithms in a simple and elegant manner, overcoming the first limitation of existing FEV techniques (specialized hard-coded priority algorithms).

2. Once we derive similarity attributes as above, DM is used to automatically generate a broad variety of FEV visualizations in a flexible manner, overcoming the second limitation of existing FEV techniques (static and limited presentation strategies).

Thus the combination of flexible views, DQ and DM can be used to generate a broad variety of interactive visualizations characterized along two dimensions: the nature of the derived similarity attributes, and the collection of DM graphic properties. We have called this class of interactive visualizations Dynamic Fisheye Views to emphasize the fact that we exploit DQ and DM to simulate existing FEV techniques.

Finally, since derived similarity attributes can participate in dynamic queries just as any other (base or derived) attributes, by manipulating similarity attributes with DQ components, DQ can be used to evaluate similarity queries by direct manipulation.

We first described the use of view definition languages to encode FEV degree of interest functions in [80]. At that time, however, we suggested that view definition languages should also be used to compute static mappings between attributes and graphic properties. Thus our current approach both simplifies and enhances our original proposal by eliminating the added complexity associated with the embedding of static mapping rules in a view definition language, and by exploiting the flexibility of DM.

In what follows, note the distinction between the declarative view definition language Dye, and its implementation in the Graphite system. Dye is an abstract declarative language that can be implemented in many ways. Graphite, summarized in Chapter 2, is a visualization system that contains a database component (see Appendix A) that implements Dye views.

4.2 Background

The *relational database model* [15] is based on the concept of the set-theoretic relation, which is the subset of the Cartesian product of a list of domains. A *domain* is a set of values, and the Cartesian product of domains

$D_1, D_2, \ldots, D_k$ is denoted by $D_1 \times D_2 \times \cdots \times D_k$, which is the set of k-tuples $(v_1, v_2, \ldots, v_k)$ such that $v_i \in D_i$. A *relation* that is any subset of the Cartesian product of a set of k domains is set to have *arity* k. The members of the relation are called *tuples*; a tuple $(v_1, v_2, \ldots, v_k)$ has k components.

Relations are often called *tables*, where each *row* is a tuple and each *column* corresponds to one component. The columns are usually given names, called *attributes*; the set of attribute names is called the *relation scheme*.

In contrast to the relational database model, the main features of the *object-oriented* database model are:

1. *Object identity*. Unlike tuples in the relational model, objects in the object-oriented model have unique addresses through which their fields (attributes) are accessed.
2. *Complex types*. New data types (classes) can be formed by combining existing types; for example, it is possible to have set-valued fields. *Methods* serve as operations on objects of a particular class.
3. *Type hierarchy*. Types can be specialized through class inheritance.

While operations in the object-oriented model can be arbitrary methods, most systems provide a few basic operations such as selection (which returns a set of pointers to objects that pass the selection criteria).

There are two notations for expressing operations on relations: relational algebra (an algebraic notation) and relational calculus (a logical notation). The operands of relational algebra are either constant relations or variables denoting relations of fixed arity. There are five basic operations in relational algebra.

1. *Union.* The union of relations R and S, denoted $R \cup S$, is the set of tuples that are in R or S or both.
2. *Set difference.* The difference of relations R and S denoted $R - S$, is the set of tuples in R but not in S.
3. *Cartesian product.* The product of relations R and S of arity k_1 and k_2, respectively, denoted $R \times S$, is the set of all possible $(k_1 + k_2)$-tuples whose first k_1 components form a tuple in R and whose last k_2 components form a tuple in S.
4. *Projection.* A projection on a relation R of arity k, denoted $\pi_{i_1, i_2, \ldots, i_m}(R)$, where i_j are distinct integers in the range 1 to k, is the set of m-tuples $a_1 a_2 \cdots a_m$ such that there is some k-tuple $b_1 b_2 \cdots b_k$ in R for which $a_j = b_{i_j}$ for $j = 1, 2, \ldots, m$.
5. *Selection.* Selection on a relation R, denoted $\sigma_F(R)$, is the set of all tuples μ for which the formula F is true. F may involve operands that are constants or component numbers, the latter denoted $\$i$, the arithmetic comparison operators $<, =, >, \leq, \neq, and \geq$, and the logical operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not).

A query language that can simulate relational algebra is said to be *complete* [114]. Most commercial relational query languages, however, are more than complete. Typical additional features include:

1. *Arithmetic.* Selections in algebraic expressions can involve arithmetic computation in addition to comparison.

2. *Aggregation.* Operations such as count, average, sum, minimum, or maximum can be applied to columns of a relation to obtain a single quantity. Although there is no agreed-upon notation for aggregate operators in relational algebra, one proposal is $\mathcal{F}$ AgOp R_i (R) [29], where AgOp is one of the standard aggregate functions (minimum, maximum, count, sum, average), R is a relation, and R_i is the attribute of R being aggregated over.

A less common feature of relational query languages is the ability to compute the transitive closure of a relation. One means to add this capability is to permit query languages to compute *generalized transitive closure* operations based on the *closed semiring* algebraic structure [115]. This approach can be used to compute three transitive closure-like operations:

1. transitive closure;
2. all-pairs-shortest-paths; and,
3. parts explosion.

A query in which fields are restricted to a range of values rather than a single value is called a *range query*, while a query in which several (but not all) fields are restricted to single values is often called a *partial-match query*. A number of data structures for evaluating range and partial-match queries (*e.g.,* partitioned hash functions, k-d-trees) have been studied and compared [55, 93, 94].

An *integrity constraint* is a restriction on some aspect of a relation. There are two different kinds of restrictions: structural constraints such as functional dependencies, and semantic constraints that restrict the values of an attribute to some range or express arithmetic relationships among several attributes. One example of the former is the *unary inclusion dependency* (or *existence restriction*), which states that the domain of one attribute must be a subset of the domain of another attribute. For example, such a constraint can be expressed in relational algebra as: $\pi_{A_i}(R) \subseteq \pi_{A_j}(S)$.

A *view* or *subscheme* is a portion of the conceptual database or an abstraction of part of the conceptual database. Most database systems provide a facility for declaring views, called a subscheme data definition language (or view definition language) and a facility for expressing queries and operations on the views, called a subscheme data manipulation language (or view manipulation language). While a view is often a small conceptual database, a view can also contain data that is constructed (computed) from the conceptual database – this *derived* data is not present in the conceptual database.

An *update* operation on a relation is one of: insert tuple, delete tuple, or modify tuple. Object-oriented languages support two additional update operations: add (object to a set) and remove (object from a set) which, in contrast to insert and delete, do not create or destroy objects. A *view update* is an update operation specified through a view. Not all views can be updated – the problem of how updates on views can be translated into appropriate updates on the underlying relations is discussed in [21, 61]. View updates in object-oriented models can be simplified if operations are *object-preserving* [101], meaning that the results of queries are pointers to the input objects (rather than new objects or tuples), allowing updates to immediately propagate from views to base objects and vice versa.

View maintenance is the inverse problem of how updates on database relations should be reflected in views defined in terms of these relations. Production rules derived from an analysis of the view definition can be used to determine changes to views [119, 120]. As recomputing views can be expensive, several incremental methods for view maintenance have been proposed [10, 41, 116]. *View redefinition* refers to a change in a view definition; the usual way to update a view upon redefinition is to compute the new view from scratch; a more efficient alternative may be to exploit existing views through *view adaptation* [42].

4.3 Visualization System Architectures

In this section we describe how the main components of DQM systems are interrelated through an evolving sequence of high-level event-action architectures depicted as directed acyclic graphs. In particular, this allows us to situate the view definition component within the framework of DQM systems. We discuss the related incremental view maintenance techniques in the following section.

4.3.1 Static DQ Systems

The simplest architecture is that of a static DQ system, as shown in Figure 4.1. Dark nodes represent events while light nodes represent actions. The *MODIFY DQC* event is generated when a user manipulates a dynamic query component. This event causes a dynamic (selection) query to be evaluated (*EVAL Dynamic Query*), which in turn causes objects to be added or removed (filtered) from the display (*FILTER*).

The naive way to evaluate the dynamic selection query is simply to compute the result from scratch. A more efficient approach is to view this problem as an instance of *view redefinition*, which permits us to exploit the result of the previous query through view adaptation [42]. In particular, we can keep track of the set of displayed objects and the set of masked objects; two scenarios are then possible:

- If a DQ component is manipulated to narrow the range of the selection, we evaluate the selection query against the set of currently displayed objects; objects that do not satisfy the selection criteria are moved from the set of displayed objects to the set of masked objects (*i.e.,* object pointers are added or removed).
- If, on the other hand, a DQ component manipulation enlarges the range of the selection, we evaluate the revised selection query against the set of masked objects; objects that satisfy the selection criteria are moved from the set of masked objects to the set of displayed objects.

Another improvement can be achieved by exploiting the fact that DQ components are manipulated one at a time. One approach is to maintain a count for each object [55]; when a DQ component masks an object, its count is incremented, and when it unmasks an object, its count is decremented; when the count changes from 1 to 0 the object is displayed, and when it changes from 0 to 1 it is masked. This simple technique permits us to greatly simplify selection queries – each selection query need only contain a single criterion – the one corresponding to a single DQ component modification.

Finally, note that even greater efficiency can be achieved by incorporating specialized data structures for evaluating range and partial-match queries; several alternatives are compared in [55], however, most proposed data structures have considerable space overheads. Furthermore, in the above study [55] it was assumed that databases were static (database updates were not permitted), thus the cost of updating such structures in the presence of updates and its impact on the performance of DQ systems remains to be studied.

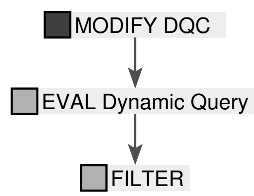


Figure 4.1: Event-action graph for static DQ system.

4.3.2 Static DQM Systems

The next more powerful architecture incorporates dynamic mapping in addition to DQ (*i.e.,* a DQM system), as shown in Figure 4.2. The *MODIFY DM* event is generated when a user interactively establishes a mapping between a data attribute and a graphic property. Notice that in this simple architecture, DQ and DM are independent, meaning that DQC events do not affect dynamic mapping and vice versa.

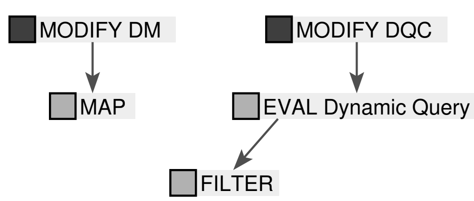


Figure 4.2: Event-action graph for basic DQM system.

4.3.3 Static DQM Systems with Dynamic Attributes

The architecture of a DQM system that supports dynamic attributes is shown in Figure 4.3. In this arrangement, the *UPDATE dynamic attribute* event corresponds to users adding or removing focal points or path sources or sinks, modifying regular expression patterns, or redefining similarity attribute expressions. Such events cause dynamic attribute values to be recomputed, as indicated by the *RECOMPUTE View* action. The recompute view action, in turn, must be propagated to the DM and DQ components in order to maintain display invariants:

- if a dynamic attribute is mapped to a graphic property, then the DM component must update this graphic property for each object to reflect the change in the values of the dynamic attribute (*e.g.,* adding a focal point causes a similarity attribute to be recomputed which in turn causes the colour of similar objects to become highly saturated and the colour of dissimilar objects to become pale);

- when dynamic attribute values are modified, we must also update the settings of the corresponding dynamic query component to reflect these changes:

  1. for example, we must adjust slider extrema values, or the shape of frequency distribution histograms;
  2. next we must decide what to do with existing DQ component settings (*e.g.,* the positions of the slider minimum and maximum, or which histogram bars should be displayed or masked); the approach taken in the Graphite system, is to maintain histogram bar visibility (*i.e.,* if a bar was masked it remains masked and if it was displayed it remains displayed); thus although the histogram shape and extrema may change, the effect of changing a dynamic attribute is to mask the same proportion of attribute value ranges;

- finally, the dynamic selection query must be re-evaluated causing objects to be shown or masked to reflect the modified selection criteria.

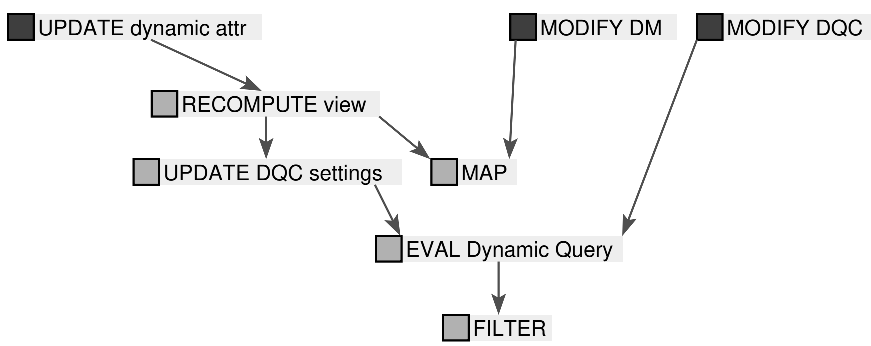


Figure 4.3: Event-action graph for static DQM system with dynamic attributes.

4.3.4 Active DQM Systems

Figure 4.4 shows the architecture of a system that permits graph updates. The *UPDATE base attr* event is generated when an external data source (or user) updates one of the base classes (vertices, arcs, or blobs), by inserting, deleting, or modifying an object.

The effect of updating a base object is similar to the effect of updating a dynamic attribute; however, insertions and deletions of base objects cause all derived attributes to be recomputed (rather than just dependent dynamic attributes). Another difference between the two events, is that updates to base objects also generate events related to maintaining referential integrity; for example, inserting an arc object causes an arc pointer to be automatically added to the set of incoming-arcs (`IA(head(a))`) and outgoing arcs (`OA(tail(a))`) of its head and tail vertices respectively, while deleting an arc causes its pointer to be removed from these sets. Similarly, deleting a vertex causes all arcs incident to that vertex to be deleted in order to maintain referential integrity.

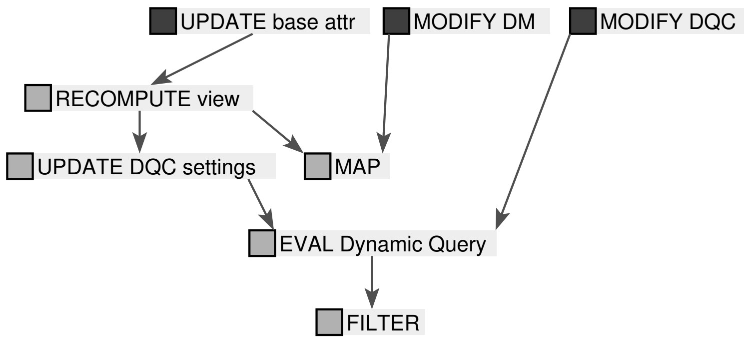


Figure 4.4: Event-action graph for active DQM system.

Finally, Figure 4.5 shows the event-action architecture of the complete active DQM system – one that incorporates dynamic queries, dynamic mapping, graph updates, and dynamic attributes. This architecture is realized by the Graphite visualization system.

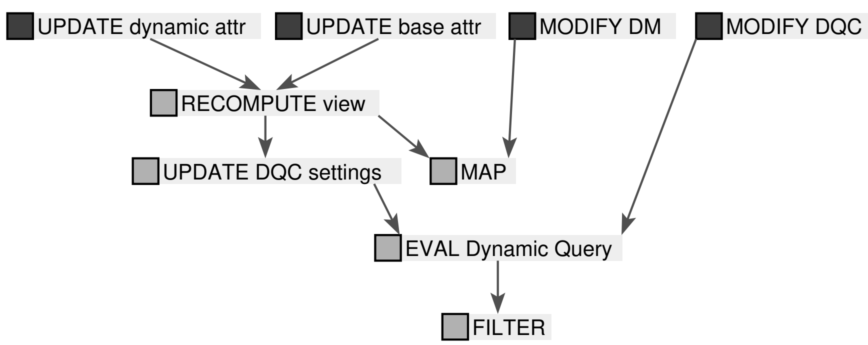


Figure 4.5: Event-action graph for active DQM system with dynamic attributes.

4.4 The Dye View Definition Language

Dye is a database view definition language which we propose for the purpose of defining and maintaining active views of hygraphs.

The Dye language is an object-oriented algebra that employs schema-preserving relational algebra operations[1] with object-preserving semantics [101]. Objects (and their data) are created and destroyed by database insertions and deletions, and can be added to or removed from any number of sets. The idea of object-oriented algebras is not new; one similar model is employed in the COCOON system [101]. One goal of such hybrid languages is

[1] The schema-preserving relational algebra operations are set union, set intersection, set difference, and selection.

to exploit the strengths of both approaches, namely: the declarative nature of relational languages, and the more flexible modelling features and more direct implementation strategies afforded by object-oriented languages.

4.4.1 Core Features

In this Section we describe the core view definition features of Dye. By *core*, we have attempted to capture a set of features that could be easily implemented with most commercial database systems. In later sections, we describe several extensions, such as the ability to compute transitive closure and shortest paths, that may be not as readily realizable with current commercial systems.

We describe the language in the following enumerated format.

1. All classes are derived from base class `OBJ`:

```
OBJ(id:STRING,          % unique identifier
    label:STRING,       % textual label
    mag:DOUBLE,         % focal point magnification
    A:DOUBLE[]          % an array of numeric attribute values
```

2. All objects belong to one of the following three derived classes:

```
V:OBJ(              % a vertex object
    IA:SETOF(A),       % set of incoming arcs
    OA:SETOF(A),       % set of outgoing arcs
    pb:B,              % parent blob or 0
    CB:SETOF(B)        % set of child blobs
)
A:OBJ(              % an arc object
    head:V,            % vertex at the head of the arc
    tail:V             % vertex at the tail of the arc
)
B:OBJ(              % a blob object
    pv:V,              % parent vertex
    CV:SETOF(V)        % set of child vertices
)
```

3. The data members of each object are referred to using nested notation: `Member(optr)`, as in `pv(b)` to refer to the parent vertex of blob `b`.

4. For syntactic brevity, we denote numeric attributes as `o[i]` rather than the lengthier `A[i](o)`.

5. Set expressions are formulated using the schema-preserving relational algebra operators: `UNION`, `INTERS`, `DIFF`, `SELECT`.

6. All sets are homogeneous (*e.g.,* cannot have pointers to both vertices and arcs in the same set).

7. A view definition consists of set declarations of the form:

```
s := SetExpr
```

   where `s` is a set-valued variable, and expressions are built up with relational algebra operators, and derived numeric attribute declarations in one of the following two forms:

```
r:R r[i] := NumericExpr
r:R { r[i] := NumericExpr ; r[j] := NumericExpr ; ... }
```

where `r` is a looping variable that ranges over the objects in set `R`, and `NumericExpr` is a numeric expression involving:

- numeric constants
- references to numeric attributes of looping variables, denoted `r[j]`
- the usual mathematical operators and functions (see, for example, Section A.4)
- an *if-then-else* expression in the form `Expr ? TrueExpr : FalseExpr` which evaluates `Expr` and if it is non-zero returns `TrueExpr` otherwise `FalseExpr`
- aggregate operators of the form: `AGOP([s:]S,NumericExpr)` or `CNT(S)`, where `AGOP` is one of: `SUM`, `AVG`, `PROD`, `MIN`, and `MAX`, `s` is a variable that ranges over the set `S`, and `S` is either a set-valued data member of an enclosing looping variable, or one of the base or derived sets.

8. Attributes and sets cannot be multiply defined (*i.e.,* each attribute or set can appear exactly once as the left-hand-side of a definition).

9. Circular attribute references (including self-references) are not permitted in definitions (*e.g.,* no cyclical dependencies among attributes just as there cannot be cyclical dependencies among the cells in a spreadsheet).

Notice that the core component of the Dye language can be thought of as a spreadsheet language in which each "cell definition" automatically applies to an entire column of cells (all objects in a set), and in which additional relational operators are used to create new columns of cells (sets of objects). As such, an extended spreadsheet user interface is one alternative for enabling users to define derived attributes. Our similarity dialog box approach described in Section 2.7 is a simpler but less powerful alternative for interactively defining similarity attributes.

Next we present several examples. The simplest type of derived attribute is the class of single-variable expressions (expressions that involve just one looping variable). Following the airline flights example, suppose arc attributes 0 and 1 are the price and duration of a one-way flight. Then:

```
a:A a[2] := a[0] / a[1]
```

declares arc attribute 2 to be the ratio of flight price to flight duration, which permits flight prices to be compared independently of their points of departure and arrival.

Derived attributes can also be expressed using two or more looping variables by incorporating aggregate operators, as in:

```
v:V v[2] := AVG(a:IA(v),a[2])
v:V v[3] := CNT(a:UNION(IA(v),OA(v)))
```

which declares that for each vertex (city), attribute 2 is the average over the set of incoming arcs of arc attribute 2 (price to duration ratio) which indicates the relative cost of incoming flights, and vertex attribute 3 is the total number of flights entering or leaving each city.

Set-based operations can be used to declare subsets of objects, as in:

```
sources := SELECT(v:V,CNT(IA(v)) = 0)
sinks   := SELECT(v:V,CNT(OA(v)) = 0)
```

which declare `sources` to be the subset of vertices that do not have any incoming arcs, and sinks to be the subset of vertices that do not have any outgoing arcs (with luck, both sets would be empty in the airline flights example).

Navigation through pointers is used to compute derived values involving related objects, as in:

```
a:A a[3] := a[0] / AVG(b:IA(head(a)),b[0])
```

which declares that for each arc (flight), attribute 3 is the ratio of attribute 0 to the average value of attribute 0 over the set of arcs incident on an arc's head (destination city), which computes the ratio of a flight's price to the average price of all flights with that destination.

4.4.2 Focal Points and Similarity Functions

The following three additional sets are used to maintain focal points:

- a set of focal vertices `VF` $\subseteq$ `V`,
- a set of focal arcs `AF` $\subseteq$ `A`, and
- a set of focal blobs `BF` $\subseteq$ `B`.

Object pointers are added and removed from these sets in response to user actions – typically by pointing and clicking on the corresponding object in the visualization.

Aggregation is then used to derive similarity attributes that capture notions of similarity between, say every vertex and the set of selected focal vertices. Thus in general, a similarity function is any mathematical expression that contains at least one term of the form:

$$AgOp(f : [VF|AF|BF], Expr)$$

where $AgOp$ is one of the standard aggregate functions, $Expr$ is a mathematical expression involving attribute values of an enclosing looping variable and the focal point f; in Graphite mathematical expressions are formed using the mathematical operators and functions described in Section A.4, and the specialized functions (*e.g.,* transitive closure, shortest paths) described in the following sections. This general form can be used to capture both geometric [40] (*e.g.,* Minkowski metric [95]) and feature-based [113] notions of similarity, as well as domain-specific similarity functions such as those used in the Affinity Browser [83].

As an example, the following view captures the notion of Euclidean proximity (the -1 scale factor is used to invert Euclidean distance):

```
v:V v[2] := -1 * AVG(f:VF,hypot(v[0]-f[0],v[1]-f[1]))
```

which declares that for each vertex, attribute 2 is proportional to the average proximity of the vertex to the set of vertex focal points in the plane defined by vertex attributes 0 and 1 (note that $hypot(x, y) = \sqrt{x^2 + y^2}$).

In addition, as shown in the class definition in Section 4.4.1, each object can be assigned an associated "magnification" (or relevance) which can be assigned to focal points and used in similarity expressions. For example, the following view is a variant of the above which uses magnification as a means to scale the contribution of each focal point: larger magnification values lead to higher proximity values.

```
v:V v[2] := -1 * AVG(f:VF,hypot(v[0]-f[0],v[1]-f[1]) / mag(f))
```

Thus the provision of focal point magnification scale factors enables users to not only select arbitrary sets of focal points, but also to specify their relative importance (with a slider, for example) thereby causing objects that are similar to more important focal points to have higher similarity values than those that are similar to less important focal points.

4.4.3 Transitive Closure

Since transitive closure (reachability) is a fundamental property of graphs, we have incorporated transitive closure in Dye by introducing the following operator:

```
TC(v,(forward|reverse|undirected),R)
```

which returns the set of vertices that can be reached from vertex `v`, where `v` is a vertex looping variable, and `R` is either a set of arcs or a set of blobs (indicating paths in the nesting hierarchy). The second parameter is a direction that specifies whether arcs (or inclusion arcs) should be traversed in the forward direction, in the reverse direction, or both. Transitive closure can be computed by a single depth-first traversal of the graph starting at vertex `v`, and takes $O(|R|)$ time since each arc in the transitive closure must be traversed once. This can makes transitive closure a potentially expensive operation as in:

```
v:V v[0] := CNT(TC(v,forward,A))
```

which declares that for each vertex, attribute 0 is the number of vertices that can be reached following forward paths starting at the vertex, and requires $O(|V| \cdot |A|)$ time to compute. As such, transitive closure should be used carefully. One situation where it can be used efficiently, is to compute the depth of each vertex in a hierarchical graph or in the blob nesting hierarchy as in:

```
v:V v[0] := CNT(TC(v,reverse,B))
```

which, if implemented naively, would cost $O(|V| \cdot \log |V|)$ to compute. An optimizer could be used to implement the above operation in $O(|V|)$ time by setting the depth of the root to 0, and with a single depth-first traversal of the hierarchy setting the depth of each node to be one greater than the depth of its parent in the hierarchy (since the nesting graph is hierarchical).

4.4.4 Shortest Paths

The notion of shortest paths plays a fundamental role in many graph-oriented applications.

We have incorporated the all-pairs-shortest-paths algorithm in Dye by first assuming that there are two additional system-maintained sets:[2] a set of path sources $S \subseteq V$, and a set of path sinks $E \subseteq V$, and then by introducing the following operator and two functions:

```
p := APSP([v:]RV,[a:]RA,(forward|reverse|undirected),VExpr,AExpr)
SP(u,v,p)
NPATHS(p)
```

The `APSP` operator declares that variable `p` can be used to access information about shortest paths between any pair of vertices in set `RV` obtained by traversing the (adjacency or inclusion) arcs in set `RA` in the given direction(s), and evaluating the expression `VExpr` for each vertex along the path and expression `AExpr` for each (adjacency or inclusion) arc along the path. The next two functions can be used in arithmetic expressions. The first returns the shortest path distance between vertices `u` and `v` given the notion of path associated with variable `p`. The second returns the number of shortest paths between the current set of sources `S` and sinks `E` that object `o` partakes in.

The APSP can be computed in $O(|V|^3)$ time using a variant of Floyd's algorithm [4, 31] that considers not only arc weights but also vertex weights (see Section B.1). The function `SP` is just a table look-up and thus can be evaluated in constant time. If we maintain a count of the number of current shortest paths that any vertex, arc, or inclusion arc partakes in, then the function `NPATHS` can likewise be evaluated in constant time (see Section B.2).

In the following example we show how APSP can be used to compute transitive closure, that makes it possible to subsequently test whether one vertex is in the transitive closure of another in constant time.

```
p := APSP(V,A,undirected,0,1)
v:V v[0] := CNT(SELECT(v:V,SP(u,v,p) >= 0))
```

The above declares that for each vertex `v`, attribute 0 is the number of vertices in the (undirected) connected component that contains `v`.

4.4.5 Special-purpose Operators

Normalized Attribute Values

Often it is convenient to scale attribute values into some range. To facilitate this, we have provided a function that will return the normalized value of an attribute, defined as:

```
N(r[i]) = ( r[i] - MIN(s:R,s[i]) ) / ( MAX(s:R,s[i]) - MIN(s:R,s[i]) )
```

The above simply says that `N(r[i])` is the value of `r[i]` scaled into the interval $[0, 1]$. As an example, the following declaration is an alternative way to encode Euclidean proximity to the one given in the previous Section.

```
v:V v[3] := AVG(f:VF,hypot(v[0]-f[0],v[1]-f[1]))
v:V v[2] := 1 - N(v[3])
```

In this version, attribute 3 is the Euclidean distance, while attribute 2 is the Euclidean proximity in the range $[0, 1]$.

[2] Path sources and sinks are selected interactively by pointing and clicking as discussed in Section 2.8.

Note that in practice, the attribute extrema values are not repeatedly recomputed, but are stored and evaluated incrementally in response to database updates. Furthermore, as the extrema values are materialized, they can be directly accessed by the DM component when mapping database attributes onto graphic properties, which facilitates the scaling of attribute values to intervals of graphic properties such as node size (*e.g.,* 10 pixels to 50 pixels), link width (*e.g.,* 1 pixel to 5 pixels), or hue (*e.g.,* 0 degrees to 330 degrees).

Regular Expression Matching

The notion of Graphite's regular expression matching functionality described in Section 2.5 is captured by the Dye `REGEXP` construct. The values of a dynamic attribute defined in terms of `REGEXP` are updated every time the user interactively modifies the associated regular expression string; that is, for each object, the value of the attribute is set to 1 or 0 depending on whether its identifier or label matches the current value of the regular expression. The following example declares that the value of vertex attribute 0 for each vertex will be set to either 1 or 0 every time the user modifies the associated regular expression.

```
v:V v[0] := REGEXP()
```

User-Defined Similarity Attributes

The notion of Graphite's user-defined similarity attribute is captured by the Dye `USERSIM({Arg1,...,ArgN})` construct, where `ArgJ` refers to an argument in one of the following forms:

- n where n is the number of a quantitative attribute (see Section 2.1)
- n:o where n is the number of an ordinal (or nominal) attribute
- m-n where $m < n$ and m, n are quantitative attributes, and is the short-hand for the sequence of attributes numbered $m, m+1, \ldots, n$ inclusive.

The following declaration, states that vertex attribute 6 is a user-defined similarity attribute which can be defined in terms of the quantitative attributes 0 through 4 and the ordinal attribute 5.

```
v:V v[6] := USERSIM({0-4,5:o})
```

Declarative Layout

In the Graphite realization of Dye, each blob container has an internal `layout` field that can be manipulated to control how the nodes directly contained in the blob should be drawn. This feature allows Dye to be used to declaratively describe complex composite layouts and is one way to automate Henry's interactive subgraph selection scheme [49, 50]. In the following example, the child nodes of all blobs that contain more than 10 child nodes are drawn by the *grid* layout algorithm, otherwise by the *circle* layout algorithm. A complete list of layout algorithms and parameters is given in Section A.3.

```
b:B layout(b) := CNT(CV(b)) > 10 ? "grid" : "circle"
```

4.5 View Evaluation and Maintenance

Given a declarative view specification, a query evaluator must be able to generate a program to materialize views of database instances. We discuss two possible approaches.

In the first scenario, we assume that the database is static and show how to obtain a program to materialize a view based on the view specification.

In the second scenario, we assume that the database is active; that is, base objects can be inserted or deleted during the lifetime of an interactive session. While it is possible to recompute the entire view from scratch each time a base object is inserted or deleted, doing so can be extremely inefficient. Instead, it is desirable to be able to make relatively small updates to the materialized view to reflect updates to base objects. Thus in the active database scenario, we must create a program that will perform view maintenance in the presence of database updates. More specifically, the purpose of such a program is:

- to ensure that all affected derived attributes are updated when an object is inserted or deleted from a base class;
- to ensure that all affected derived attributes are updated when an object is added or removed from a dynamic set (focal points, path sources and sinks); and,
- to ensure that all affected derived attributes are updated when a user modifies a user-defined similarity expression or regular expression pattern.

Our approach is to generate update programs based on an analysis of the dependencies in the view definition. The basic approach presented here is not new; one of the earliest related proposals was based on the use of a transformational framework for maintaining derived data [62]. The idea described therein was to replace costly global computations first suggested as a means to materialize views [107], by efficient incremental modifications. A more recent version of this approach is to generate production rules from view specifications [119, 120]; production rules are typically implemented as Event-Condition-Action rules [71]. Related incremental view maintenance techniques are also described in [10, 41, 116].

4.5.1 Materializing Views from Scratch

In order to generate a program to materialize a view from scratch, we begin by creating a *view parse graph* from the declarative view specification.

The vertices in the parse graph correspond to the constants, attributes, sets, and operations (mathematical, aggregate or relational) that appear in the view definition. Arcs connect inputs (expression arguments) to outputs (mathematical subexpressions, sets or derived attributes). Since Dye specifications cannot contain recursively defined derived attributes or sets, the parse graph of a (valid) Dye view specification is always a DAG.

For example, the following view definition gives rise to the parse graph shown in the left portion of Figure 4.6.

```
v:V {
  v[0] := CNT(a:UNION(IA(v),OA(v)))
  v[1] := AVG(a:UNION(IA(v),OA(v)),a[2])
  v[2] := v[0] + v[1];
}
a:A a[2] := a[0] / a[1]
```

Once we obtain a view parse graph, we compute its *view dependency graph*, which is the subgraph obtained by removing each vertex which does not refer to an attribute or set, and then adding a (u, v) arc between remaining vertices u and v if there is at least one $u - v$ path that does not pass through any other remaining vertex. The presence of a (u, v) arc in the view dependency graph implies that the value of v depends on the value of u, which means that in order to compute v, we must first compute u (if u is not a base attribute or base set). The dependency graph of the above view is shown in the right portion of Figure 4.6.

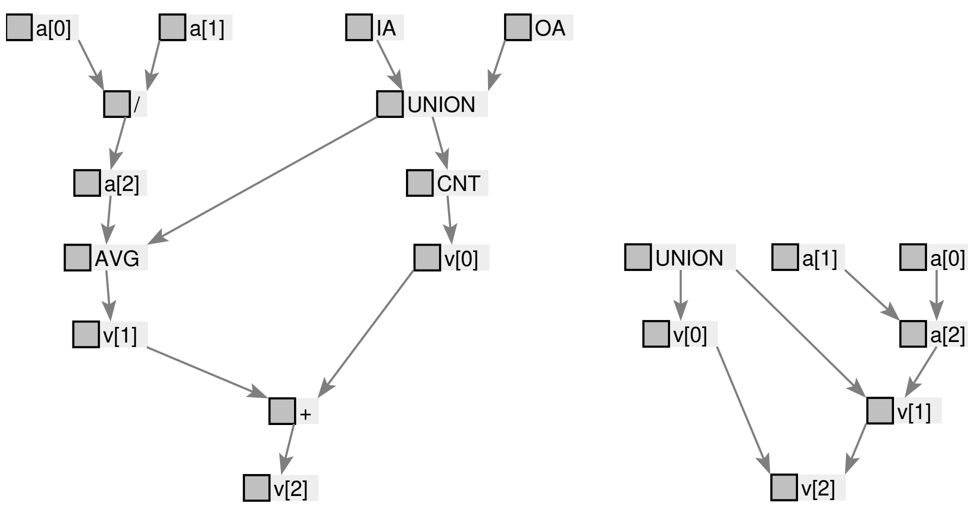


Figure 4.6: A view specification parse graph and its view dependency graph.

Recall that as previously suggested, attribute minima and maxima values are implicitly maintained for every (base and derived) attribute; the existence of materialized extrema values means that the `N(o[i])` normalization function can be evaluated in constant time, and that extrema values are available to the DM component (which must normalize attribute values in order to bind them to graphic properties).

With that in mind, the next step is to create a revised dependency graph which is constructed by:

- adding a pair of vertices of the form `mino[i]` and `maxo[i]` for each attribute, and pair of arcs connecting the attribute to each extremum vertex; and,
- adding a dummy root and an arc from the dummy root to each source in the original dependency graph.

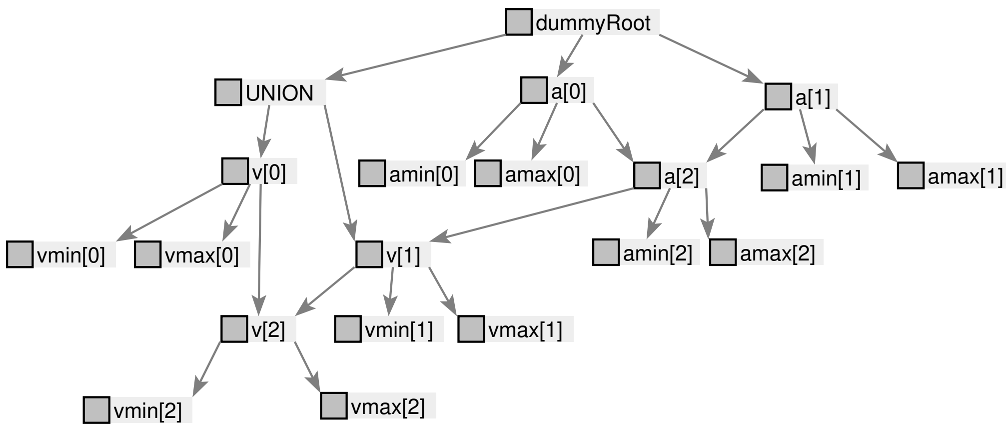


Figure 4.7: A revised dependency graph.

The revised dependency graph for the sample view is shown in Figure 4.7. Next we compute the *longest-path layer number* for each vertex in the revised dependency graph (rooted at the dummy root) using the algorithm given in Section B.3, which first topologically sorts the revised dependency graph using the algorithm given in Section B.4. The longest-path layer number assigned to each vertex is then used to generate a "schedule" (partial order) for computing each derived attribute or set. Each vertex is placed in its longest-path layer, the effect of which is to push all vertices down to the lowest level in the graph such that all arcs remain downward pointing. As the final step in generating the schedule, we remove all vertices that correspond to base attributes or base sets.

The schedule for the sample view specification is shown in Figure 4.8. Each layer in this graph corresponds to a "step" in the schedule – sets and attributes in the same layer are independent and thus can be computed in any order. Notice that if we were to evaluate attributes one at a time, we would require $O(n)$ loop iterations where n is the number of derived attributes. However, we can obtain significant performance improvements by grouping derived attributes that are defined over a common set and appear in the same layer and computing all attributes in a given group with a single loop iteration rather than one iteration per derived attribute.

The schedule shown in Figure 4.8 can then be used to generate a program to compute values for all derived attributes and sets; for example, the sample view specification would give rise to the following Graphite program (written in Tcl; see Appendix A).

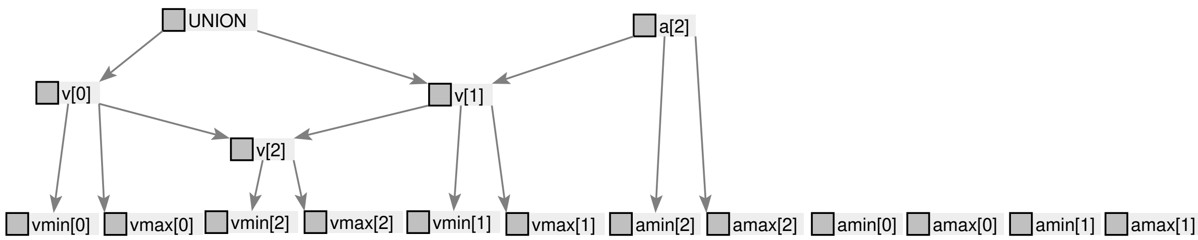


Figure 4.8: An evaluation schedule graph.

```
foreach a [grA] {
  grSet $a 2 [expr a($a,0) / a($a,1)]
}
foreach v [grV] {
  set setvar0 [grUnion [grIA($v)] [grOA($v)]]
  grSet $v 1 [grAvg a $setvar0 {a($a,2)}]
  grSet $v 0 [grCnt $setvar0]
}
foreach v [grV] {
  grSet $v 2 [expr a($v,0) + a($v,1)]
}
set vmin0 999999; set vmax0 -999999
set vmin1 999999; set vmax1 -999999
set vmin2 999999; set vmax2 -999999
foreach v [grV] {
  if { [expr a($v,0)] < $vmin0 } { set $vmin0 [expr a($v,0)] }
  if { [expr a($v,0)] > $vmax0 } { set $vmax0 [expr a($v,0)] }
  if { [expr a($v,1)] < $vmin1 } { set $vmin1 [expr a($v,1)] }
  if { [expr a($v,1)] > $vmax1 } { set $vmax1 [expr a($v,1)] }
  if { [expr a($v,2)] < $vmin2 } { set $vmin2 [expr a($v,2)] }
  if { [expr a($v,2)] > $vmax2 } { set $vmax2 [expr a($v,2)] }
}
set amin0 999999; set amax0 -999999
set amin1 999999; set amax1 -999999
set amin2 999999; set amax2 -999999
foreach a [grA] {
  if { [expr a($a,0)] < $amin0 } { set $amin0 [expr a($a,0)] }
  if { [expr a($a,0)] > $amax0 } { set $amax0 [expr a($a,0)] }
  if { [expr a($a,1)] < $amin1 } { set $amin1 [expr a($a,1)] }
  if { [expr a($a,1)] > $amax1 } { set $amax1 [expr a($a,1)] }
  if { [expr a($a,2)] < $amin2 } { set $amin2 [expr a($a,2)] }
  if { [expr a($a,2)] > $amax2 } { set $amax2 [expr a($a,2)] }
}
```

Several points are worthy of mention.

- Notice how vertex attributes 0 and 1 are computed with a single loop iteration, rather than two individual loop iterations.

- Notice how the vertex and arc extrema are computed with two loop iterations rather than 12 individual loop iterations. What makes this possible is the use of the incremental form of the MIN and MAX aggregate operators. Each aggregate operator has a simple incremental form that allows the aggregate value to be updated in constant time when an object is inserted into the set; most incremental forms for handling deletions similarly require constant time, the exceptions being the deletion of an actual minimum value for MIN, an actual maximum value for MAX, or of a 0 for PROD, all of which require recomputation costing linear time. The incremental forms of the aggregate operators are summarized in Section B.5.

- Notice that common subexpressions are identified and reused to avoid repeated computation, as done with the `UNION(IA(v),OA(v))` set expression in the preceding example.

4.5.2 Incremental View Maintenance of Dynamic Attributes

We have already distinguished attributes and sets as being either *base* or *derived*; derived attributes or sets can be further distinguished as:

- *static* – attributes or sets whose values, once derived, remain static
- *dynamic* – attributes or sets whose values change when one or more of the following events is triggered by a user action:
  1. add focal point;
  2. remove focal point;
  3. add path source or sink;
  4. remove path source or sink;
  5. change similarity function; and,
  6. change regular expression pattern.

We now describe how the technique to materialize views described in the previous section can be adapted to efficiently maintain dynamic attributes and sets in response to each of the above events.

The basic idea is to generate a program corresponding to each event; each such program is generated by a slightly revised version of the procedure described in the previous section. In particular, rather than generating a single dependency graph, we generate one dependency subgraph for each event rooted at specific vertices in the graph; the vertices in each dependency subgraph correspond to the attributes or sets that are dynamic with respect to the given event.

To facilitate the generation of update programs, and without loss of generality, we assume that attribute expressions in the view definition are either aggregate-free (do not involve any aggregate terms) or consist of a single aggregate term; generality is not sacrificed since aggregate terms in expressions can be replaced by attribute references. For example, the expression:

```
v:V v[2] := AVG(f:F,v[0]-f[0]) + SUM(f:F,v[1]-f[1]) + v[1] - v[0]
```

can be rewritten as:

```
v:V {
  v[i] := AVG(f:F,v[0]-f[0])
  v[j] := SUM(f:F,v[1]-f[1])
  v[2] := v[i] + v[j] + v[1] - v[0]
}
```

where `i` and `j` are two unused attribute numbers, and the three resulting attribute expressions are all either aggregate-free or consist of a single aggregate term. A related discussion of decomposable aggregate expressions appeared in [18].

Each attribute that is expressed in terms of an aggregate-free expression, is updated by simply evaluating the corresponding expression, while each attribute that is expressed in terms of a single aggregate term is updated by using an incremental form of the corresponding aggregate operation (see Section B.5).

For example, the dynamic attribute definition:

```
v:V v[3] := SUM(f:F,abs(f[0]-v[0]))
```

would give rise to the following Graphite update statement for an add event:

```
grSet $v 3 [expr a($v,3) + abs(a($f,0) - a($v,0))]
```

and the following update statement for a remove event:

```
grSet $v 3 [expr a($v,3) - abs(a($f,0) - a($v,0))]
```

In addition to maintaining dynamic attributes, the update program must also maintain dynamic derived sets. Set expressions are defined in terms of the relational operators `UNION`, `INTERS`, `DIFF`, and `SELECT`, each of which also has a simple incremental form, which, given an addition or deletion into an input set can update the result set in logarithmic time as opposed to linear time required for recomputation. The exact incremental forms of the relational operators are summarized in Section B.6.

For example, the dynamic set definition:

```
SE := UNION(S,E)
```

would give rise to the following Graphite update statement when an object `o` is added to either input set:

```
set SE [grUnion $SE {$o}]
```

and the following Graphite update statements when an object is removed from the first and second input set respectively:

```
set SE [grDiff $SE [grDiff {$o} $E]]
set SE [grDiff $SE [grDiff {$o} $S]]
```

As an example, consider the following extended version of the view specification presented in the previous subsection:

```
P    := APSP(A,V,undirected,0,1)
myF  := SELECT(f:VF,f[0] > 0)
v:V {
  v[0] := CNT(a:UNION(IA(v),OA(v)))
  v[1] := AVG(a:UNION(IA(v),OA(v)),a[2])
  v[2] := v[0] + v[1];
  v[3] := SUM(f:myF,abs(f[0]-v[0]))
  v[4] := NPATHS(P)
  v[5] := REGEXP()
  v[6] := USERSIM({0,1})
  v[7] := v[3] - v[4]
}
a:A {
  a[2] := a[0] / a[1]
  a[3] := NPATHS(P)
}
```

This view definition gives rise to the dependency graph shown in Figure 4.9.

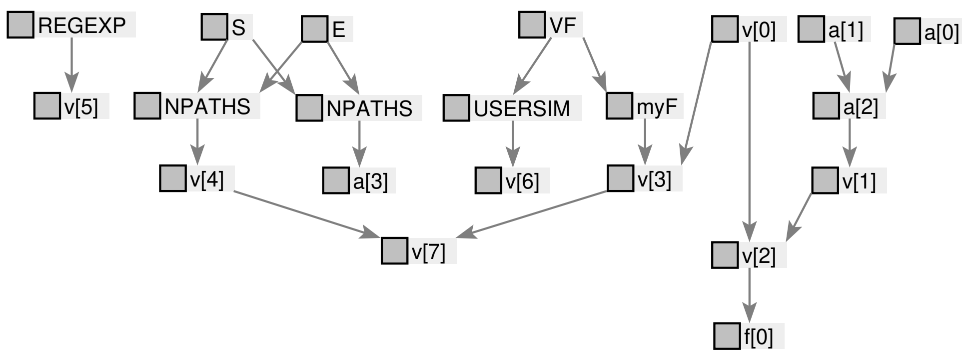


Figure 4.9: A dynamic view dependency graph.

Several features of the dependency graph require elaboration. In order to ensure that dynamic attributes can be maintained efficiently, we make the following restrictions and assumptions:

1. path expressions cannot refer to dynamic attributes (otherwise would have to recompute shortest paths every time a dynamic attribute was updated);

2. selection criteria cannot refer to dynamic attributes (which justifies our claim that `SELECT` has a simple incremental form – *i.e.,* the value of the condition part of the `SELECT` does not change);

3. similarity attribute aggregate expressions (expressions evaluated inside the aggregation) cannot refer to dynamic attributes, although derived attributes can be defined in terms of dynamic attributes; lifting this restriction would mean that changes in aggregate expression inputs would require aggregate values to be recomputed from scratch costing $O(n \cdot f)$ where n is the size of the relation and f is the number of focal points;

4. as presented, the `USERSIM` function aggregates over the complete set of focal points, and so is implicitly dependent on one of `VF`, `AF`, or `BF` (which explains the arc from `VF` to `USERSIM`); for greater generality, we could also have defined `USERSIM` in terms of some derived set of focal points, such as `myF` in the above example and allow users to select one of the predefined focal point subsets interactively (*e.g.,* via a small number of radio-buttons); and,

5. all `NPATHS` functions are implicitly dependent on the set of path sources and sinks (which explains the arcs from `S` and `E` to the two instances of `NPATHS`).

Having created a dependency graph, we can generate a program to materialize a view from scratch as before; we can also exploit the same dependency graph to generate programs to update dynamic attributes in reaction to each of the events enumerated at the beginning of this section.

By using a dummy root to tie together all sources in the dependency graph we were able to generate a program that computed all derived attributes. The trick to generating update programs for specific events is to apply the same procedure to subgraphs rooted at specific vertices in the dependency graph. In what follows, we summarize the dependency subgraphs associated with each event. Each dependency subgraph is revised by adding a pair of extrema vertices and arcs for every attribute in the subgraph; this in turn can be used to obtain an evaluation schedule and hence the final update program as done in the previous section.

Adding or removing a focal point. The addition or removal of a focal point affects all the derived attributes and sets that can be reached in the dependency graph from the corresponding focal point sets `VF`, `AF`, and `BF`. In our sample view definition, an addition or removal of a vertex focal point, reaches the dependency subgraph shown in Figure 4.10. This dependency graph gives rise to the schedule graph shown in Figure 4.11, in which the related attribute extrema updates are explicitly shown. The schedule graph then is used to generate the following update program when a vertex `f` is added to the vertex focal point set:

```
proc VFAdd {f} {
  global myF
  global vmin6 vmax6 vmin7 vmax7 vmin3 vmax3

  set modified0 0
  if { [expr a($f,0) > 0 } {
    set myF [grUnion $myF [list $f]]
    set modified0 1
  }
  if { $modified0 > 0 } {
    foreach v [grV] {
      grSet $v 3 [expr a($v,3) + abs(a($f,0) - a($v,0))]
    }
  }
  foreach v [grV] {
    grSet $v 6 [expr a($v,6) + [UserSim0 $f $v]]
    if { $modified0 > 0 } {
      grSet $v 7 [expr a($v,3) - a($v,4)]
    }
  }
  foreach v [grV] {
    if { [expr a($v,6)] < $vmin6 } { set $vmin6 [expr a($v,6)] }
    if { [expr a($v,6)] > $vmax6 } { set $vmax6 [expr a($v,6)] }
    if { $modified0 > 0 } {
      if { [expr a($v,7)] < $vmin7 } { set $vmin7 [expr a($v,7)] }
      if { [expr a($v,7)] > $vmax7 } { set $vmax7 [expr a($v,7)] }
    }
    if { $modified0 > 0 } {
      if { [expr a($v,3)] < $vmin3 } { set $vmin3 [expr a($v,3)] }
      if { [expr a($v,3)] > $vmax3 } { set $vmax3 [expr a($v,3)] }
    }
  }
}
```

Notice the use of the local variable `modified0` to flag whether the set `myF` is modified by the focal point insertion. In general, one such variable is created for every derived set that can be updated by the event; then every derived attribute that is dependent on one or more derived sets[3] is computed conditionally – if the sum of the flags associated with all such derived sets is greater than 0 (implying that at least one dependent set has been modified).

Adding or removing a path source or sink. The addition or removal of a path source or sink affects all the derived attributes and sets that can be reached in the dependency graph from the vertices corresponding to the `S` and `E` dynamic sets. In our sample view definition, an addition or removal of a path source or sink reaches the dependency subgraph shown in Figure 4.12.

Modifying a user-defined similarity expression. The modification of a user-defined similarity expression affects all the derived attributes and sets that can be reached in the dependency graph from the vertex associated with the corresponding `USERSIM` function. In our sample view definition, modifying the similarity function associated with vertex attribute 6 reaches the dependency subgraph shown in Figure 4.13.

[3] Attribute v is dependent on derived set u if there is a uv path in the dependency graph that does not pass through any other derived set w.

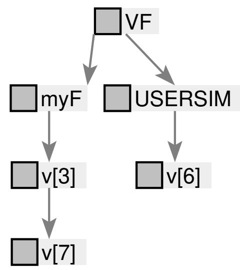


Figure 4.10: A dependency subgraph rooted at the set of focal vertices.

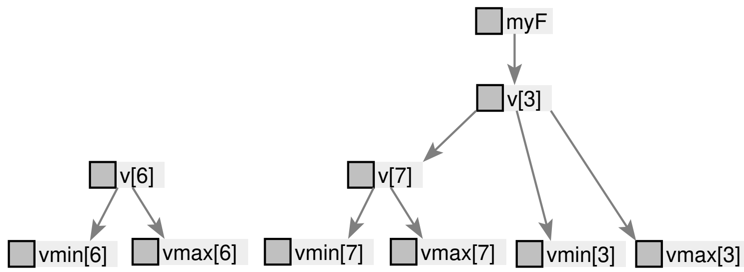


Figure 4.11: A schedule graph for adding or removing a vertex focal point.

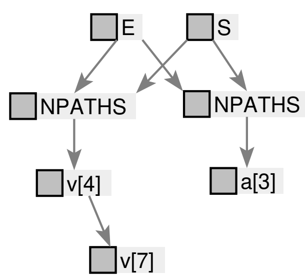


Figure 4.12: A dependency subgraph rooted at path sources and sinks.

Modifying a regular expression pattern. The modification of a regular expression pattern affects all the derived attributes and sets that can be reached in the dependency graph from the vertex associated with the corresponding `REGEXP` function. In our sample view definition, modifying the regular expression associated with vertex attribute 5 reaches the dependency subgraph shown in Figure 4.14.

4.5.3 Incremental Evaluation of Graph Updates

Graph updates consist of the following events:

1. inserting a vertex;
2. deleting a vertex;
3. inserting an arc;

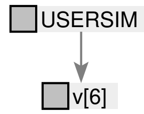


Figure 4.13: A dependency subgraph rooted at a user-defined similarity function.

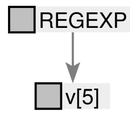


Figure 4.14: A dependency subgraph rooted at a regular expression matching function.

4. deleting an arc;
5. inserting a blob;
6. deleting a blob;
7. adding a vertex to a blob; and,
8. removing a vertex from a blob.

Notice that the transitive closure and shortest paths operations do not have efficient (linear or better) incremental forms. For this reason, Graphite applications that permit graph updates (*e.g.,* an interactive Web browser) do not use these features. One direction for future work may be to incorporate incremental (on-line) algorithms for maintaining transitive closure and shortest paths in limited cases.

To generate update programs for graph updates we can employ the procedure described in the previous Section by selecting appropriate subgraphs of the dependency graph. However, since graph updates change not only the values but the structure of graphs, the update programs must also peform actions related to the maintenance of referential integrity constraints.

Inserting or deleting a vertex. The insertion or deletion of a vertex affects all the derived attributes and sets that can be reached in the dependency graph from the vertex corresponding to the set of vertices `V`. The revised dependency graph for vertex insertions and deletions contains an arc from `V` to every base vertex attribute, which allows vertex insertions to propagate to all derived attributes.

In addition, the deletion of a vertex triggers:

- the deletion of all incident arcs, meaning that an update program contains a loop that invokes the arc deletion update program for each incident arc); and,
- the deletion of all child blobs, meaning that an update program contains a loop that invokes the blob deletion update program for each child blob.

For example, the vertex insert update program for the original sample view is:

```
proc InsertV {v} {
  global vmin0 vmax0 vmin1 vmax1 vmin2 vmax2
  grSet $v 1 0
  grSet $v 0 0
  grSet $v 2 [expr a($v,0) + a($v,1)]
  if { [expr a($v,0)] < $vmin0 } { set $vmin0 [expr a($v,0)] }
  if { [expr a($v,0)] > $vmax0 } { set $vmax0 [expr a($v,0)] }
  if { [expr a($v,1)] < $vmin1 } { set $vmin1 [expr a($v,1)] }
  if { [expr a($v,1)] > $vmax1 } { set $vmax1 [expr a($v,1)] }
  if { [expr a($v,2)] < $vmin2 } { set $vmin2 [expr a($v,2)] }
  if { [expr a($v,2)] > $vmax2 } { set $vmax2 [expr a($v,2)] }
}
```

Inserting or deleting an arc. The insertion or deletion of an arc affects all the derived attributes and sets that can be reached in the dependency graph from the vertex corresponding to the set of arcs `A`. The revised dependency graph for arc insertions and deletions contains an arc from `A` to every base arc attribute (which allows arc insertions to propagate to all derived attributes), and an arc to each of the set-valued vertex fields `OA` and `IA` (which allows arc insertions and deletions to propagate to the set of outgoing and the set of incoming arcs of the arc's head and tail).

Inserting or deleting a blob. The insertion or deletion of a blob affects all the derived attributes and sets that can be reached in the dependency graph from the vertex corresponding to the set of blobs `B`. The revised dependency graph for blob insertions and deletions contains an arc from `B` to every base blob attribute (which allows blob insertions to propagate to all derived attributes), and an arc to the set-valued vertex field `CB` (which allows blob insertions and deletions to propagate to the set of child blobs of the blob's parent vertex). In addition, the deletion of a blob causes all child vertices to be moved out of the blob, meaning that the blob deletion update program contains a loop that invokes the *remove vertex from blob* update program for each child vertex.

Adding or removing a vertex from a blob. The addition or removal of a vertex from a blob affects all the derived attributes and sets that can be reached in the dependency graph from the vertices corresponding to the blob `CV` and the vertex `pb` fields.

4.6 Simulating Existing Techniques

In this Section we show how our proposed approach can be used to generate different types of graph views. In our proposed model, dynamic FEVs can be characterized along two orthogonal dimensions:

- the expressive power requirements of the underlying view definition language, and
- the collection of required DM graphic properties.

The former dictates the kinds of similarity (distance) attributes that must be expressed, while the latter states how the derived attributes are to be mapped onto graphic (or more generally, presentation) properties. Table 4.1

View Category	*View Def'n*	*DM*
Dynamic Implicit FEV		
PLUM [88]	core	{rel pos}
Filtering FEV		
Generalized FEV of trees [34]	core, tc, sp	{}
Geometric Distorting FEV		
CATGraphs [60]	core	{rel/abs pos, rel/abs size}
Graphical FEV of Graphs [96, 97]	core	{rel/abs pos, rel/abs size}
Non-geometric Distorting FEV		
D-ABDUCTOR [74]	core, tc, sp	{rel size}
Layout-indep. FEV [78, 79]	core, tc	{rel size}
Treemaps [56, 112]	core, tc	{rel size}
Adorned FEV		
PLUM [88]	core	P - {pos,size}

Table 4.1: Characterization of Dynamic Fisheye Views. View definition language requirements refer to the core component of Dye (including focal points), transitive closure (tc), and shortest paths (sp). The "P" in the bottom right cell refers to the set of all available DM properties.

summarizes the requirements to simulate the 5 main categories of FEV techniques as defined in Section 3.2.2.

4.6.1 Dynamic Implicit FEVs

One of the very few systems that generated dynamic implicit FEVs was PLUM [88]. In this system, a data attribute could be mapped onto the node relative z position in a three dimensional display. As a consequence, nodes with high attribute values would automatically be brought to the front of the display and appear larger (due to the effect of point perspective), while nodes with low attribute values would be placed in the back of the display and appear small. In principle, any data attribute can be so mapped; in PLUM nodes had a priority attribute, but its values were precomputed. In Graphite, we could use the Dye view definition language to first define any number of similarity attributes and then map any one of these onto the relative z position property. The current implementation of Graphite, however, is limited to two dimensional drawings – mapping onto the relative z position thus simply changes the relative stacking order of nodes in the display, placing nodes with high attribute values on top. One example of this is given in Figure 5.18 on page 82, in which the *Elevation* attribute is mapped onto relative z position causing nodes associated with cities at higher elevations to be placed on top of nodes representing cities at lower elevations.

4.6.2 Filtering FEVs

In principle, a filtering FEV can be obtained by first defining any number of similarity attributes, and then using DQ to mask elements with low similarity attribute values.

A more specific notion of filtering FEV is Furnas' FEV of trees [34], in which the notion of similarity is given by:

$$DOI(p, f) = -1 \cdot (Depth(p) + Dist(p, f))$$

where $Depth(p)$ is the depth of point p in the tree, and $Dist(p, f)$ is the undirected path distance between p and (focal point) f in the tree. Thus, the degree of interest in point p (or similarity between p and f) increases with decreasing depth and decreases with increasing distance. An nth order FEV is obtained by only displaying points with $DOI(p) \geq -(Dist(p, root) + 2 \cdot n)$.

Figure 5.26 on page 90 shows a directory tree visualization. Figure 5.29 on page 92 shows a first-order filtering FEV, while Figure 5.30 on page 93 shows a second-order filtering fisheye view. Notice that the right-most histogram bar contains nodes in the first-order FEV, the next bar contains nodes in the second-order FEV, and so on. Thus to produce an nth order FEV, the user must simply mask all histogram bars to the left of the nth bar (which can be accomplished with a single mouse click).

4.6.3 Geometric Distorting FEVs

Distorting FEVs are obtained by distorting the size and/or the position of nodes based on a geometric notion of distance: nodes that are geometrically distant from the focal point(s) are compressed, while those that are not, are magnified. Scaling can be uniform (equal magnification in all dimensions) or non-uniform (independent magnification in each dimension) which correspond to what we've termed polar and orthogonal distorting FEVs respectively.

In the following examples, we assume that the original node positions are obtained through dynamic mapping using two (or three) node attributes. In particular, as geometric distorting FEVs are primarily used to distort geographical maps, original node positions are typically obtained by mapping longitude and latitude data. This is precisely the case in the Geographic Information System application shown in Figure 5.18 on page 82.

To generate a compressed arctangent graph [60] (orthogonal distorting FEV), we must define a view in which we derive attributes whose values correspond to the transformed x and y coordinates and the distorted sizes, and then map these attributes onto the relative x and y position and relative x and y size properties. The result is shown in Figure 5.24 on page 88 as is the associated Dye view definition.

The same approach can be used to generate both graphical fisheye views [96, 97] and multi-viewpoint perspective techniques [75, 108]. A graphical fisheye view thus obtained is shown in Figure 5.25 on page 88.

Several points require further elaboration.

First, note that by employing relative size properties, the above approach does not compute the exact distorted sizes. To do so, the DM system would have to provide absolute position and size properties. The exact distorted width and height could then be computed by transforming the top left and bottom right corners of each node, and computing the differences of the right and left x coordinates and the bottom and top y coordinates respectively.

Second, note that to generate polar distorting fisheye views, it suffices to derive Euclidean proximity between nodes and to map this derived attribute onto relative size (which scales uniformly in x and y).

4.6.4 Non-geometric Distorting FEVs

Non-geometric distorting FEVs are obtained by uniformly scaling the sizes of nodes based on the value of a non-geometric similarity attribute. These techniques are especially useful in distorting nested graph representations. Figure 5.31 on page 94 shows a nested representation of a Unix file directory hierarchy.

Several notions of similarity for this application are defined on page 89. The first is identical to the notion of similarity used in filtering fisheye views and similar to the notion of priority in the Abridgement facility of the D-ABDUCTOR system [74]. The resulting non-geometric distorting FEV is shown in Figure 5.32 on page 95. The second notion of similarity is defined in terms of the proximity in the nesting hierarchy and is scaled by focal point magnification; the result is shown in Figure 5.33 on page 96.

These examples demonstrate the key idea of the layout-independent fisheye view technique which we described in [79]. The view definition on page 93 captures a simple variant of this technique, and Figure 5.34 on page 97 shows the resulting distorted visualization containing two focal points.

4.6.5 Adorned FEVs

Adorned FEVs are obtained by mapping similarity attributes onto non-geometric presentation properties such as colour, line type and width, audio (pitch), degree of blurring, and so on. The PLUM system [88], for example, permitted a predefined priority attribute to be mapped onto any presentation property. Although some network visualization systems [8, 26] permitted data attributes to be mapped onto presentation properties, they did not possess the notion of focal points or similarity.

Figure 5.1 on page 66 shows an airline transportation network with focal points *Toronto* and *London*. In this visualization, the *Trip Duration* similarity attribute is mapped onto the *percentage fill* node property, causing nodes that are temporally far from the selected focal points to appear filled – in a sense emphasizing the notion of temporal *dissimilarity*.

In Figure 5.15 on page 79 and Figure 5.20 on page 84 the similarity attributes are mapped onto the node colour saturation property, which visually emphasizes nodes that are similar to the selected focal point(s).

4.6.6 Treemaps

The Dye view definition language can also simulate a radically different hierarchical data visualization technique: the Treemap [56, 112]. The view definition is given on page 94, while Figures 5.35 and 5.36 show two resulting Treemap visualizations. An interesting aspect of the Treemap simulation is the use of layout algorithms to position nested nodes. By exploiting simple existing layout strategies (one dimensional layout) to position nested nodes, we can off-load this task to the layout engine and thereby simplify the view definition.

Chapter 5

Sample Applications

5.1 Transportation Networks – Airline Flights

The airline flights visualization application described in this Section is a specific instance of the class of *transportation network* visualization applications. As such, the visualization techniques outlined below can be applied to most transportation network problems.

The base vertex relation in the airline flights database is *City*:

```
City(CityId,Name,
  0:Longitude,          % degrees longitude
  1:Latitude            % degrees latitude
)
```

The base arc relation in the airline flights database is *Flight*:

```
Flight(Number,FromCity,ToCity,
  0:Price,              % price in dollars of one-way ticket
  1:Duration            % duration in minutes of one-way trip
)
```

Although this basic set of attributes could be used in a dynamic query visualization system, it provides very limited means for exploring the underlying data. In particular, the route information is not explicitly stored in the base relations, but must be computed by evaluating APSP. We have constructed the following view on the database.

```
P0 := APSP(V,a:A,undirected,30,a[1])             % quickest routes
P1 := APSP(V,a:A,undirected,25,a[0])             % cheapest routes
P2 := APSP(V,A,undirected,0,1)                   % most-direct routes

v:V {
  v[2] := CNT(UNION(IA(v),OA(v)))                % busy destination
  v[3] := AVG(a:UNION(IA(v),OA(v)),a[2])         % expensive destination
  v[4] := AVG(f:VF,hypot(f(0) - v(0),f(1) - v(1))) % average trip distance
  v[5] := AVG(f:VF,SP(P0,v,f))                   % average trip duration
  v[6] := AVG(f:VF,SP(P1,v,f))                   % average trip cost
  v[7] := AVG(f:VF,SP(P2,v,f))                   % average trip legs
  v[8] := (NPATHS(P0) > 0 ? 1 : 0) +             % trip cities mask
          (NPATHS(P1) > 0 ? 2 : 0) +
          (NPATHS(P2) > 0 ? 4 : 0)
}
a:A {
  a[2] := a[0] / a[1]                            % price/time ratio
  a[3] := (NPATHS(P0) > 0 ? 1 : 0) +             % trip flights mask
          (NPATHS(P1) > 0 ? 2 : 0) +
          (NPATHS(P2) > 0 ? 4 : 0)
}
```

Notice that the `v[8]` and `a[3]` attributes are 3 bit masks that permit the user to select cities or flights that partake in any combination of currently selected fastest, cheapest, or most direct routes.

Figure 5.1 shows a screen shot in which *Toronto* and *London* are focal points, *Honolulu* is a path source and *Reykjavik* and *Bergen* are path sinks. The *TripDuration* attribute has been mapped onto the *percentage fill* node property causing nodes that are temporally far from the focal points to appear filled, and the *TripFlights* attribute has been mapped onto link *width* property causing flights that take part in the most direct, cheapest, and quickest routes between the selected end-points to appear thicker. The nodes were positioned by first mapping the *Longitude* and *Latitude* attributes onto the relative x and y position, and then employing a force-based layout algorithm to spread out the nodes and thereby eliminate overlaps while maintaining the cities' approximate geographic locations.

Figure 5.2 demonstrates how interactive filtering can be used to generate informative visualizations. The flights visualization is interactively filtered on the *TripDuration* attribute with focal points *Toronto* and *London*, which can be used to quickly determine which destinations are temporally close to both Toronto and London. Notice how masking through a similarity attribute can emphasize *islands of relevance* centered on the current set of focal points.

Figure 5.3 shows the flights database interactively filtered on the *BusyDestin* and *TripCost* attributes. The answer shows busy destinations that are relatively inexpensive to reach from Toronto or London.

Figure 5.4 shows the flights database interactively filtered on the *TripCities* dynamic attribute – the set of visible nodes are those that appear on any fastest route between the selected sources and sinks.

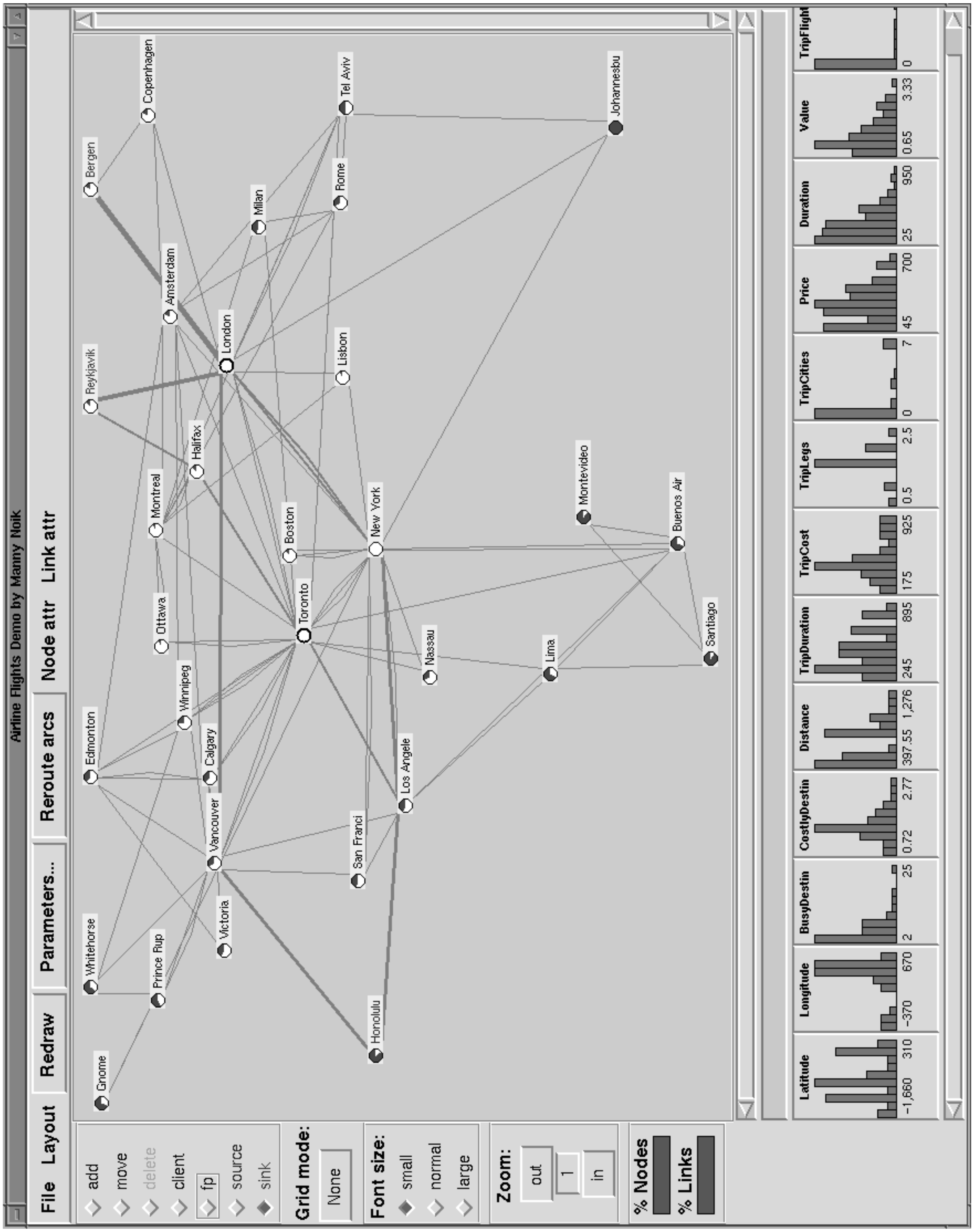


Figure 5.1: The airline flights application.

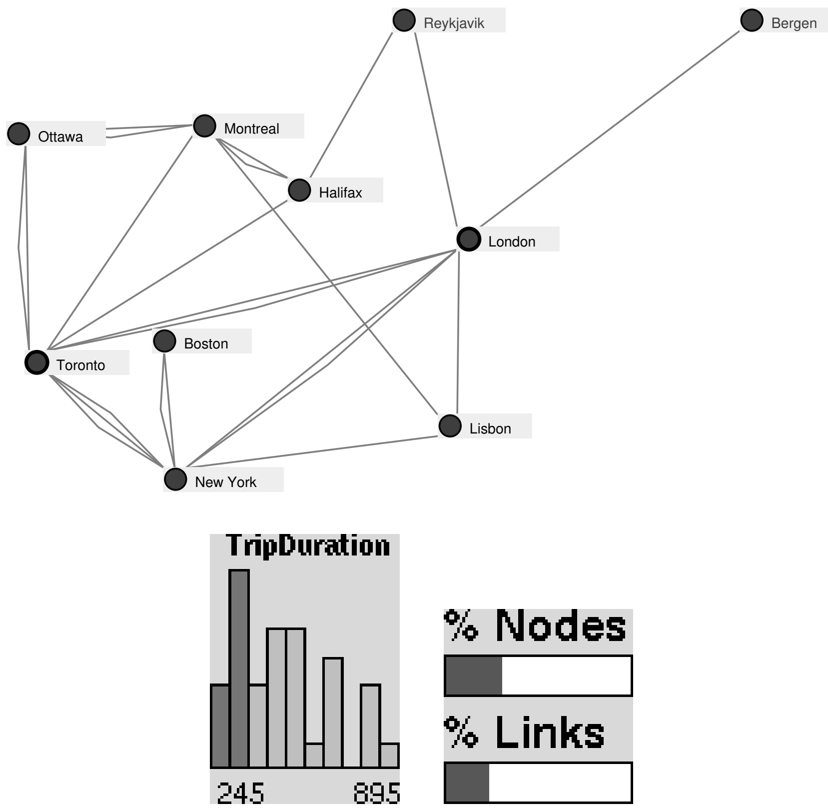


Figure 5.2: Finding temporally close destinations; this is accomplished by masking with the *TripDuration* similarity attribute.

5.2 Web Structure and Navigation

Browsing the World-Wide-Web introduces the same sorts of difficulties as browsing more traditional forms of hypermedia, and these problems are often further exacerbated by the sheer size of the Web and the lack of specialized navigation tools. One of the oldest documented user-interface problems in hypertext systems is the *lost-in-hyperspace* phenomenon [77]. As a partial solution, many have proposed a variety of overview diagrams [43, 69, 77, 117], and in earlier work, we showed how the layout-independent fisheye view approach [78, 79] can be applied to the visualization of nested hypertext networks.

In the following set of examples, we show a number of screen shots from the Multisurf prototype [47] – a system that combines local (database) and global (Web) searching with the Queries-R-Links graphical markup of boolean queries technique [39], and several graph visualizations realized with the Graphite system.

Figure 5.5 shows a screen shot of the Multisurf Web structure/navigation graph (Multisurf uses two other graphs which we do not discuss here). The nodes in the graph represent documents and the links hypertext links among the documents. More specifically, the base database has the following schema:

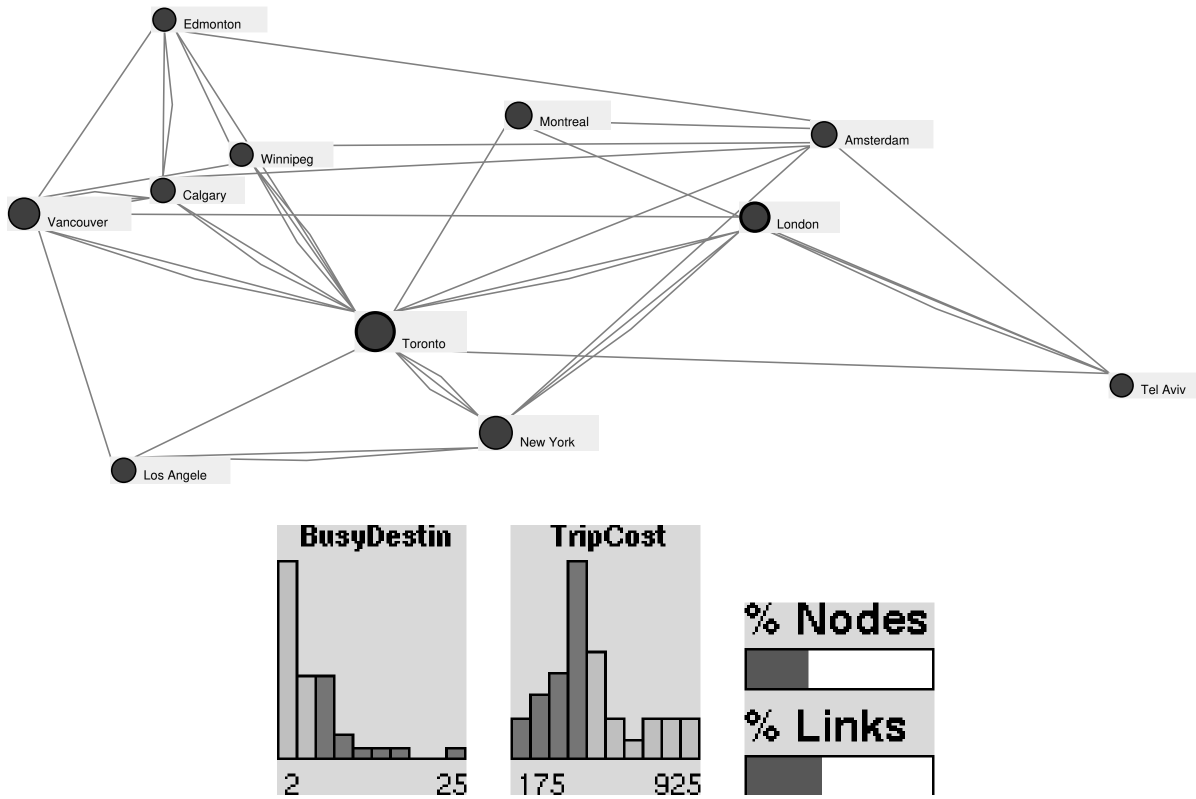


Figure 5.3: Finding busy destinations that are not expensive to reach; this is accomplished by masking with the *BusyDestin* attribute and the *TripCost* similarity attribute.

```
Document(Id,Name,
 0:Size,            % size in bytes
 1:Type,            % 0-http 1-ftp 2-mail 3-file 4-gopher 5-telnet 6-other
 2:AddTS,           % add logical time stamp
 3:BrowseTS         % browse logical time stamp
)
Link(Id,SourceDocument,TargetDocument,
 0:AddTS,           % add logical time stamp
 1:BrowseTS         % browse logical time stamp
)
```

We have defined the following view that is maintained dynamically as documents and links are inserted into the database during a Web browsing session:

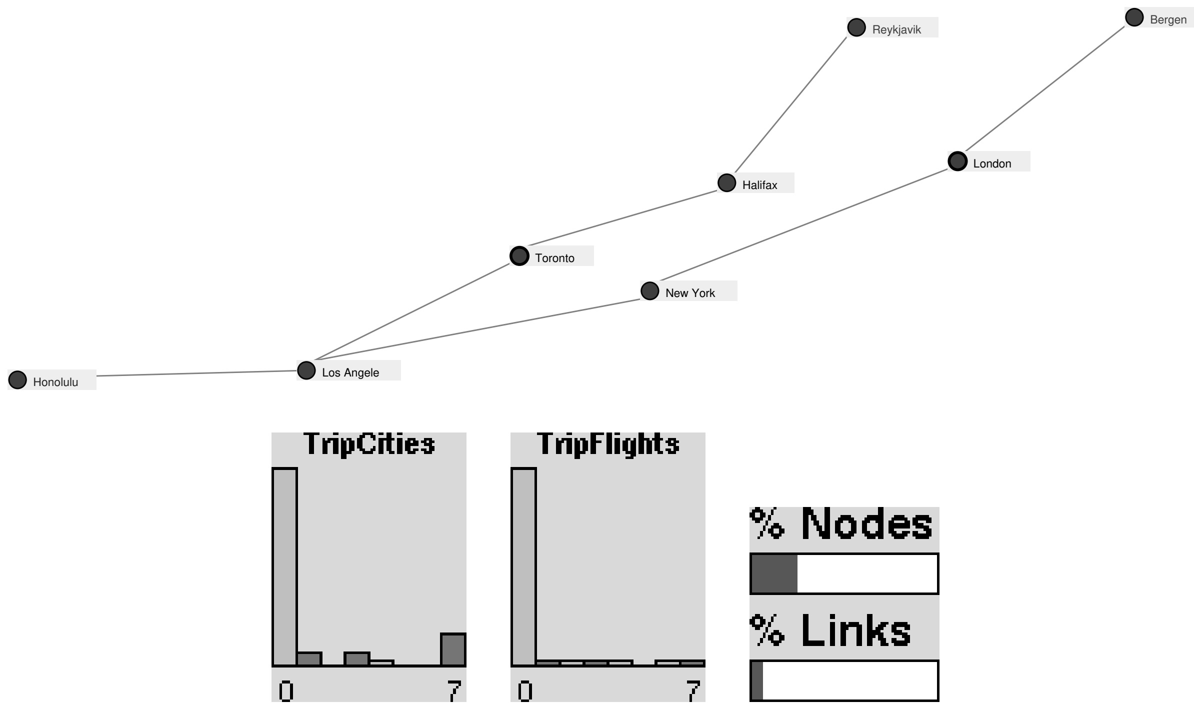


Figure 5.4: Emphasizing fastest routes between selected cities.

```
v:V {
  v[4] := CNT(IA(v))           % document in-degree
  v[5] := CNT(OA(v))           % document out-degree
  v[6] := AVG(f:VF,-1 * abs(v[4]-f[4]) / mag(f))
                               % document add neighbourhood
  v[7] := AVG(f:VF,-1 * abs(v[5]-f[5]) / mag(f))
                               % document browse neighbourhood
  v[8] := REGEXP()             % document regular expression match
}
a:A a[2] := REGEXP()           % link regular expression match
```

The *node browse time stamp* attribute has been mapped onto node colour hue and saturation graphical properties and the *link browse time stamp* has been mapped onto link width and colour. The *node add time stamp* attribute has been mapped onto node relative x position.

Figure 5.6 shows a poor Web visualization which is instructive to point out the weakness of algorithm-only approaches. Clearly, attempting to draw even a relatively small graph results in a very poor drawing.

Figure 5.7 shows how with just a few interactions, the user can turn a mess into an informative display. This is accomplished in several steps (performed only once):

1. dynamically map the *Node Add Time Stamp* onto the horizontal axis;
2. spread the nodes out vertically by use of one-dimensional spring layout algorithm;
3. mask the links with a zero *Link Add Time Stamp* by clicking on the associated histogram bar;

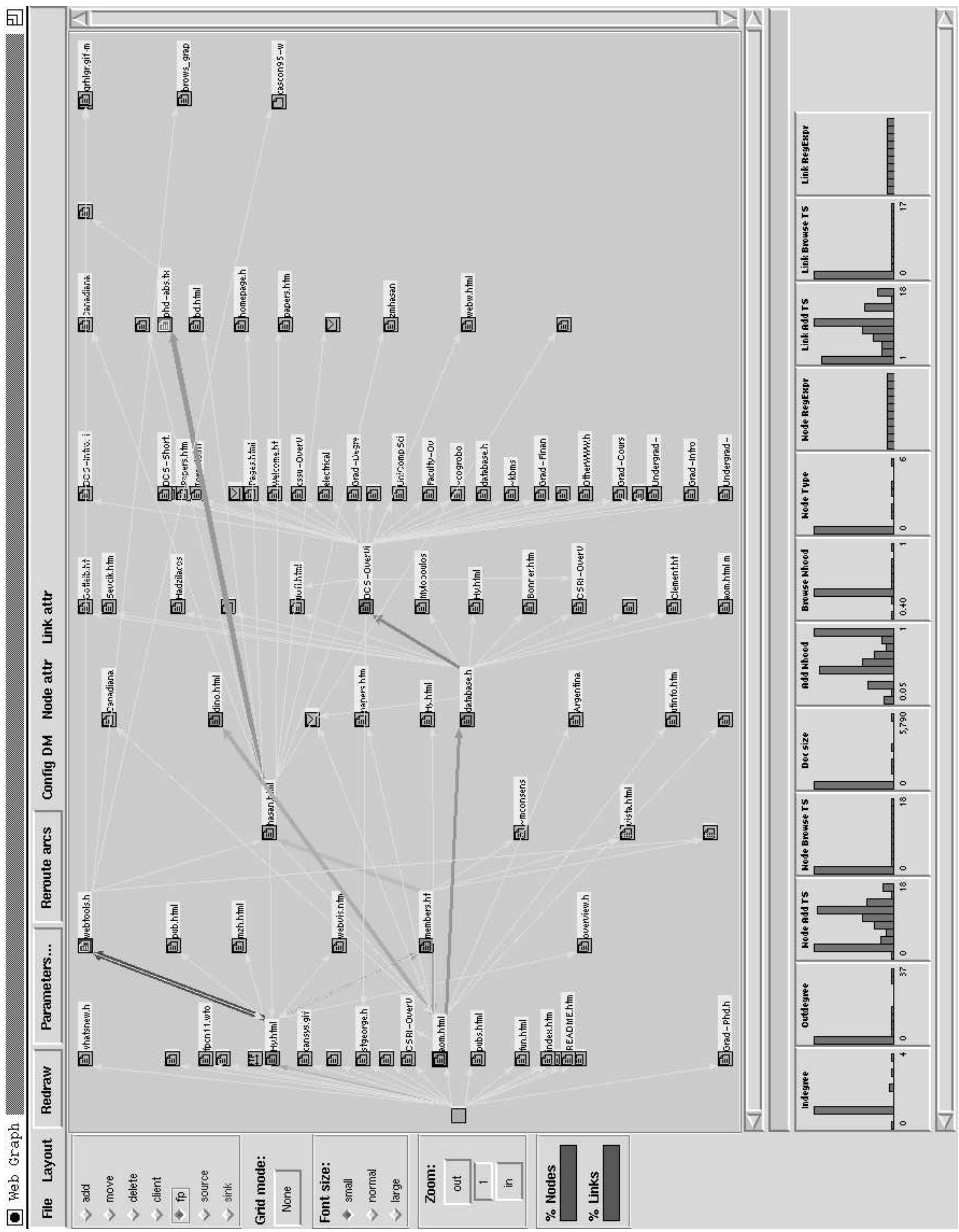


Figure 5.5: Web structure/navigation visualization.

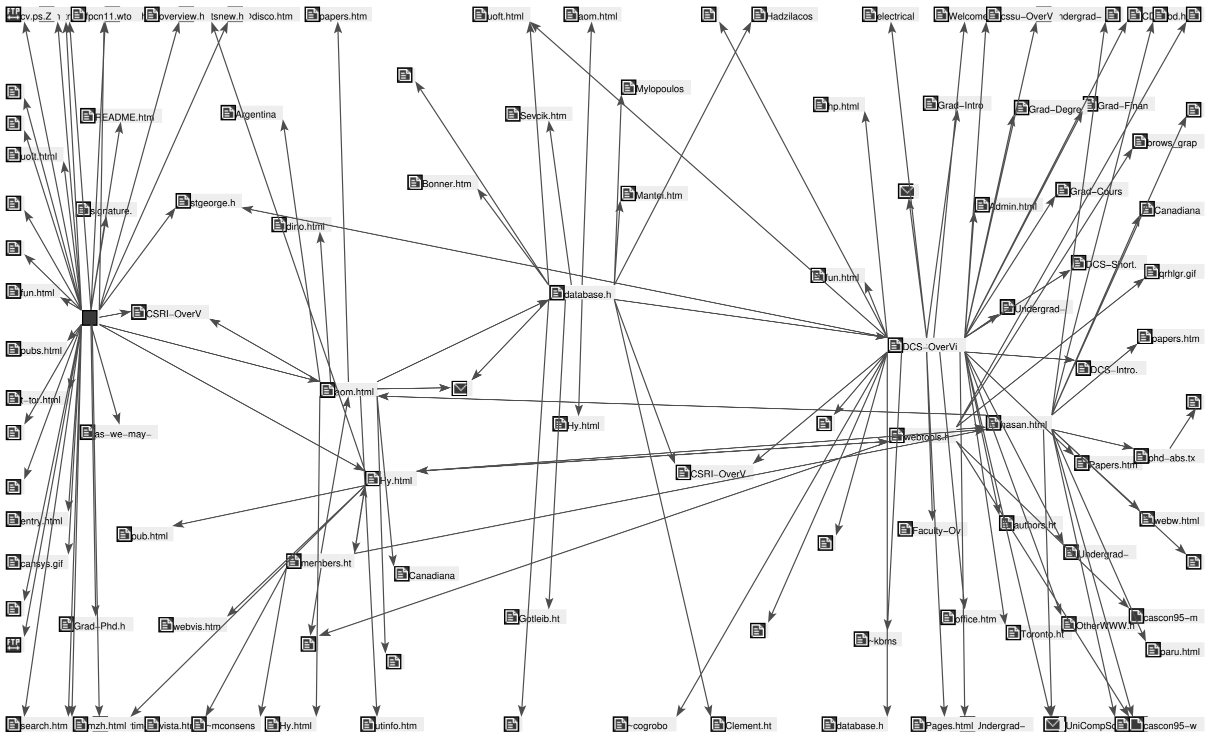


Figure 5.6: A poor Web visualization.

4. dynamically map *Link Browse Time Stamp* onto spectrum colour and link width;

5. dynamically map *Node Browse Time Stamp* onto colour saturation (red is default hue);

Figure 5.8 shows a graphical Web navigation history visualization which is obtained from the previous display by masking the nodes with a zero *Node Browse Time Stamp* and then redrawing the resulting subgraph with the hierarchical left-to-right algorithm. Unlike popular Web browsers (*e.g.,* Netscape Navigator, Mosaic) which only maintain a single current history path, the graphical history view shows the entire structure of the navigation history.

Figure 5.9 shows a variation on this idea. Here, the *Node Browse Time Stamp* is dynamically mapped on the horizontal axis and the nodes are spread out vertically with the one-dimensional spring layout algorithm. This arrangement makes it easy to see the order in which documents were visited during the browsing session.

Figure 5.10 shows the use of the *Browse Nhood* attribute to identify documents that were browsed at roughly the same time as the documents corresponding to selected focal points. In particular, this feature is useful when the node *BrowseTS* attribute is not mapped onto one of the axes.

Figure 5.11 shows the use of the node *Add Nhood* attribute to identify documents that were added to the graph at roughly the same time (corresponding to the first appearance of an anchor or hypertext link to a document). Similarly, this feature is useful when the *Node AddTS* attribute is not mapped onto one of the axes.

Figure 5.12 shows the use of the *Outdegree* attribute to identify comprehensive documents – documents

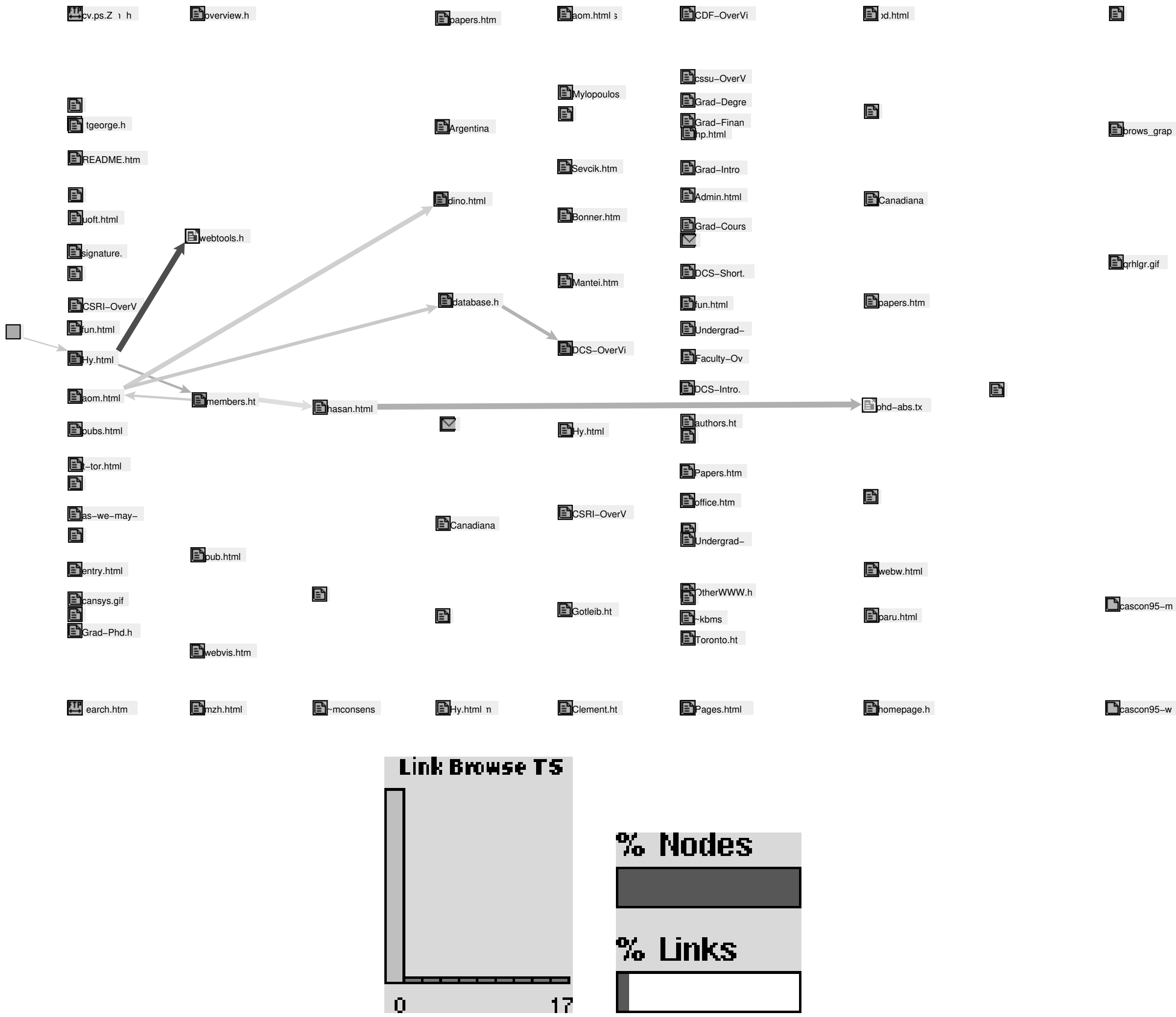


Figure 5.7: A better Web visualization.

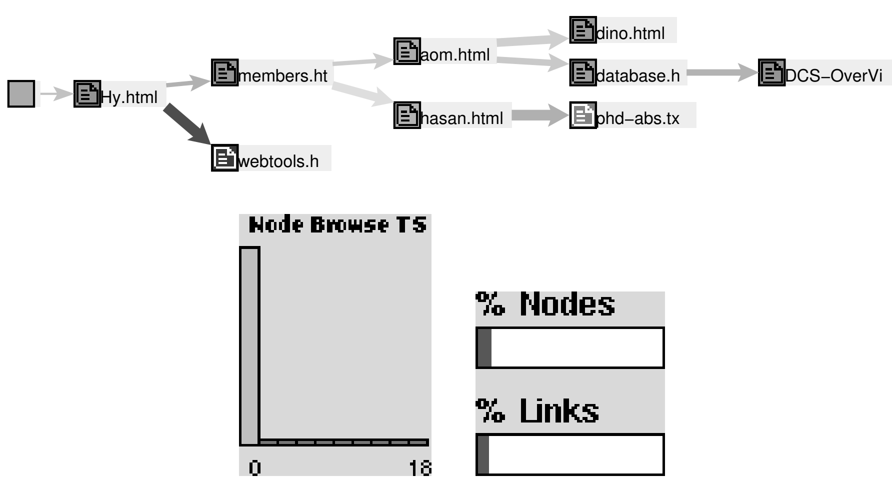


Figure 5.8: Graphical Web navigation history.

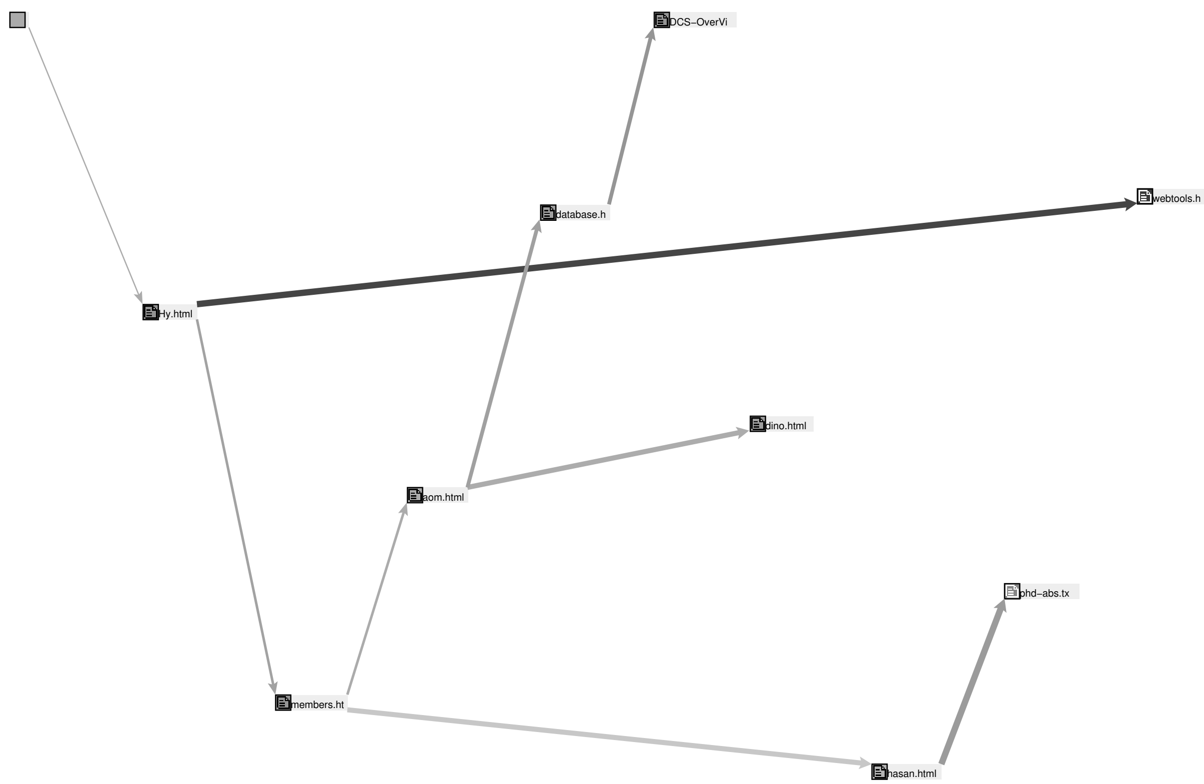


Figure 5.9: An alternative navigation history view.

that contain many hypertext links, while Figure 5.13 shows the use of the *Indegree* attribute to identify popular documents – documents that are pointed to by many hypertext links.

Note that the Graphite system could be easily customized to visualize Web server log statistics, in a fashion similar to WebViz [84]. In addition to posing dynamic queries in terms of existing attributes, the ability to define similarity attributes would enable users of the Graphite version to quickly identify documents with similar usage patterns.

5.3 Genome Map Structure Graphs

One of the main goals of the Human Genome Project is to construct detailed physical maps of the human genome. A physical map is an assignment of DNA fragments to their locations on the genome.

There have been a number of different approaches to solving this problem, including the use of logic programming and graph visualization with the Hy+ tool [45]. In this work, Hy+ was used to assemble contigs (partial maps) from real and simulated STS (Sequence Tagged Site) content mapping data with little programming effort. An STS probe is a very short piece of DNA that "sticks" (or hits) to a particular site on the chromosome. Since probes cannot be assigned exact locations, one way to determine their order is to determine which pairs of

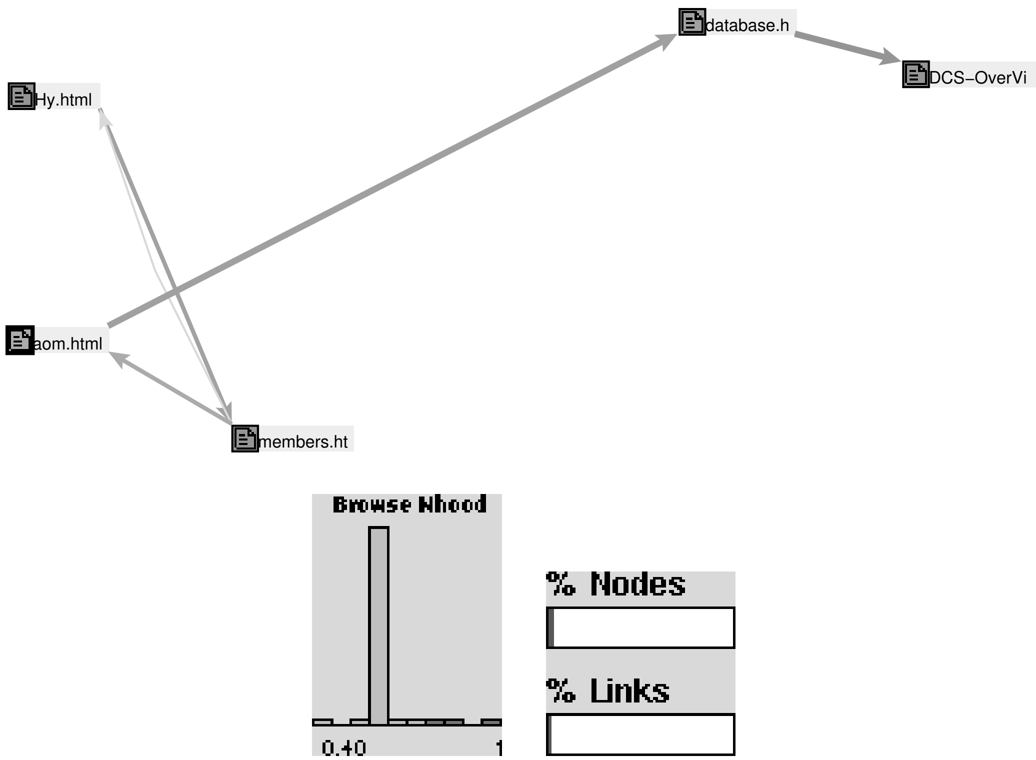


Figure 5.10: Document "browse neighbourhoods".

probes are "close" together on the chromosome. This is accomplished by randomly cutting many copies of the chromosome into small fragments called YACs (Yeast Artificial Chromosomes), and looking for pairs of probes that hit the same YAC. By finding such pairs of probes, it is possible to construct an STS adjacency graph [45], in which each node represents an STS probe, and a link between two nodes means that the two probes are adjacent. In a perfect world one would expect to generate a line graph (a graph in which the vertices lie on a single path) that would give the exact ordering of probes along the length of the chromosome, however, various forms of noise greatly complicate the map assembly problem.

Recent work has attempted to overcome two problems in map assembly: integration of overlap data, and detection of errors and anomalies in data [46]. Integration is achieved through the notion of a *cluster*, a maximal set of mutually overlapping fragments, while error detection is achieved by extracting and displaying the underlying structure of cluster or STS adjacency graphs. Thus in such a structure graph, a node represents a collection of STS probes that are found at approximately the same location in the STS adjacency graph, where two nodes are connected by a link if the corresponding probes hit at least two common YACs.

The graph algorithms used to convert adjacency graphs into structure graphs outlined in [46] have been applied to data from the CEPH YAC library [20] and data from the Human Physical Mapping Project at the Whitehead Institute/MIT Genome Center [53].

We have used Graphite to visualize genome map structure graphs.

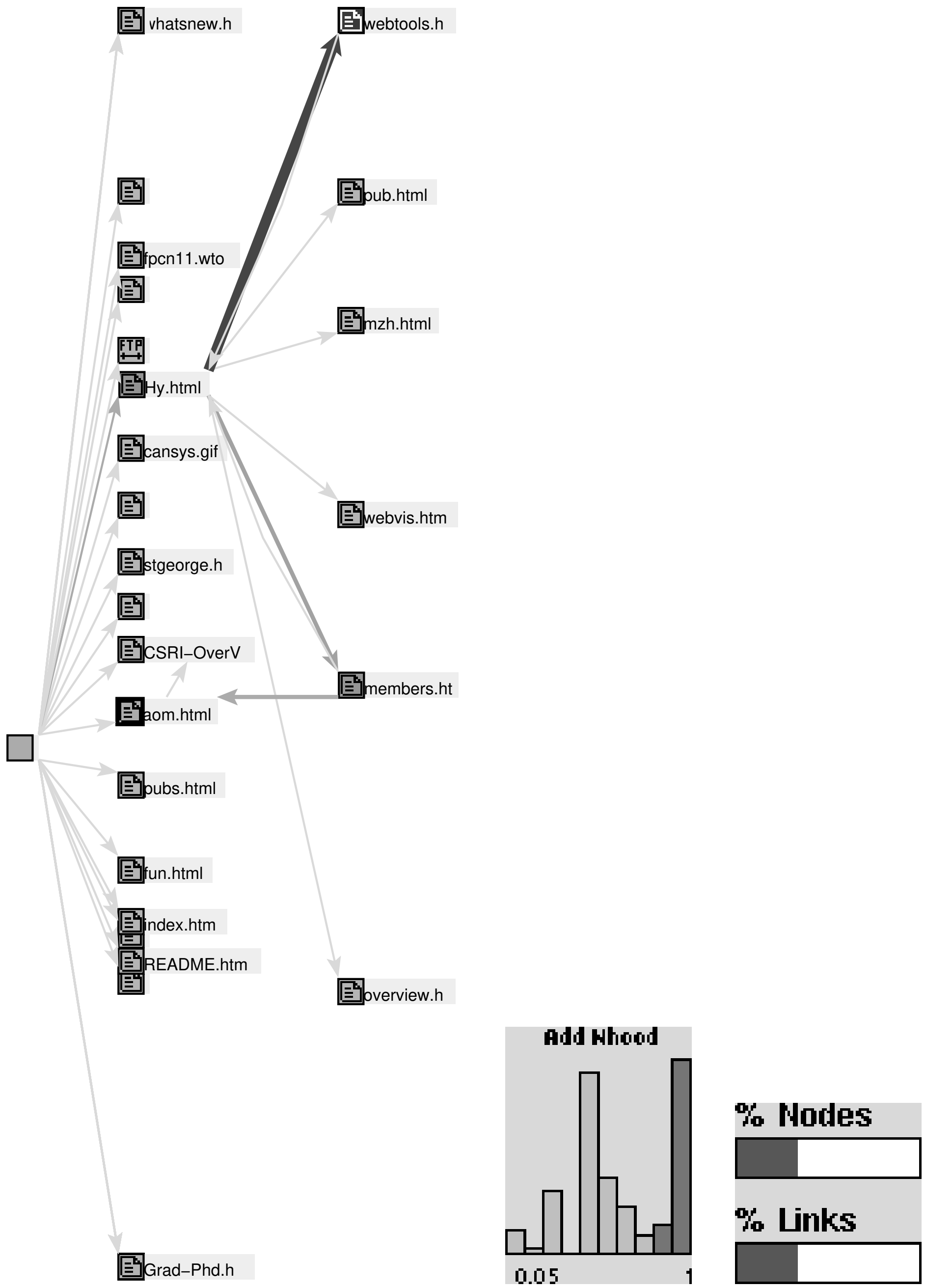


Figure 5.11: Document "add neighbourhoods".

The base vertex relation in the genome map database is *Cluster*:

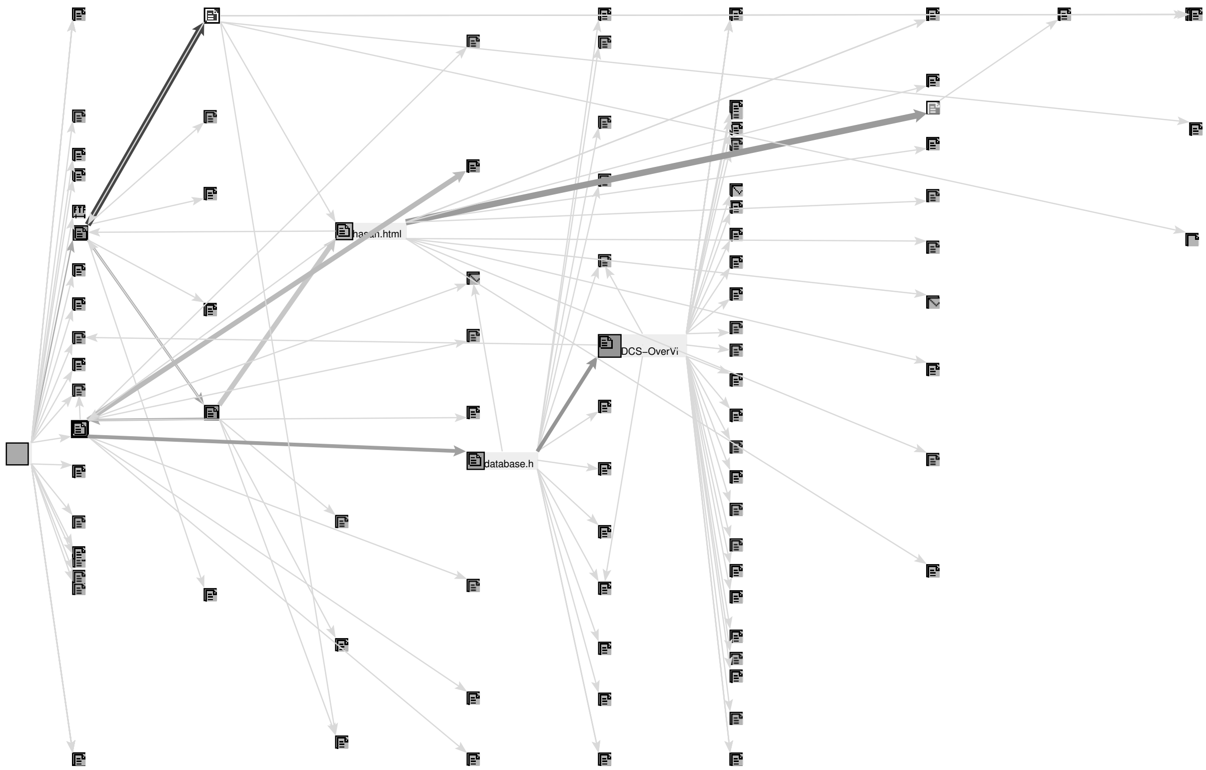


Figure 5.12: Emphasizing "comprehensive" documents.

```
Cluster(Id,Name,
  0:NoProbes,          % number of STS probes in cluster
  1:NoAssigned,        % number of probes that have been assigned
                         to a chromosome
  2:NoHits,            % number of hits by all probes in cluster
  3:NoYACs,            % number of YACs hit by probes in cluster
  4:NoChroms,          % number of chromosomes to which probes are assigned
  5:Chromosome,        % most common chromosome in cluster
  6:ProbesOnChrom,     % number of probes on most common chromosome
  7:RadHybridPos,      % radioactive hybrid position
  8:GeneticPos         % genetic mapping position
)
```

The base arc relation in the genome map database is *Connection*:

```
Connection(Id,Label,FromCluster,ToCluster,
 0:NoYACs              % number of YACs between two clusters
)
```

We have defined the following view on the database:

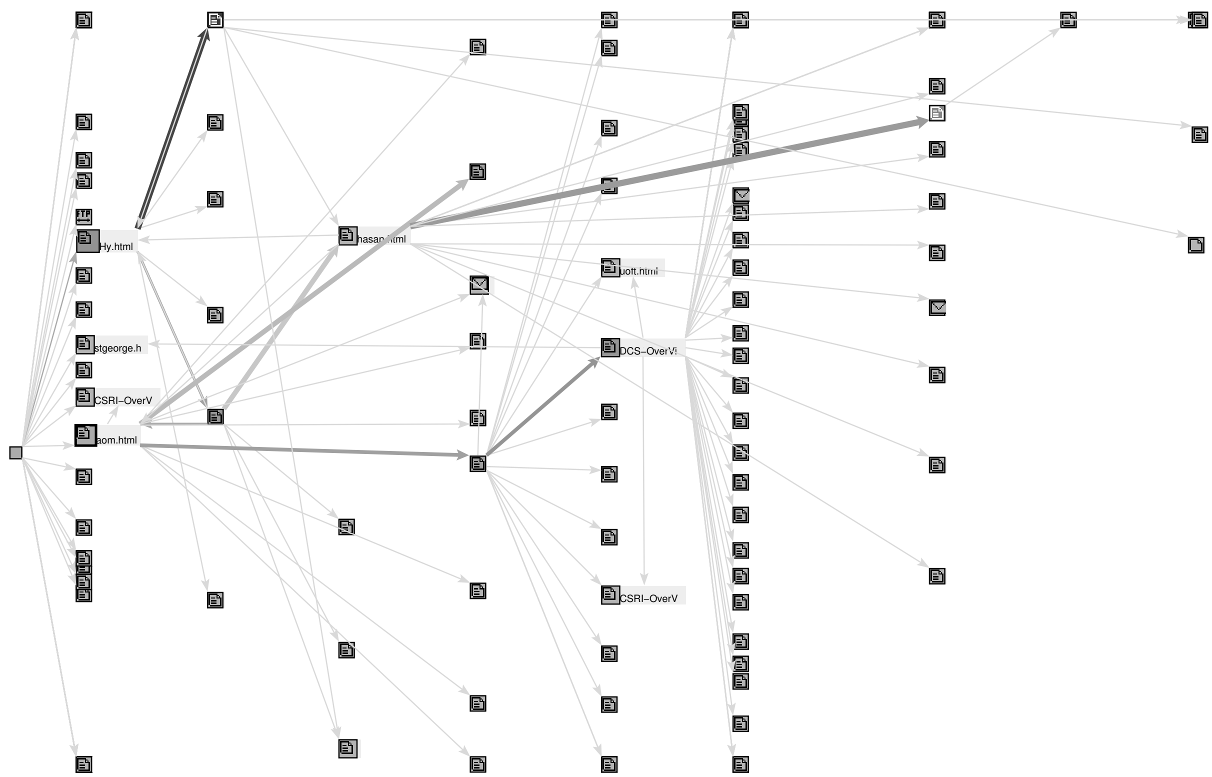


Figure 5.13: Emphasizing "popular" documents.

```
P := APSP(V,A,undirected,0,1)          % shortest paths

v:V {
  v[9]  := (v[7] > 0 ? 1 : 0) +        % Positions: a bitmask of avail. pos. data
           (v[8] > 0 ? 2 : 0)
  v[10] := v[6] / v[0]                 % Purity: ratio of probes on common chrom
  v[11] := v[4] != 0 ? v[0] / v[4] : 0 % Evidence: avg no of probes per chrom
  v[12] := v[1] / v[0]                 % Coverage: percentage of assigned probes
  v[13] := CNT(UNION(IA(v),OA(v)))     % Degree: vertex degree
  v[14] := USERSIM({0-4,5:o,6-13})     % user-configurable similarity
  v[15] := NPATHS(P) > 0               % PathNodes: nodes on shortest paths

a:A a[1]  := NPATHS(P) > 0             % PathLinks: links on shortest paths
```

Figure 5.14 shows a screen shot of the visualization of the genome map database. In this instance, the *Chromosome* attribute has been mapped onto the node color *hue* and *saturation* properties, and the *NoProbes* attribute has been mapped on the node *relative size* property. Nodes have been positioned by the hierarchical (left-to-right) layout algorithm. Notice how the visualization helps to immediately see groups of similarly coloured nodes that correspond to clusters that have been assigned to a common chromosome.

Figure 5.15 shows a view in which the notion of similarity is that of being assigned to the same chromosome; similarity is mapped onto the node colour *saturation* property, allowing nodes that are similar to the selected focal

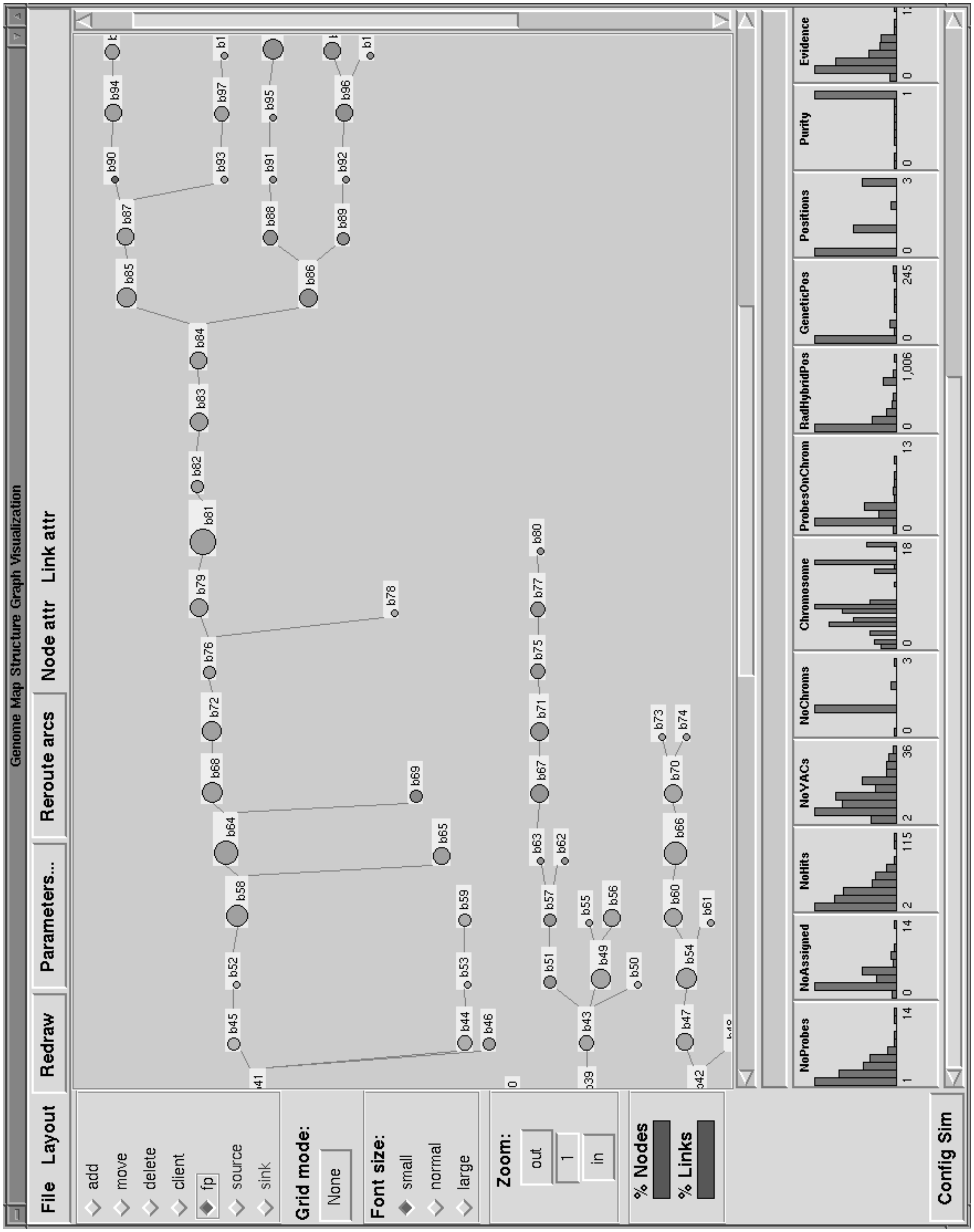


Figure 5.14: Genome Map structure graph visualization application.

point(s) to be easily identified.

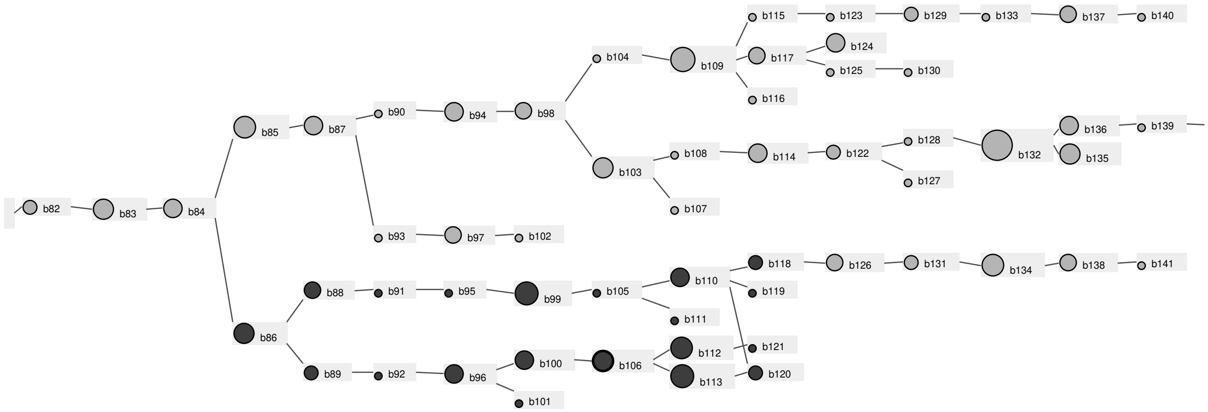


Figure 5.15: Identifying clusters assigned to selected chromosomes.

Figure 5.16 takes the preceding example one step further. Here the notion of similarity is the same – the state of being assigned to the same chromosome as at least one of the focal points, and similarity is still mapped onto the node colour *saturation* property. In addition, the link *NoYACs* attribute has been mapped onto link thickness and link colour *saturation*, and the node *RadHybridPos* attribute has been mapped onto the node label property. Thus by selecting a focal point (the node labelled "709" in the middle of the Figure), the user can identify other nodes assigned to the same chromosome, have the system emphasize these through colour saturation, and then analyze potential paths by examining the position values. In this application, users are trying to identify paths along which nodes have uniformly increasing or decreasing positions. The thickness and saturation of links along such paths indicates the degree of evidence in favour of such a path and help to visually identify potential paths. In the Figure, we see strong evidence for a path from node at position "691" in the north-west quadrant of the diagram, to node at position "716" in the south-east quadrant.

The user can check whether the genetic position information also supports the existence of this path by mapping the *GeneticPos* attribute onto the node label property; the visualization can be simplified by temporarily masking all dissimilar nodes and redrawing the remaining subset of nodes as shown in Figure 5.17. As can be seen from the Figure, in this instance, the genetic position data does in fact support the radioactive hybrid position data.

5.4 Geographic Information Systems

Geographic Information Systems (GIS) are computer software applications that are intended to help users to maintain and understand data relating to a wide variety of geographic phenomena, such as weather patterns, polution, crop and wildlife distribution, natural resource management, and so on.

We have a very simple GIS database – in fact it only contains one relation: *City*:

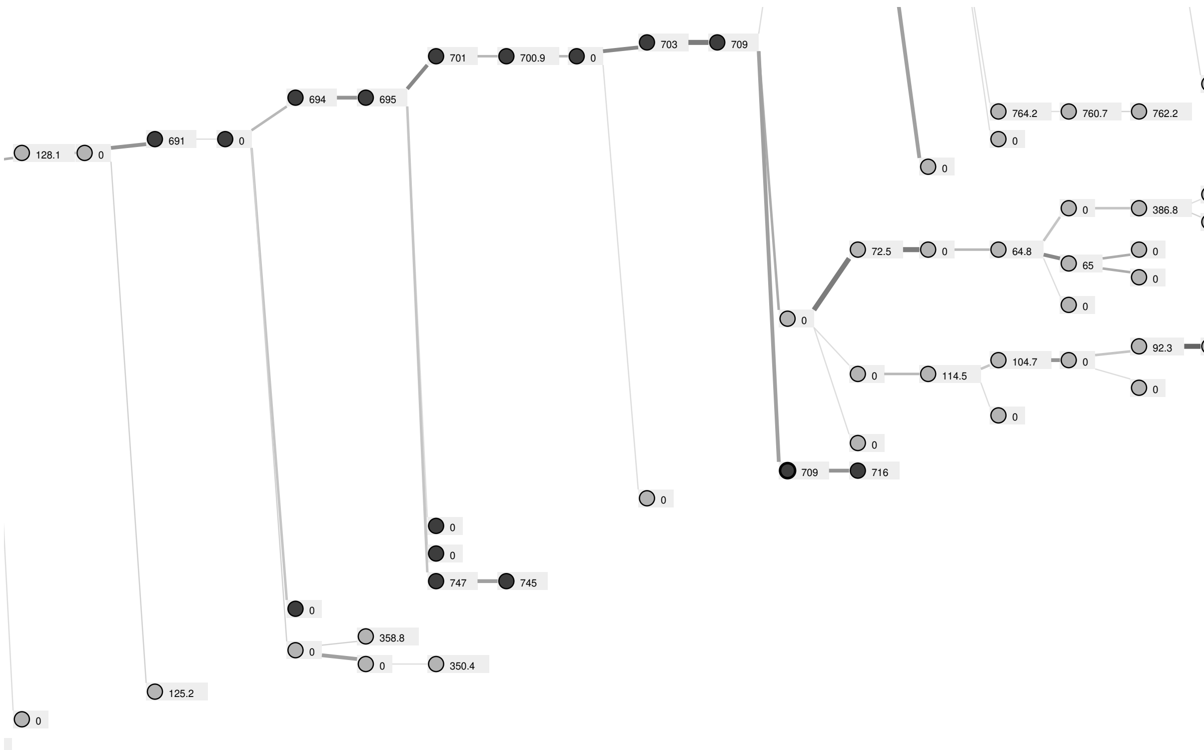


Figure 5.16: Analyzing paths among clusters on selected chromosomes.

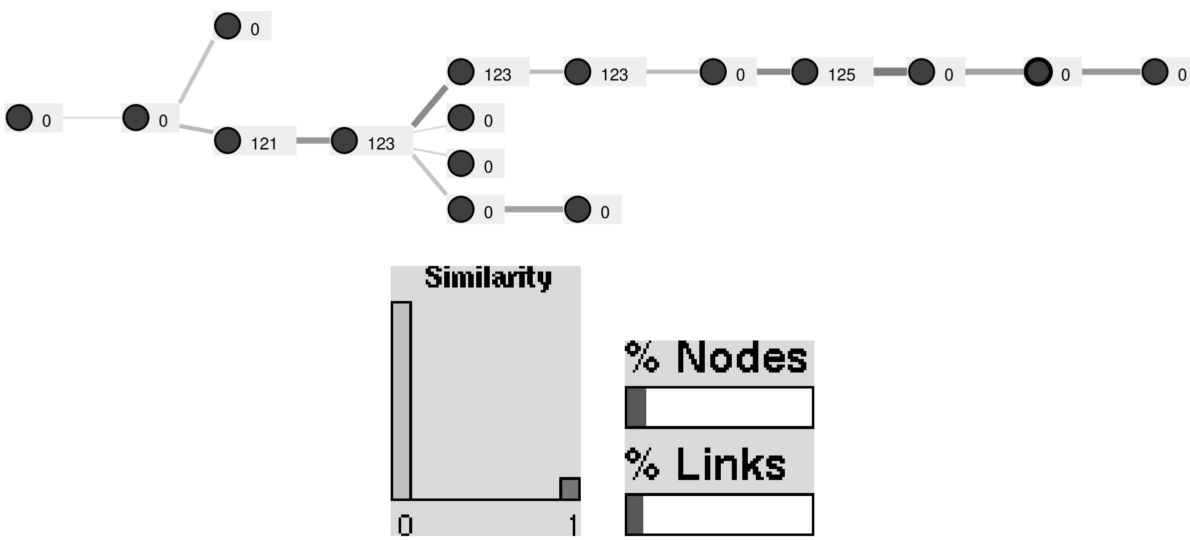


Figure 5.17: Verifying candidate path with genetic position data.

```
City(Id,Name,
 0:State,              % state number in [0,54]
 1:Longitude,          % no. degrees longitude
 2:Latitude,           % no. degrees latitude
 3:Elevation,          % in feet above sea level
 4:Population          % in thousands, based on 1980 census
)
```

We have created the following view on the database:

```
v:V v[5] := USERSIM({0:o,1-4})   % user-defined similarity
```

Figure 5.18 contains a screen shot of our GIS sample application. In the visualization *Longitude* is bound to relative x, *Latitude* is bound to relative y, *Elevation* is bound to relative z (stacking order) and color saturation, and *Population* is bound to relative size.

Figure 5.19 shows how a user can define the notion of similarity to be Euclidean distance between cities ($(dx^2 + dy^2)^{\frac{1}{2}}$), and then use this attribute to mask cities that are far away from Atlanta.

Figure 5.20 shows how a user can define the notion of similarity to be that of *same-state* as any selected focal point, and then map this attribute onto the node color saturation property to quickly emphasize cities that belong to the same state as the selected focal points. Naturally, this attribute could also be used to mask cities that do not belong to the same state.

Figure 5.21 shows how a user can define the notion of similarity to be *1 part elevation and 2 parts population*, and then identify cities that are similar to Wichita (this notion of similarity says that having similar population size is twice as important as having similar elevation above sea level).

Figure 5.22 shows a zoomed in view of the Los Angeles area, while Figure 5.22 shows the same area, however, the *Elevation* attribute has been mapped onto the node label property. The former demonstrates Graphite's ability to zoom in and out of two-dimensional scatterplots as done in the Starfield display [2], while the latter demonstrates a common dynamic mapping technique used in GIS visualizations.

We now demonstrate how our approach can be used to simulate two well-known orthogonal geometric distorted view techniques: graphical fisheye views [96, 97] and compressed arctangent (CAT) graphs [60].

As discussed in Section 4.6.3, in order to simulate geometric distorted views, we must be able to

1. compute the necessary transformations that determine node positions in the distorted view in terms of their positions in the normal view

2. use a similar approach to compute node sizes in the distorted view

3. map positions in one of two possible ways: to either absolute or relative x and y properties;

4. map sizes in one of four possible ways: relative x and y sizes versus absolute x and y sizes (non-uniform scaling), and relative size versus absolute size (uniform scaling).

In the following view definition, attributes 6 and 7 are the CATGraph transformed x and y coordinates based on the original longitude and latitude data, while attributes 8 and 9 compute the average distance to the focal points. The former are then mapped onto relative x and y position and the latter onto relative x and y size. Figure 5.24 shows the resulting CATGraph with focal points *Cincinnati* and *Laredo*.

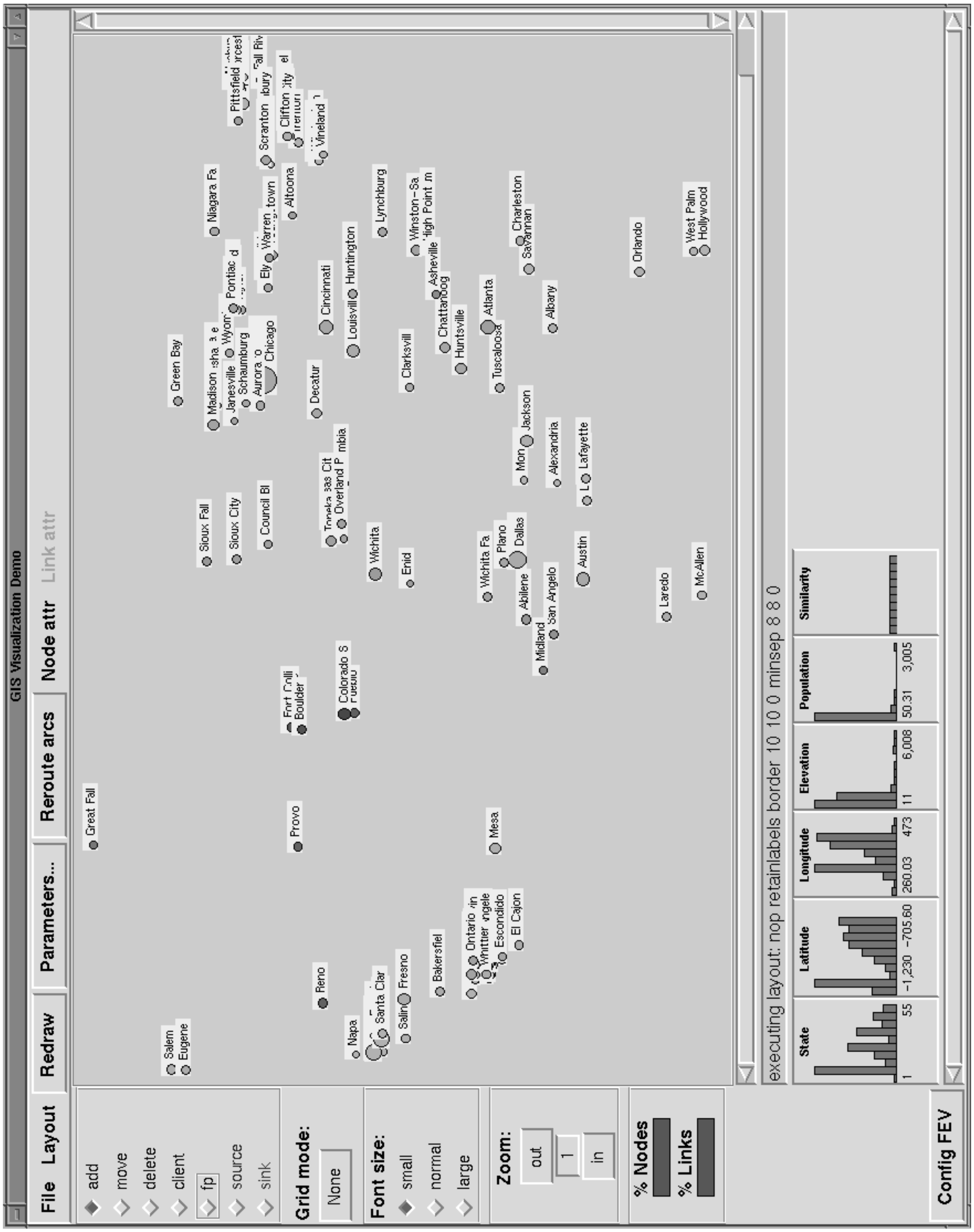


Figure 5.18: GIS visualization application

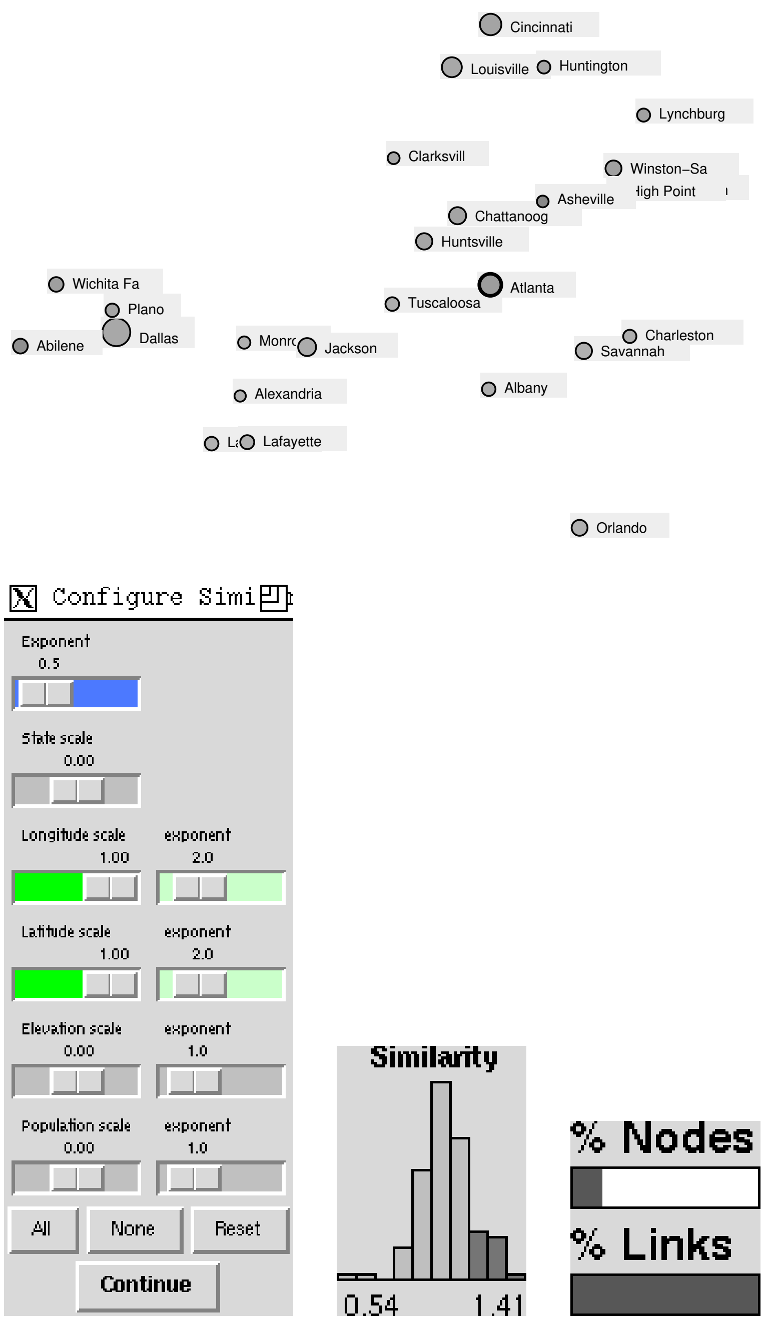


Figure 5.19: Masking distant cities.

```
v:V {
  v[6] := CNT(VF) > 0 ? AVG(f:VF,atan(0.5 / mag(f) * (v[1] - f[1]))) : v[1]
  v[7] := CNT(VF) > 0 ? AVG(f:VF,atan(0.5 / mag(f) * (v[2] - f[2]))) : v[2]
  v[8] := AVG(f:VF,abs(v[6] - f[6]) * -1)
  v[9] := AVG(f:VF,abs(v[7] - f[7]) * -1)
}
```

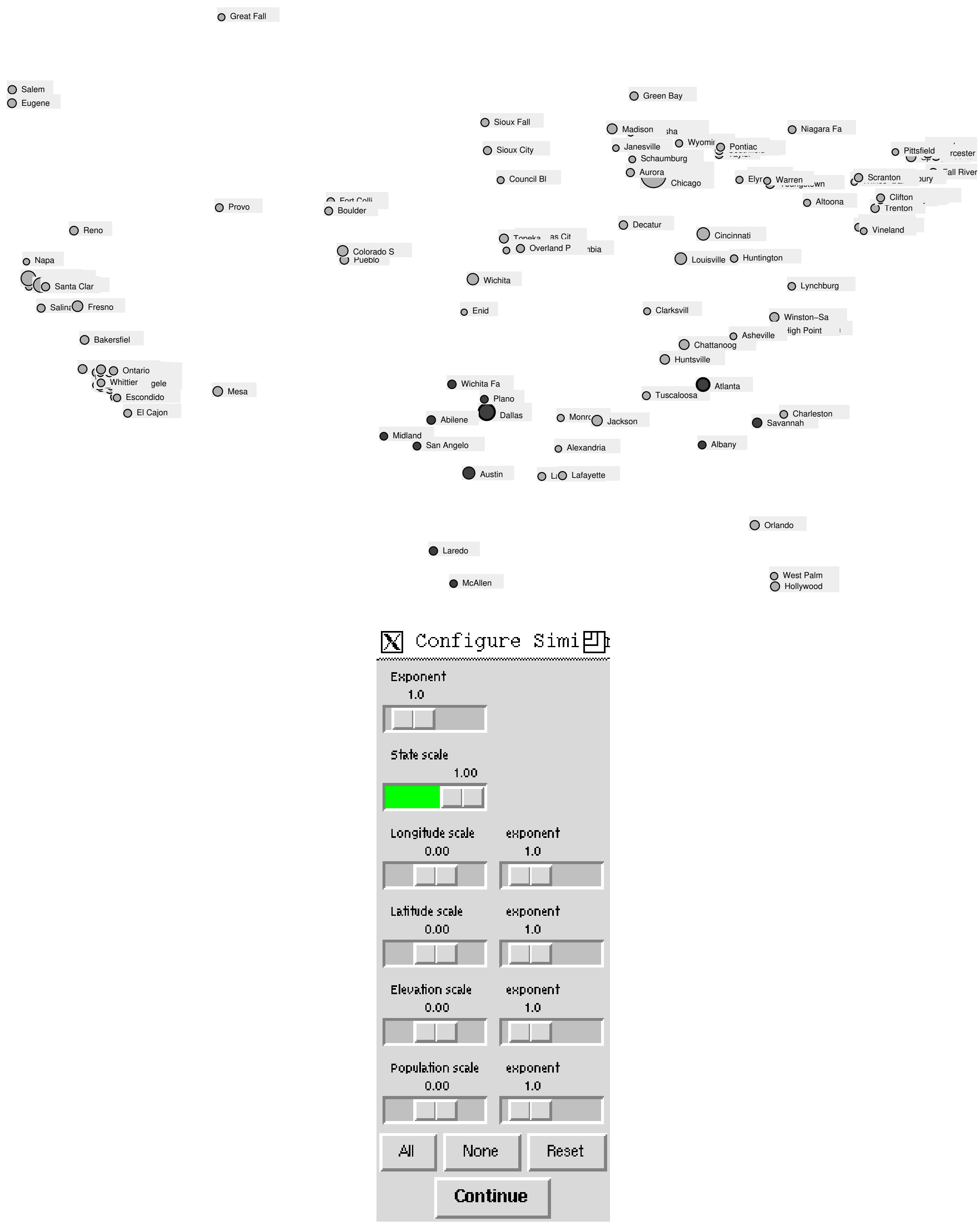


Figure 5.20: Emphasizing cities in selected states (Texas and Georgia).

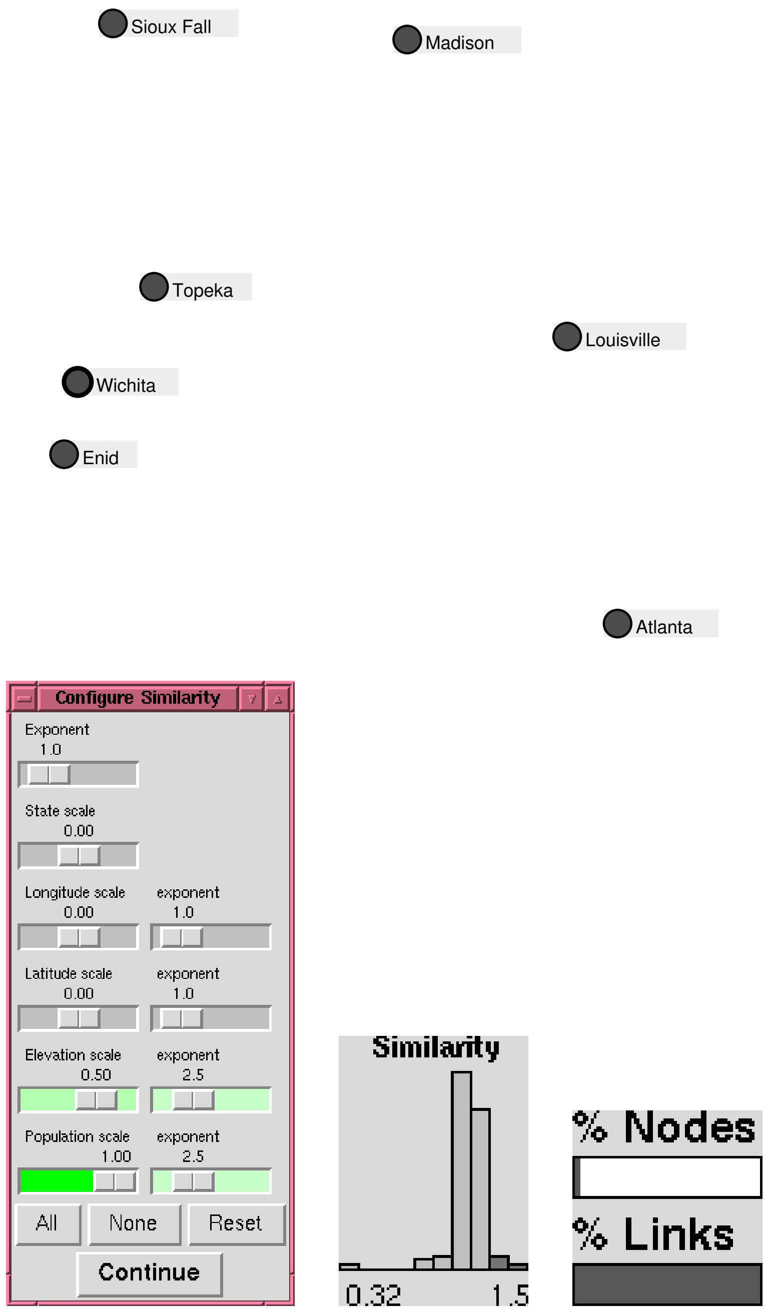


Figure 5.21: Finding cities of comparable size and elevation to that of Wichita.

Graphical fisheye views can be generated in exactly the same manner as CATGraphs, except that the formula for the distorted coordinates becomes considerably more complicated; the formula for the x position is shown below (the formula for y is the same with the obvious substitutions). Figure 5.25 shows a graphical fisheye view with focal point *Dallas*.

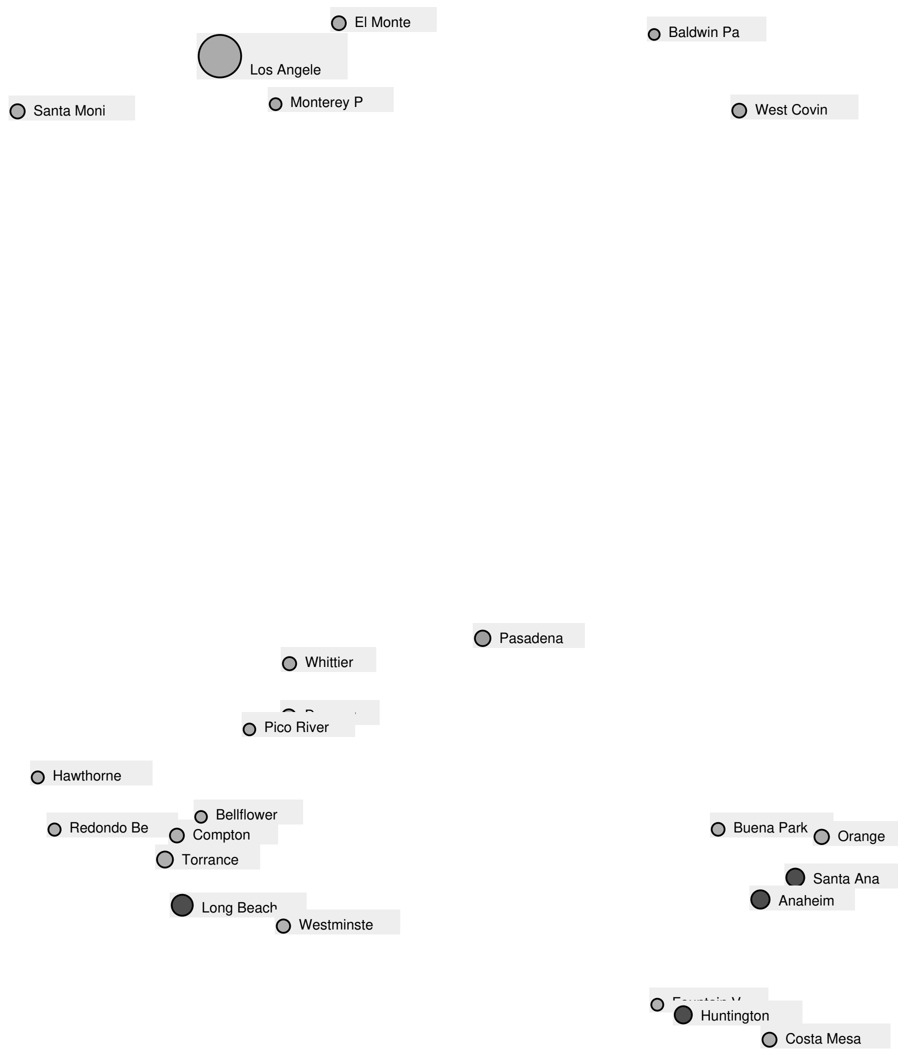


Figure 5.22: Zoomed-in view on the Los Angeles area.

```
v:V v[6] := (mag(f) + 1) * dnorm * dmax / (mag(f) * dnorm * dmax) + f[1]

where:
  dnorm is       v[1] - f[1]
  and dmax is    v[1] >= f[1] ? MAX(u:V,u[1]) - f[1] : MIN(u:V,u[1]) - f[1]
```

5.5 Hierarchical Information – File Directories

Current personal computers and workstations are equipped with hard-disk drives that can typically store anywhere from 200 megabytes to as much as 4 gigabytes of data. Not surprisingly, large disks often lead to large and complex file directory structures. Since file directories are hierarchical (if we ignore symbolic links), the sheer number

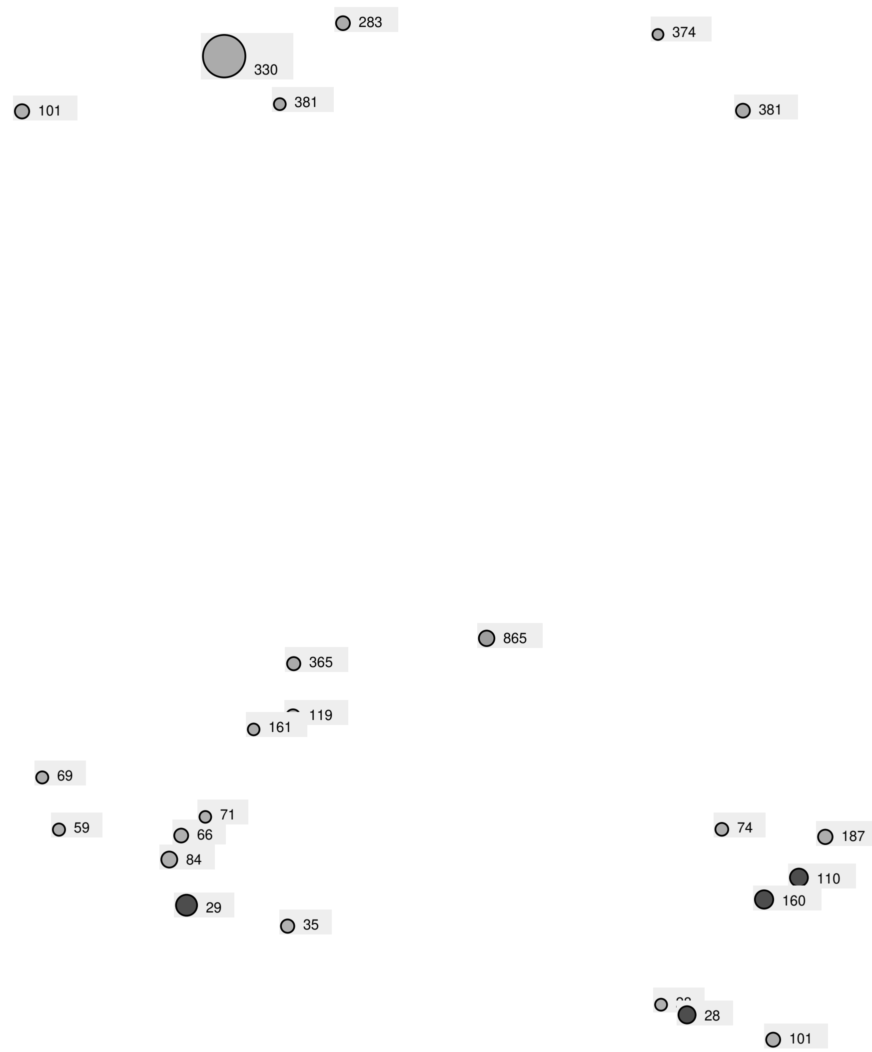


Figure 5.23: Los Angeles area; here node labels show elevation figures.

of directories gives rise to very large trees. Although some commercial operating systems include software for viewing and manipulating file directories, these systems stand to benefit from DQ and DM techniques.

We have applied our approach by viewing file directory hierarchies on a Unix workstation. The base relations in the file directory database are *Directory* and *Containment*:

```
Directory(Id,Name,
 0:Size              % total number of bytes in all files in directory
)
Containment(Id,ParentDir,ChildDir)
```

Since the directory information is hierarchical, we have the option to visualize it in two radically different ways: as ordinary node-and-link graphs, and as nested graphs. We divide our discussion accordingly.

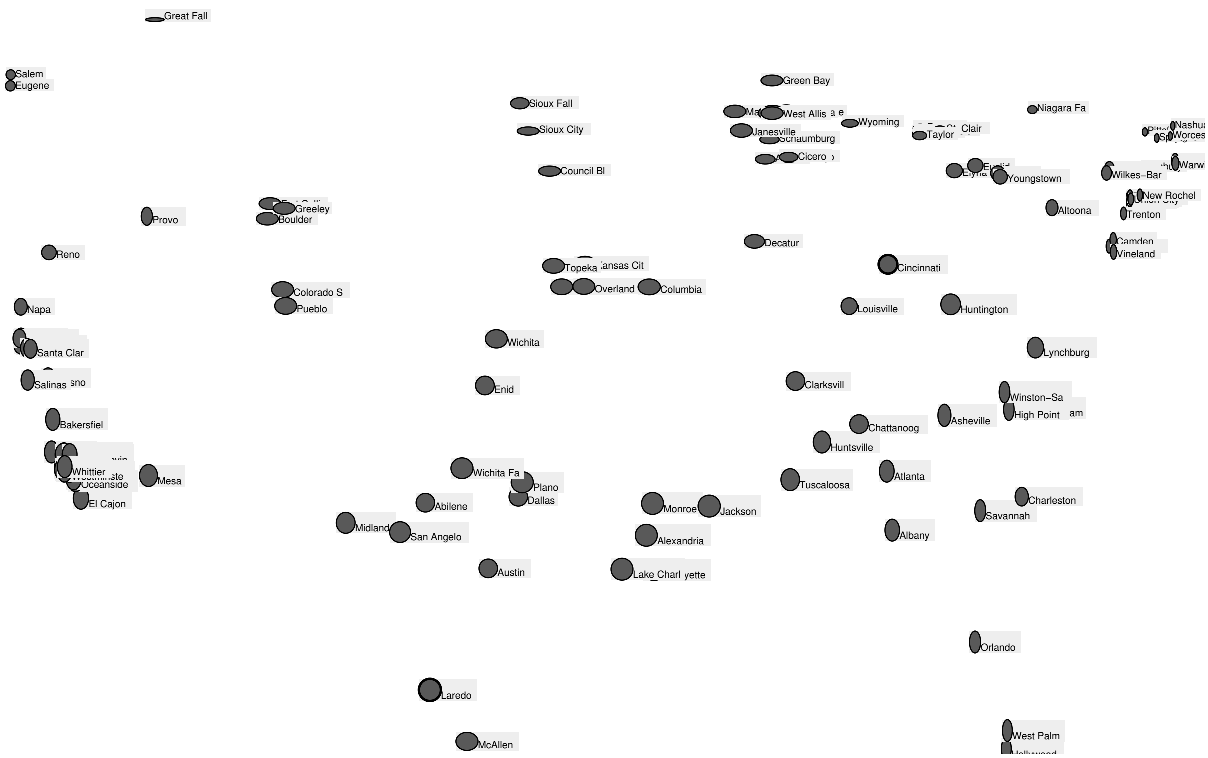


Figure 5.24: CATGraph distorted view simulation; focal points are *Cincinnati* and *Laredo*.

Figure 5.25: Graphical fisheye view simulation; focal point is *Dallas*.

5.5.1 Flat Graph Representations

We have defined the following view on the database:

```
P := APSP(V,A,0,1)                   % shortest paths

v:V {
  v[1] := CNT(TC(v,reverse,A)) - 1   % Depth in the hierarchy
  v[2] := AVG(f:VF,SP(P,v,f))        % Distance in the hierarchy
  v[3] := CNT(VF) > 0 ? -1 * (v[1] + v[2]) : 0
}                                    % Similarity based on depth and distance
```

Notice that the notion of similarity captured by attribute 3 is identical to the notion of *degree of importance* in the generalized fisheye view [34].

Figure 5.26 shows a screen shot of our directory viewer displaying a portion of a directory tree on 232 nodes. Directory size has been mapped onto the *hue*, *saturation*, and *size* node properties, and the nodes were positioned by a left-to-right hierarchical layout algorithm.

Figure 5.27 shows the same portion of the tree, but the deepest nodes have been masked and the graph has been redrawn.

Figure 5.28 shows the directory subtree after the small subdirectories have been masked and the remaining nodes have been redrawn. This type of visualization is helpful for quickly locating large directories, especially, for example, when the disk is full and some unnecessary files should be removed. Although we did not have last-modification date and time information, adding these attributes to the database could similarly help to identify large old files that could be candidates for deletion or archiving.

Figure 5.29 shows a first order filtering fisheye view with focal point *Components*, while Figure 5.30 shows a second order filtering fisheye view of the directory subtree with the same focal point.

5.5.2 Nested Graph Representations

An alternative strategy for viewing (hierarchical) file directory data is as a nested graph in which each subdirectory is visually nested inside its parent directory. This approach can be easily realized in the Graphite system by mapping the *Containment* relation into a set of blobs rather than a set of (adjacency) arcs.

Thus assuming that we are now given a graph with a set of blobs instead of arcs, we can create the following view:

```
P := APSP(V,I,0,1)                   % shortest paths in nesting hierarchy

v:V {
  v[1] := CNT(TC(v,reverse,I)) - 1   % Depth in nesting hierarchy
  v[2] := AVG(f:VF,SP(P,v,f))        % Distance in nesting hierarchy
  v[3] := CNT(VF) > 0 ? -1 * (v[1] + v[2]) : 0
                                     % Similarity based on depth and distance
   v[4] := MIN(f:VF,((SP(P,v,f)+1) / mag(f)) * -1)
}                                    % Similarity based on distance only
```

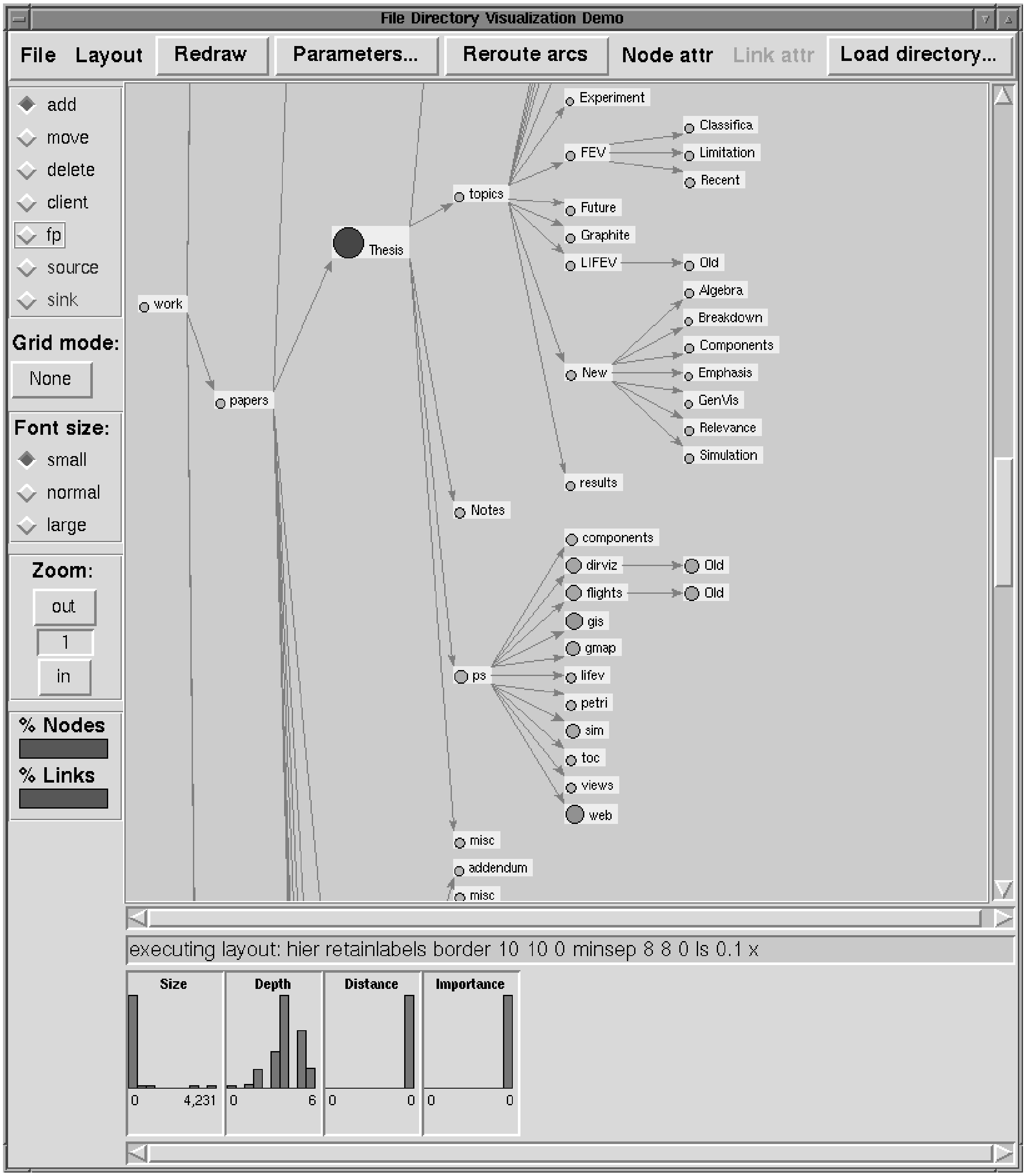


Figure 5.26: The directory tree visualization application.

The above view defines two similarity attributes; the first is identical to the notion *degree of importance* in the generalized fisheye view [34] and the notion of priority in the Abridgment facility of the D-ABDUCTOR system [74], while the second is based on proximity (inverse distance) in the nesting hierarchy scaled by focal point magnification.

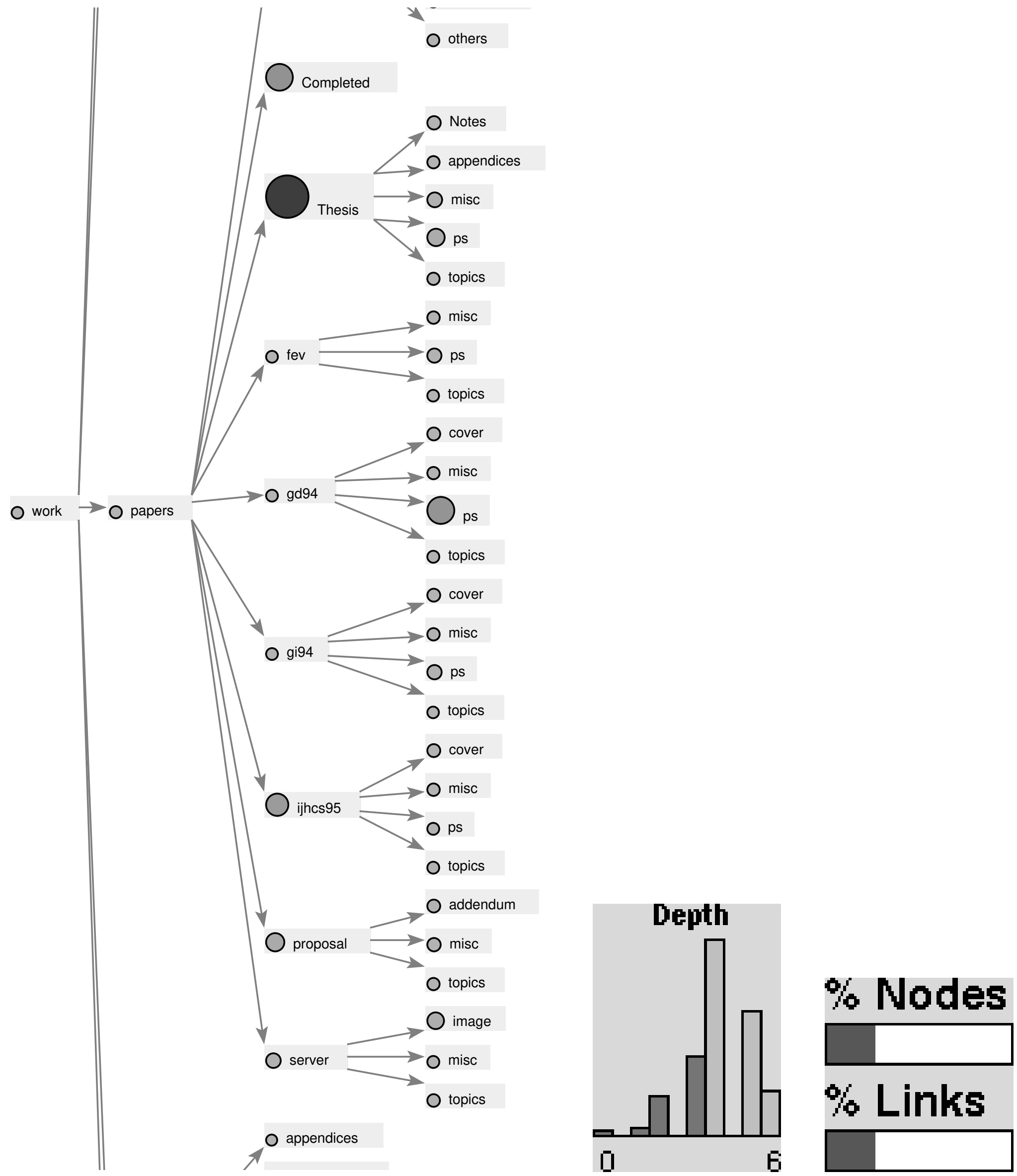


Figure 5.27: Masking deepest directories.

Figure 5.31 shows a nested graph visualization of the Unix file directory hierarchy, in which only a small portion of the entire graph is visible. The nodes are positioned using a heuristic one-dimensional layout algorithm called *stack*, that locally minimizes aspect ratios of nested regions and composes the results using a bottom-up divide-and-conquer strategy. Notice the sacrifice of global context for local detail in Figure 5.31. Although nested graphs can simultaneously display nested regions and nodes connected by links, they often tend to be more difficult to understand. In particular, it tends to be more difficult to form a mental model of the global structure of the graph. This is especially true of large graphs that require panning to view in their entirety. It is possible to use various distortion techniques to remedy this problem. To these ends, Figures 5.32 and 5.33 show

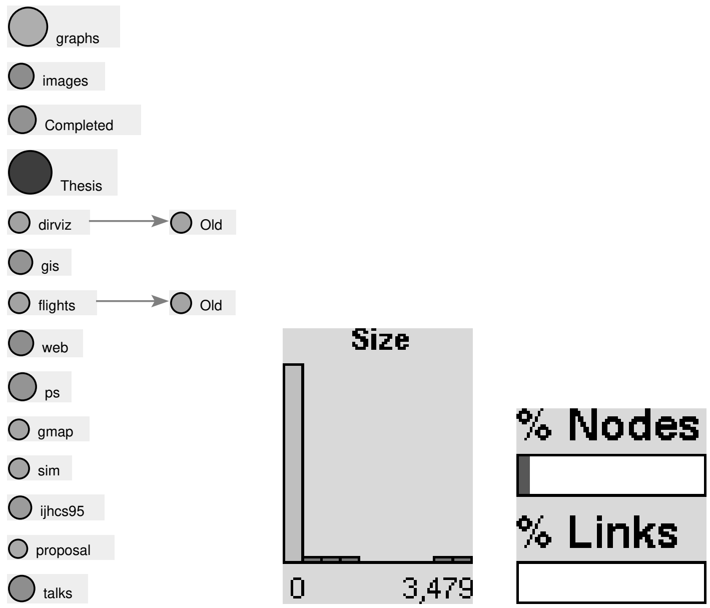


Figure 5.28: Finding large directories by masking small ones.

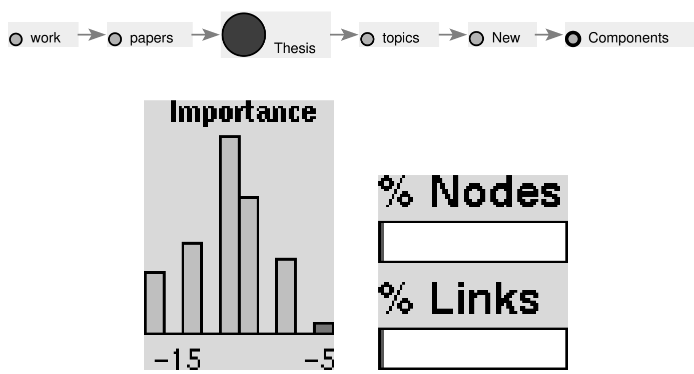


Figure 5.29: A first order filtering fisheye view.

two simple distorted views of the nested graph (with focal point *Simulation*) obtained by mapping the first and second similarity attribute onto relative node size, respectively. In both cases, the graph is redrawn with the *stack* algorithm. Notice how the second notion of similarity which factors in focal point magnification yields superior results by making it possible to compress the graph so that most of the context can fit in a single window; the effect of the compression is reflected by the shape of the corresponding histograms – in the second a larger proportion of the nodes are pushed to the left side of the histogram.

The previous two examples demonstrate the key idea of the *layout-independent fisheye view* [78, 79] a technique that we proposed several years ago. This key idea is that distorted views of nested graphs should be obtained by varying the relative sizes of nodes in the nesting hierarchy **independently** of the nodes' positions in the original layout. Another important aspect of this technique was the ability to specify multiple variable-magnification focal

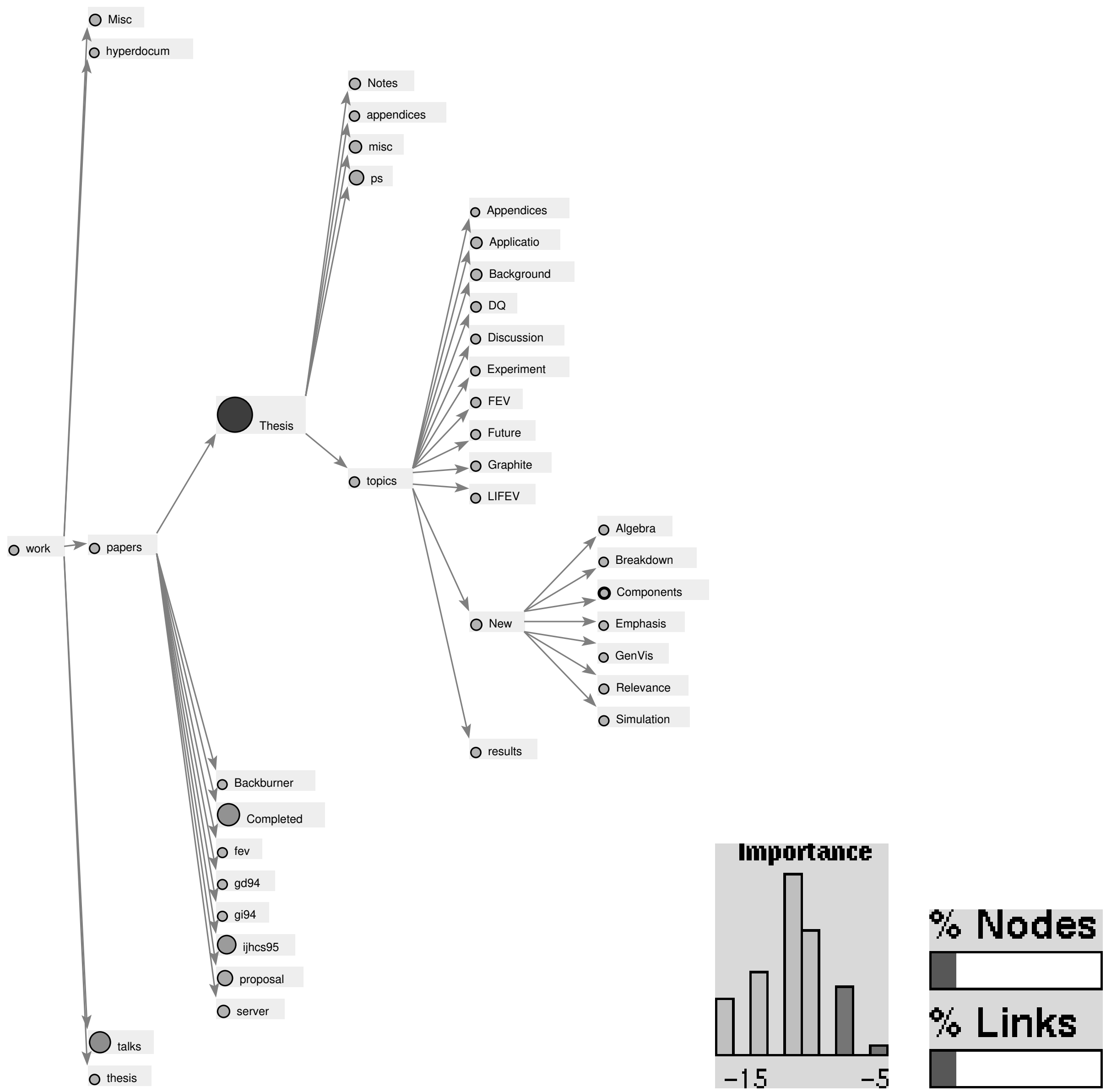


Figure 5.30: A second order filtering fisheye view.

points.

Next, we show that our view definition language can in fact simulate the layout independent fisheye view algorithm. One simplified (not exact) simulation can be obtained with the following view:

```
v:V {
  v[1] := CNT(TC(v,reverse,I)) - 1
  v[2] := AVG(f:INTERS(VF,TC(v,forward,I)),pow(mag(f),(v[1]+1)/(f[1]+1)))
  v[3] := v[2]=0 ? MAX(u:SELECT(w:TC(v,reverse,I),w[1] > 0),u[2]) : u[2]
}
```

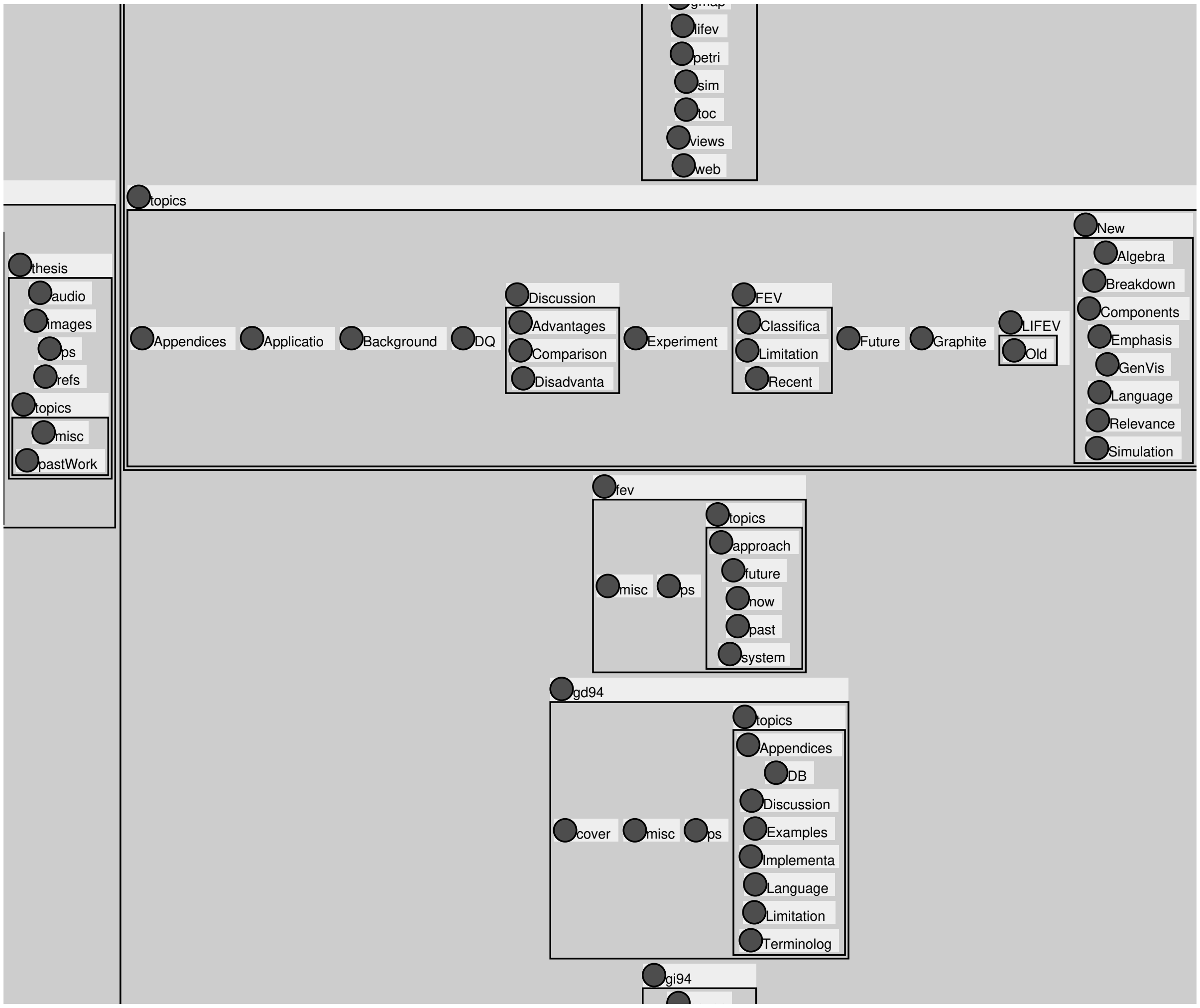


Figure 5.31: A nested graph visualization of Unix file directories showing a small portion of the entire graph.

In the above, attribute 1 is the nesting depth, attribute 2 is the magnification computed as the average propagated magnification over all focal points contained in the nesting subtree rooted at the vertex, and attribute 3 assigns a magnification to the rest of the vertices by choosing the maximum magnification assigned to any ancestor in the nesting hierarchy. While this simulation is simplified because it does not model the notion of a synthetic focal point,[1] this simplification only introduces difficulties when greatly varying magnifications are assigned to focal points in common subtrees. Figure 5.34 shows a layout-independent fisheye view with two focal points.

We conclude this section on hierarchical data visualization by showing how our view definition language can simulate a radically different technique: the Treemap [56, 112]. To accomplish this, we must define the following view:

[1] A synthetic focal point is a non-focal point vertex that contains several focal points in the subtree rooted at that node.

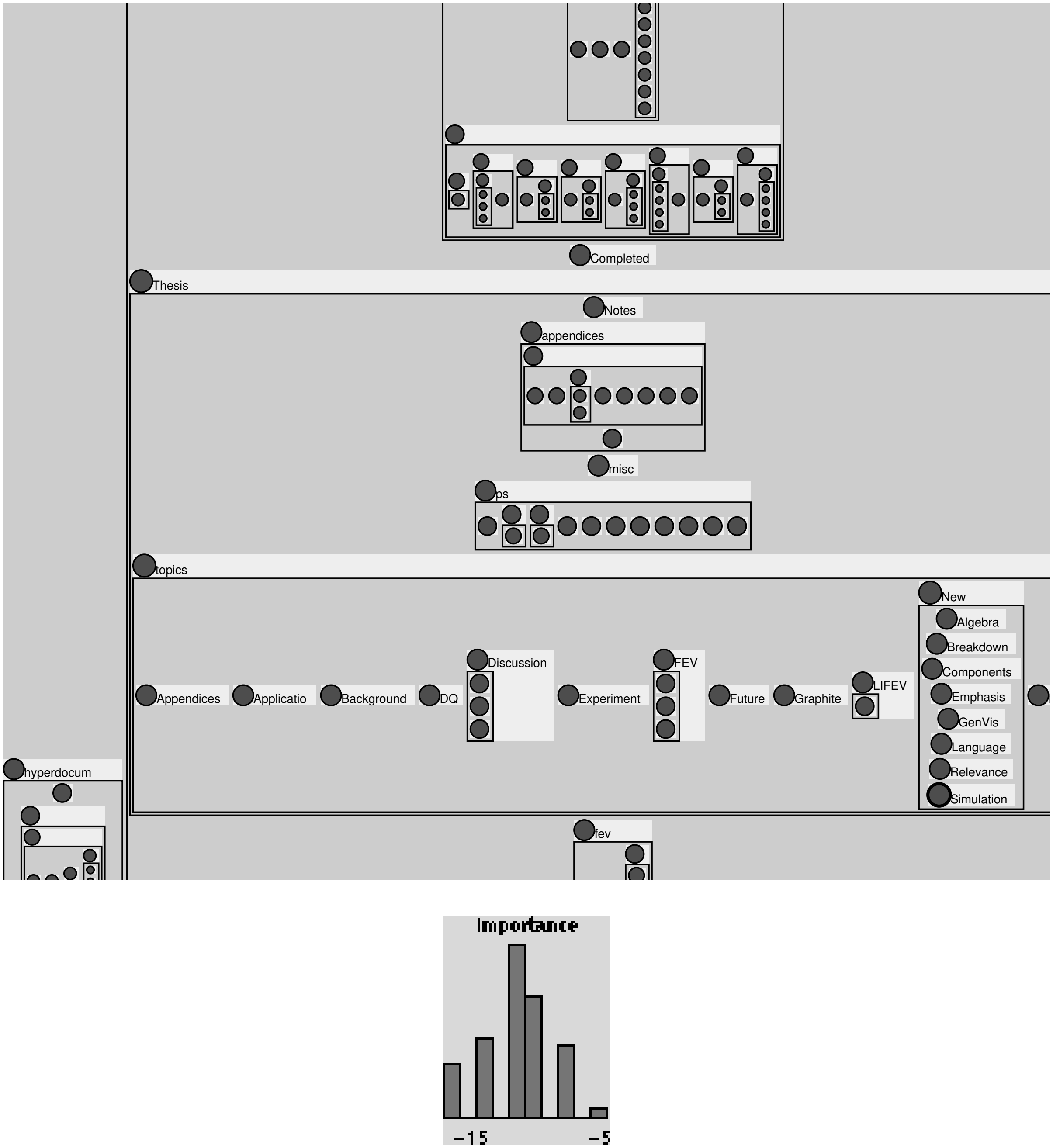
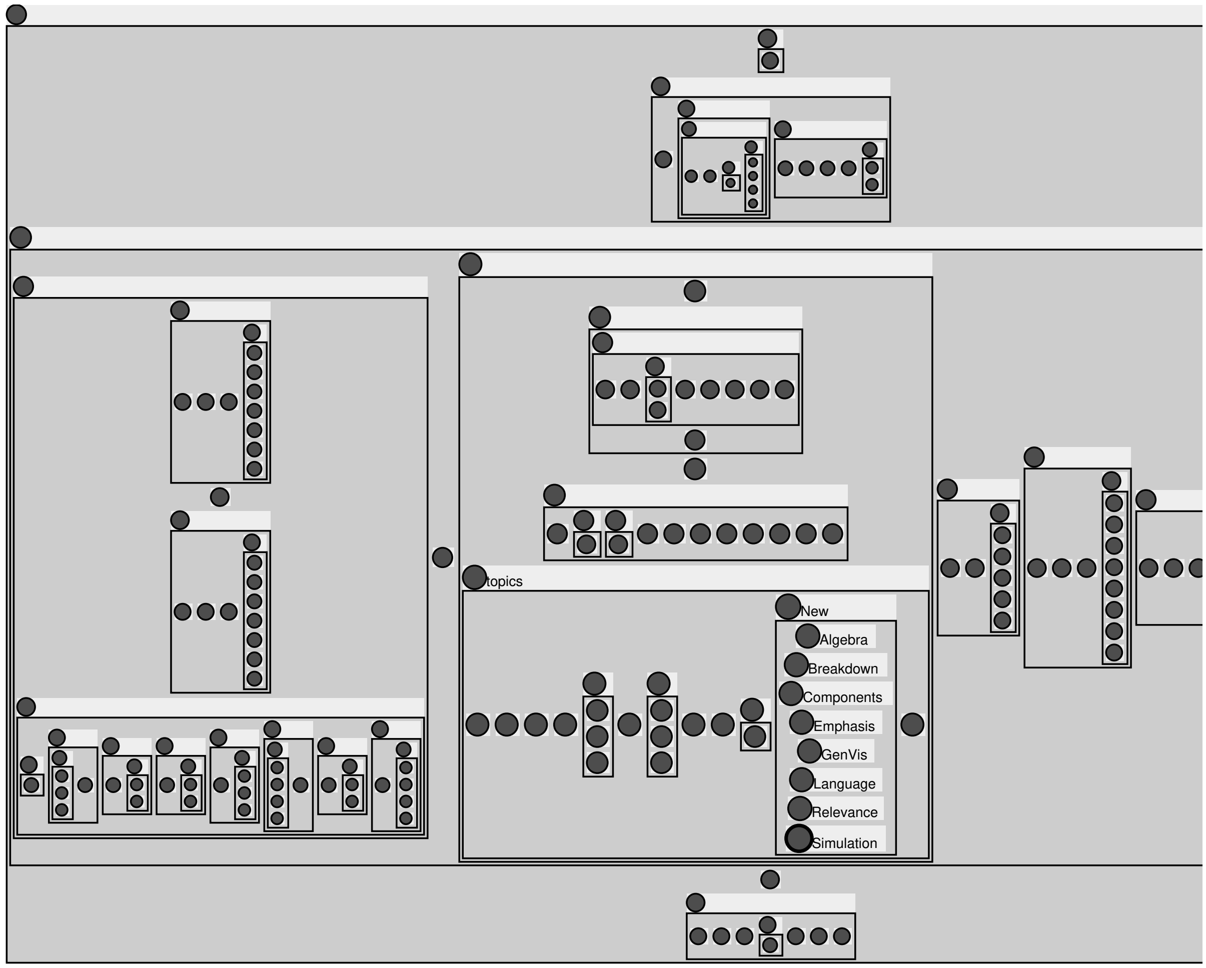


Figure 5.32: A simple distorted view of the nested graph with focal point *Simulation*.

```
v:V {
  v[1] := CNT(TC(v,reverse,A)) - 1       % Depth in the hierarchy
  v[2] := CNT(CB(v)) = 0 ? 1 : 0         % unit value for each leaf node
  v[3] := SUM(u,TC(v,forward,I),u[2])    % sum of leaf values for internal node
}
v:SELECT(u:V,pb(u) != 0) v[4] := v[3] / pv(pb(v))[3]
                                         % ratio of child value to parent value
v:SELECT(u:V,CNT(CB(u)) = 0) {
  v[5] := PROD(u:TC(v,reverse,I),u[1] % 2 = 0 ? u[4] : 1) * 500
  v[6] := PROD(u:TC(v,reverse,I),u[1] % 2 = 0 ? u[4] : 1) * 500
}                                          % final width and height of each leaf
```



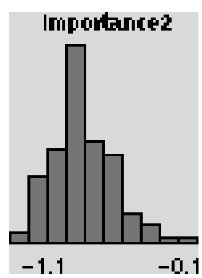


Figure 5.33: A better distorted view of the nested graph with focal point *Simulation*.

To scale the leaf nodes to proper sizes we map vertex attributes 5 and 6 to absolute x and y sizes. The position of each node can be computed using a variant of the one-dimensional layout that is similar to *stack* (see Section A.3), except that it alternates between horizontal and vertical one-dimensional layout at each nesting level in the hierarchy. Note, however, we don't have to rely on the existence of such a specialized algorithm. Instead, we can use the view definition language to declaratively assign a layout algorithm to each blob in the hierarchy as

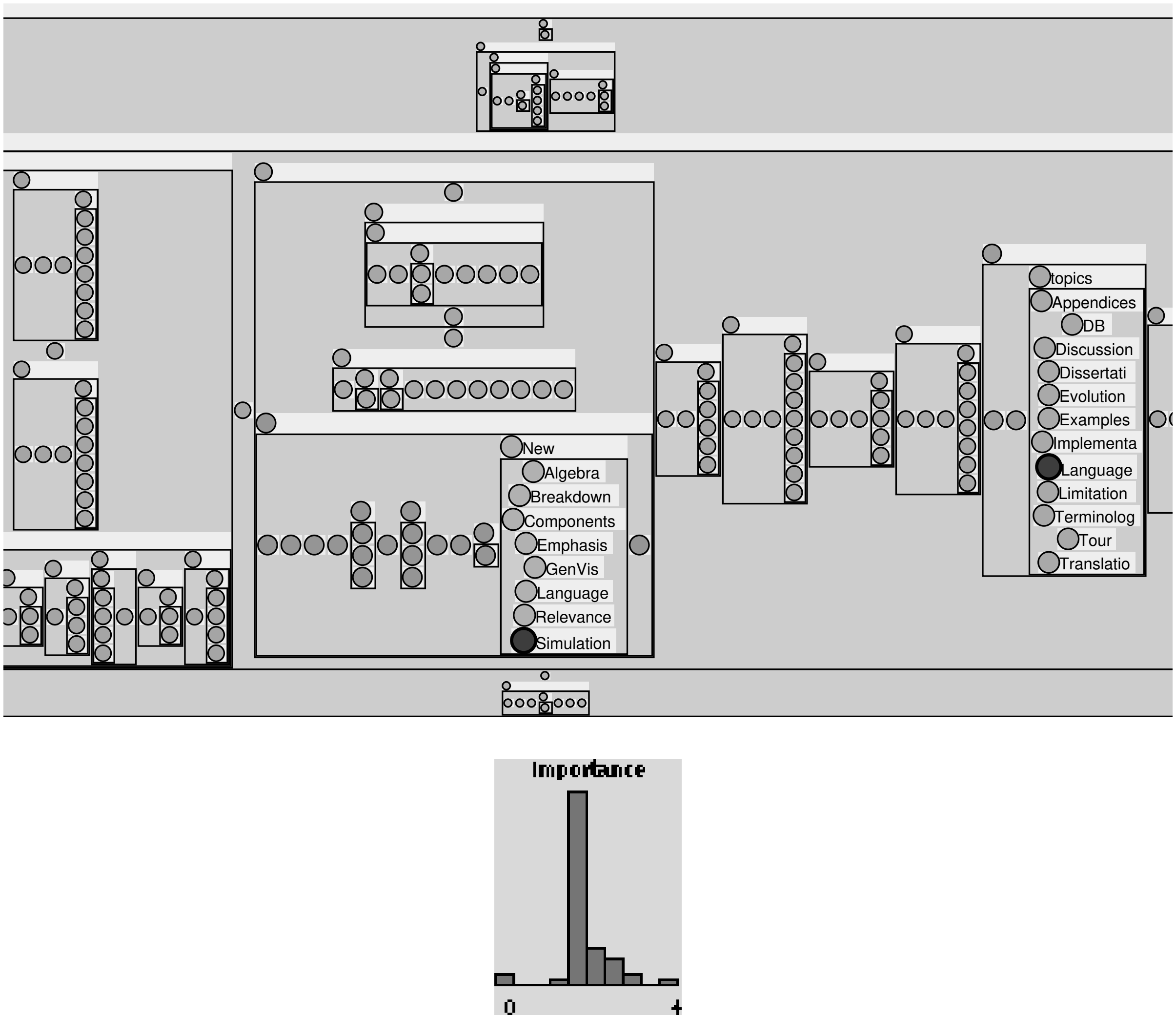


Figure 5.34: A layout independent fisheye view of the nested graph with focal points *Simulation* and *Language*.

follows:

```
b:B layout(b) := pv(b)[1] % 2 = 0 ? "1d x" : "1d y"
```

Using this technique we can generate arbitrary composite layouts by declaratively assigning layout algorithms to the blobs of the hygraph and in doing so, automate Henry's Interactive Graph Layout approach [49, 50].

Figure 5.35 shows a Treemap in which each leaf is assigned a weight of 1, while in Figure 5.36, the weight of each leaf is set to the size of the corresponding directory.

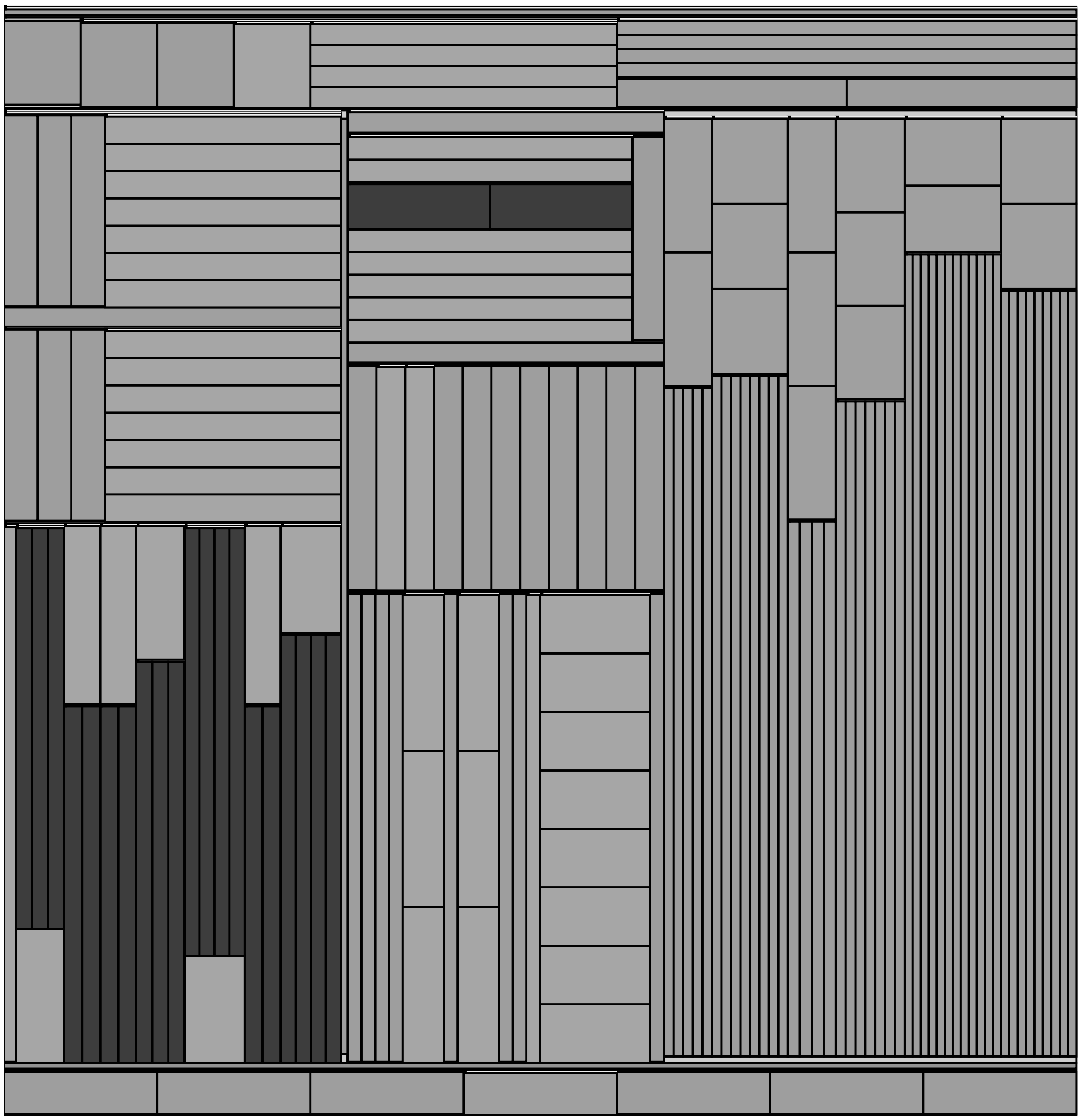


Figure 5.35: Simulating Treemaps with unit leaf weights.

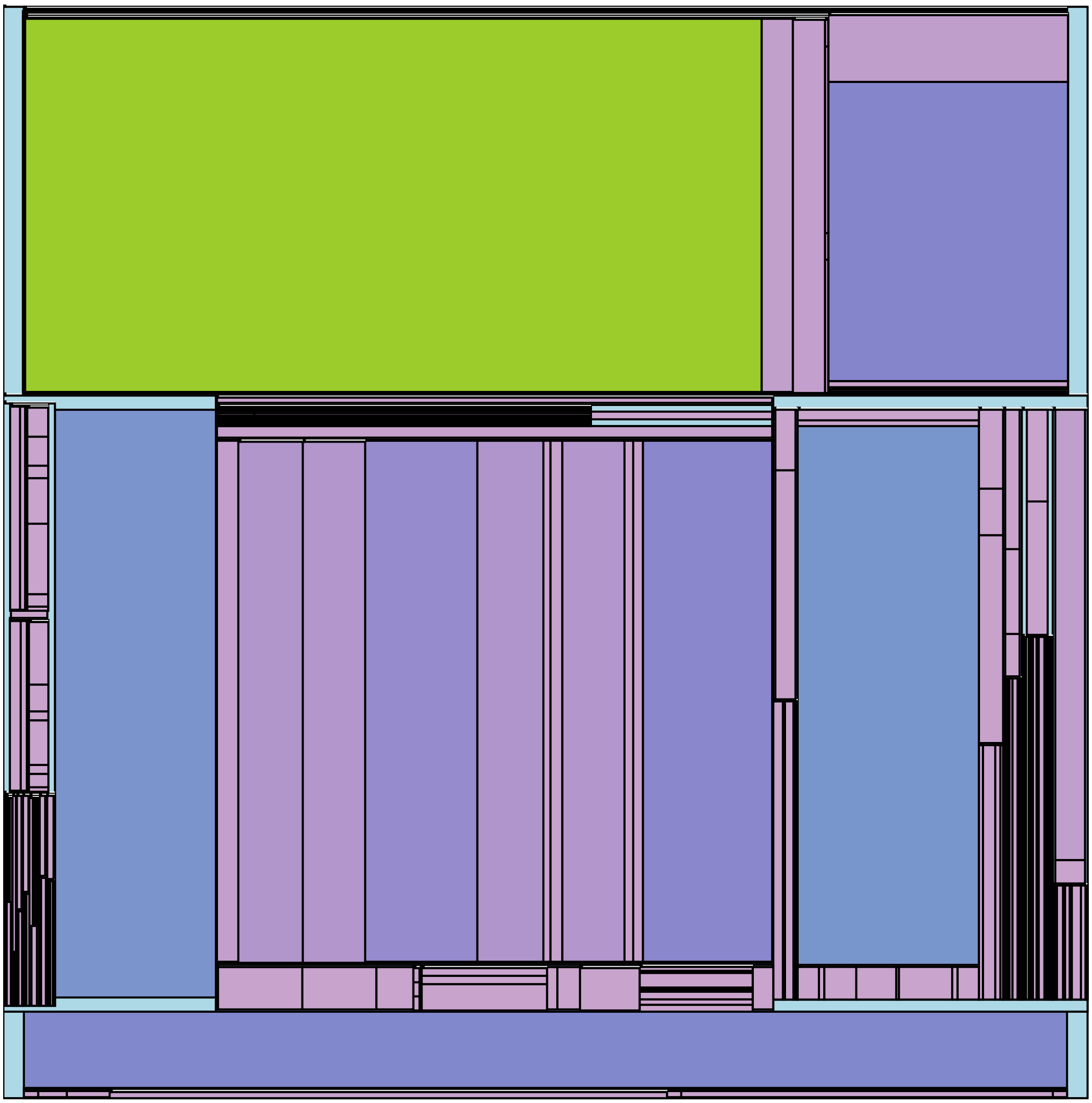


Figure 5.36: Simulating Treemaps with variable leaf weights.

Chapter 6

Summary

6.1 Lessons Learned

Data visualization is still a relatively new research area. As such, there are still many avenues to explore. In particular, unlike more established and mature research areas, there is relatively little theoretical work in this area. However, the state of affairs is gradually improving. Work in data characterization (*e.g.,* [67, 70, 90, 91]) now provides a better understanding of the role and importance of data classifications in generating meaningful and effective visualizations. Work in the *psychophysics* of visualization (*e.g.,* [5, 13, 17, 111]) is gradually creating a set of principles that user interface designers (and adaptive user interfaces) can use to produce more effective visualizations.

In the absence of any complete theory, one of the most important guiding "principles" in data visualization has been to strive for simple, flexible approaches that can be combined in powerful ways. It was the realization that *no hammer is good for all jobs* that lead researchers to seek sufficiently simple yet useful tools that could be added to the growing "toolbox" of visualization techniques. Examples of such include brushing [7], see-through tools [9, 106], selective dynamic manipulation [11, 12], and of course, dynamic mapping and dynamic queries.

We too were motivated by this *simpler is better* philosophy, as evidenced in the product of this work. Namely, we have shown that by taking two proven tools (DQ and DM) and applying some "glue" (active database views), we obtain a new tool (dynamic FEVs) which not only enhances the original tools (by allowing DQ to pose similarity queries), but also salvages the successful features (similarity functions, focal points) of an otherwise less successful tool (FEVs).

In addition, we have demonstrated through numerous examples how even simple layout algorithms can greatly enhance visualization when combined with other techniques:

- *Airline flights* (Sec. 5.1) – use DM to position nodes exactly, layout to spread nodes to reduce overlaps;
- *Web visualization* (Sec. 5.2) – use DM to position nodes horizontally, layout to spread nodes vertically;

- *Genome structure graphs* (Sec. 5.3) – use DQ to filter graph, layout to redraw result; and,
- *Treemaps* (Sec. 5.5) – use views to compute child/parent ratios, DM to compute exact node sizes, layout to compute node positions.

Furthermore, Graphite could be used to create:

- bar charts – use DM or layout to position bars, DM to set bar heights;
- Gantt (project management) charts – use DM to position nodes and set node widths; and,
- x-y charts – use DM to position nodes.

Views could be used to enhance these standard "business" visualizations by defining similarity functions that could be mapped onto other graphic properties such as hue or saturation. Users could then generate dynamic (adorned) FEVs and pose similarity queries such as *what project phases are similar to phase 1.3.2*, or test hypothesis such as *do other high-volume products have similar distribution costs to breakfast cereal*.

6.2 Limitations

The limitations of our approach can be organized into several categories: scope of applications; computational complexity-related; implementation-related; and usability-related.

Scope of Applications

In order to generate dynamic FEVs of flat data sets, our technique relies on the availability of multidimensional data; in particular, our technique requires suitable quantitative attributes to be used with DM and DQ. Thus the scope of applications for visualizing flat data is limited to those where the data either contains appropriate quantitative information in the form of base attributes, or base attributes that could be used to derive additional attributes, including dynamic similarity attributes.

The scope of applications for visualizing graph and hygraph data is not as restrictive. In particular, the Dye view definition language can be used to derive quantitative attributes in terms of graph topological features such as vertex degree, transitive closure, depth in a hierarchy, and shortest path distance.

Furthermore, in our approach similarity functions (between a pair of data points) are restricted to those that can be encoded as mathematical expressions involving attribute values, standard mathematical functions and operators (*e.g.,* see Section A.4) and topological properties (Section 4.4.2). While this approach is very general, it cannot capture notions of similarity based on content (*i.e.,* not expressed in terms of attribute values) such as similarity of image data or textual passages in documents.

Complexity-related Limitations

A common rule of thumb in building highly interactive systems is that system response time to user actions should be no greater than 100ms. For example, this figure is suggested as the maximum response time when users manipulate DQ components [105]. We believe that it is similarly important that system response to user actions such as selecting or deselecting a focal point, path source or sink, should be likewise efficient. The total time required to complete a system response to such an event can be broken into two parts: the time required to update the affected dynamic attributes, and the the time required to recompute DQ and DM (see Section 4.3). The latter is largely a function of the implementation (*e.g.,* method for evaluating partial-match and range queries and the performance of the graphic display hardware, respectively). As for the former, we've pointed out that both aggregate (Section B.5) and set-valued (Section B.6) expressions have efficient (linear) incremental forms that can be used to update the values of dynamic attributes in response to user actions. In order to obtain such linear update performance, we must guarantee that individual updates require constant time per data point. While this is true of all expressions that can be formed using the core subset of the Dye language (*i.e.,* ordinary math expressions – see Section 4.4.1), it is not true of expressions that make use of some other constructs. In particular, similarity attribute expressions that involve computing the transitive closure (set of reachable vertices) require linear time to evaluate (for each vertex). Similarly, if graph (database) updates are permitted, expressions that make use of the shortest path distances cannot be updated efficiently (in response to graph updates).

As a final comment relating to computational complexity, note that while ordinary mathematical expressions require constant time to evaluate, in practice the size of such expressions affects the magnitude of the constant factor. In particular, as such expressions grow large (*e.g.,* involve tens or hundreds of attributes, operators, or math function calls), the cost of individual updates can become significant and cause update performance to suffer noticeably.

Implementation-related Limitations

In practice, the performance characteristics of graphics display system (both software and hardware) largely dictate the size of datasets that can be efficiently displayed. For example, the current implementation of Graphite uses TCL/TK as a means to display graphs. As such, it is limited by the performance of the TCL/TK canvas widget which, in practice, is capable of effectively visualizing graphs that contain up to 500 vertices and arcs. More efficient graphics hardware-based implementations are likely to fare well at displaying and interacting with graphs containing up to 10,000 vertices and arcs.

Another type of implementation-related limitation concerns the types of visualizations that can be realized with Graphite. While it is clear that our system can generate a variety of two-dimensional plots and graph visualizations based on combinations of layouts obtained by mapping, querying, and automatic graph layout, the ability to realize existing techniques depends in large part on the power and flexibility of the DM, DQ, and graph layout components. For example, Graphite could be used to generate Cone Tree visualizations [89] if its layout component were extended to include a three-dimensional cone tree layout algorithm. Similarly, many 3-D techniques (*e.g.,* 3-D mapping strategies explored with PLUM [88]) could be realized if Graphite had 3-D graphics capability; the current Graphite implementation already supports 3-D versions of all available graph layout algorithms (see Section A.3) and 3-D DM.

The last implementation-related limitation concerns how dynamic attribute expressions are evaluated. In particular, in the current implementation of Graphite such expressions are evaluated by a run-time interpreter costing at least a ten-fold increase in compute time; an alternative may be to use an efficient run-time compiler to convert mathematical expressions into a form that could be evaluated more efficiently.

Usability-related Limitations

It is commonly known that users have difficulty understanding very large or complex graphs. Our approach to deal with visual complexity is to provide users with flexible tools that can be combined in powerful ways. As we have shown, DM, DQ, and automatic graph layout can be combined to realize many powerful presentation strategies. However, it is likely that users would have to possess a relatively high degree of skill in order to make the most of such flexible and powerful tools.

A second common limitation is users' difficulty in coping with highly-dimensional data (*e.g.,* containing tens or hundreds of attributes). In particular, our proposed interactive technique for specifying similarity functions (Section 2.7), may be too flexible in the sense that it may provide users with too many degrees of freedom, especially when dealing with a large number of attributes (*e.g.,* in the genome map structure graph application – Section 5.3), and thereby make the task of interactively identifying suitable similarity functions a difficult one. In such instances, either the number of attributes that are made available for defining similarity functions must be limited, or the range of functions that can be derived interactively could be more restrictive. Note that this difficulty does not arise in applications where the notions of similarity are predefined (*e.g.,* in the airline flights application – Section 5.1, or in the file directory visualizer – Section 5.5).

6.3 Directions for Future Work

Although we have applied database techniques to a visualization problem, the database research area has much left to offer. In particular, results on query optimization, incremental query evaluation, and incremental view maintenance can find immediate application in DQM systems. The challenge is to determine which database technologies can best support DQ (*e.g.,* main-memory databases, object-oriented databases) by virtue of providing the most appropriate features (*e.g.,* efficient incremental view maintenance). One specific challenge, for example, is to determine how to maintain shortest paths and transitive closure in the presence of graph updates; although a number of incremental (on-line) strategies have been proposed, the associated algorithms tend to be very complex and remain computationally expensive. One compromise is to restrict the use of shortest paths and transitive closure to applications that manipulate hierarchical data (*e.g.,* hygraph nesting hierarchy).

Second, as DQM systems become more complex, they become more difficult to construct and test. As such, it becomes increasingly important to be able to specify how various components are interrelated; this suggests a growing need for rules that describe display invariants (*e.g.,* as suggested in [2]); one possible implementation strategy is to exploit active database features such as triggers [71].

Third, we must not forget one of the main design goals of visualization systems, namely *ease of use*, and with that in mind, observe that there remains an unresolved issue regarding how database views for DQM systems should be defined. In particular, it is desirable to permit users to define views interactively, as with our user-defined similarity dialog box, the Aggregate Manipulator [38], or the filter/flow boolean query interface [104, 105], although the expressive power of such approaches must be carefully weighed against their comparative ease of use and ease of implementation.

6.4 Summary of Contributions

The key contributions of this thesis are:

1. to show that in addition to creating ordinary (static) derived attributes, active database views can be used to define and maintain a class of dynamic derived attributes – attributes whose values are updated in response to user actions;

2. to show that (dynamic derived) similarity attributes can be used to define similarity functions that assign to each point in the visualization a number that reflects how similar (alternatively relevant, important) that point is with respect to a subset of selected (focal) points;

3. to show that similarity functions expressible using the core component of Dye (Section 4.4.1) have a simple incremental form that makes it possible to update a similarity attribute value in constant time per data point;

4. to show that similarity attributes can be used in conjunction with DQ and DM to pose similarity queries by direct manipulation, which may afford the conversion of previously challenging cognitive tasks into relatively simple perceptual tasks; and,

5. to show that similarity attributes when used in conjunction with DQ and DM generalize and simplify the notion of fisheye view:

    - filtering FEVs are created by applying DQ to similarity attributes;
    - distorting FEVs are created by mapping similarity attributes to position and size graphic properties with DM; and,

- adorned FEVs are created by mapping similarity attributes to other graphic properties (*e.g.,* the hue and saturation components of colour).

Some secondary contributions of this thesis are:

1. an implementation and description of five representative visualization applications that benefit from the addition of similarity attributes:
   - transportation networks (airline flights)
   - abstract networks (World-Wide Web structure visualizations and genome structure graphs)
   - geographic information systems (U.S. cities)
   - hierarchical data visualizations (Unix file directories)
2. a simple method to easily specify (static) derived attributes through Dye view definitions, including attributes expressed in terms of graph topological properties such as vertex degree, transitive closure, depth in hierarchy, and shortest path distances.
3. an interactive method to define similarity functions (Section 2.7);
4. a description of how database view maintenance techniques can be used to extend the functionality of DQM visualization systems by permitting database updates to occur while data is being visualized thereby making it possible to create powerful active visualizations such as network monitoring tools (Sections 4.3 and 4.5.3);
5. simulations and DM and view definition expressiveness requirements for existing fisheye view techniques:
   - *Dynamic Implicit FEVs:* PLUM [88]
   - *Filtering FEVs:* Generalized FEV of trees [34]
   - *Geometric Distorting FEVs:* CATGraphs [60] and Graphical FEV of Graphs [96, 97]
   - *Non-geometric Distorting FEVs:* D-ABDUCTOR [74] and Layout-indep. FEV [78, 79]
   - Treemaps [56, 112]
6. examples of powerful and flexible visualization strategies realized through the composition of simpler techniques – DQ, DM, and graph layout.

6.5 Conclusion

Throughout this dissertation we have repeatedly argued that since the effectiveness of both DQ and DM techniques is fundamentally linked to the availability of suitable multidimensional data, it is crucial to be able to derive attributes. We have proposed a database view definition language based on an object-oriented algebra with object-preserving operator semantics as a means to derive attributes, and showed that even with limited expressive power (aggregation), such a language can be used to derive and maintain similarity attributes – attributes that capture abstract notions of similarity between each data point and a selected subset of (focal) points. This capability allowed us to generalize and simplify the notion of fisheye view and gave rise to a class of interactive visualizations that we called **Dynamic Fisheye Views**.

In turn, dynamic fisheye views enabled users to employ DQ to pose similarity queries (perform fuzzy searching) by direct manipulation, and reduced a previously challenging or worse yet, practically impossible, cognitive task into a simple perceptual task.

Furthermore, we showed that most similarity attribute definitions yielded a simple incremental form that can be evaluated efficiently. In particular, for each addition or removal of a focal point, most similarity attributes can be updated in constant time per data point, which supports their use in DQ systems that require rapid response to user actions. As the number of visualization systems that employ techniques such as DQ and DM is growing, our approach is likely to have wide-spread impact.

Bibliography

[1] C. Ahlberg and B. Shneiderman. The Alpha Slider: A Rapid and Compact Selector. In *Proc. ACM CHI '94*. ACM, 1994.

[2] C. Ahlberg and B. Shneiderman. Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. In *Proc. ACM CHI '94*, pages 313–317, Boston, MA, April 1994. ACM.

[3] C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic Queries for Information Exploration: An Implementation and Evaluation. In *Proc. ACM CHI '92*, pages 619–626. ACM, 1992.

[4] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[5] D. Bardon. Management of Color Usage in Dynamic Mapping Environments: Balancing Semantics, Visual Ordering and Discernability. In T. Catarci, M.F. Costabile, and S. Levialdi, editors, *Proc. of Advanced Visual Interfaces 1992*, volume 36 of *Series in Computer Science*, pages 50–67. World Scientific, 1992.

[6] L. Bartram, R. Ovans, J. Dill, M. Dyck, A. Ho, and W.S. Havens. Contextual Assistance in User Interfaces to Complex, Time-Critical Systems: The Intelligent Zoom. In *GI '94: Graphics Interface 1994*, Banff, AL, Canada, May 1994.

[7] R.A. Becker and W.S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, 1987.

[8] R.A. Becker, S.G. Eick, E.O. Miller, and A.R. Wilks. Dynamic Graphics for Network Visualization. In *Proc of IEEE Visualization Conf.*, 1990.

[9] E.A. Bier, M.C. Stone, K. Fishkin, W. Buxton, and T. Baudel. A Taxonomy of See-Through Tools. In *Proc. ACM CHI '94*, pages 358–364, Boston, MA, April 1994. ACM.

[10] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proc. of the 17th VLDB Conf.*, pages 577–589, Barcelona, Spain, 1991.

[11] M.C. Chuah and S.F. Roth. SDM: Malleable Information Graphics. In *Proc. of InfoViz '95*, 1995.

[12] M.C. Chuah and S.F. Roth. SDM: Selective Dynamic Manipulation of Visualizations. In *Proc. of UIST '95*, 1995.

[13] W.S. Cleveland. *The Elements of Graphing Data*. Wadsworth Inc., Belmont, California, 1985.

[14] W.S. Cleveland and M.E. McGill. *Dynamic Graphics for Statistics*. Wadsworth Inc., Belmont, California, 1988.

[15] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[16] M.P. Consens, F. Ch. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista. Architecture and Applications of the Hy+ Visualization System. *IBM Systems Journal*, 33(3):458–476, August 1994.

[17] A. Csinger. The Psychology of Visualization. Technical Report TR-92-28, Dept. of Comp. Sci., U. of British Columbia, November 1992. Avail. at http://www.cs.ubc.ca/tr/1992/TR-92-28.

[18] S. Dar, R. Agrawal, and H.V. Jagadish. Optimization of Generalized Transitive Closure. In *7th IEEE Intl. Conf. on Data Engineering*, Kobe, Japan, 1991.

[19] C.J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 3 edition, 1993.

[20] J. Dausset, P. Ougen, H. Abderrahim, A. Billault, J.L. Sambucy, D. Cohen, and D. Le Paslier. The CEPH YAC Library. *Behring Inst Mitt*, 91:13–20, 1992. March 1995 release of YAC data avail. by anon. `ftp` from `ceph-genethon-map.genethon.fr`.

[21] U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Transactions on Data Base Systems*, 8(3):381–416, 1982.

[22] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, to appear. Preprint avail. by anon. `ftp` from `ftp.cs.brown.edu:pub/papers/compgeo/`.

[23] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1990.

[24] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. Technical Report IIAS-RR-91-16E, Fujitsu Laboratories, August 1991.

[25] P. Eades and K. Sugiyama. How to Draw a Directed Graph. *Journal of Information Processing*, 13(4), 1990.

[26] S.G. Eick. Data Visualization at AT&T Bell Laboratories. An AT&T internal summary report. Contact eick@research.att.com, September 1994.

[27] S.G. Eick and G.J. Wills. Navigating Large Networks with Hierarchies. In *Proc. of 1993 IEEE Visualization Conf.*, pages 204–210. IEEE Computer Society, IEEE Computer Society Press, New York, NY, 1993.

[28] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[29] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 2 edition, 1994.

[30] K.M. Fairchild, S.E. Poltrock, and G.W. Furnas. SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases. In R. Guindon, editor, *Cognitive Science and its Applications for Human-Computer Interaction*, pages 201–233. Lawrence Erlbaum Associates, 1988.

[31] R.W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962.

[32] R.H. Fowler, W.A.L. Fowler, and B.A. Wilson. Integrating query, thesaurus, and documents through a common visual representation. In *ACM SIGIR '91*, pages 142–151. ACM, October 1991.

[33] T. Fruchterman and E. Reingold. Graph Drawing By Force-Directed Placement. Technical Report UIUCDCS-R-90-1609, Dept. of Comp. Sci., U. of Illinois at Urbana-Champaign, June 1990.

[34] G.W. Furnas. Generalized Fisheye Views. In *Proc. ACM CHI '86*, pages 16–23, Boston, MA, April 1986. ACM.

[35] E.R. Gansner, S.C. North, and K.P. Vo. DAG–A Program that Draws Directed Graphs. *Software–Practice and Experience*, 18(11):1047–1062, November 1988.

[36] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[37] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, NY, 1979.

[38] J. Goldstein and S.F. Roth. Using Aggregation and Dynamic Queries for Exploring Large Data Sets. In *Proc. ACM CHI '94*, pages 23–29, Boston, MA, April 1994. ACM.

[39] G. Golovchinsky and M.H. Chignell. Queries-R-Links: Graphical Markup for Text Navigation. In *InterCHI'93*, Amsterdam, Netherlands, May 1993.

[40] R.A.M. Gregson. *Psychometrics of Similarity*. Academic Press, 1975.

[41] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. *ACM SIGMOD Record*, 24(2):328–339, June 1995.

[42] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinitions. *ACM SIGMOD Record*, 24(2):211–222, June 1995.

[43] F.G. Halasz. Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*, 31(7):836–852, July 1988.

[44] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[45] E. Harley and A.J. Bonner. A Flexible Approach to Genome Map Assembly. In *Proc. of the Second Intl. Conf. on Intelligent Systems for Molecular Biology*, pages 161–169. AAAI Press, 1994.

[46] E. Harley, N. Goodman, and A.J. Bonner. Graph Algorithms for Assembling Integrated Genome Maps. To be submitted for journal publication., 1996.

[47] M.Z. Hasan, G. Golovchinsky, E.G. Noik, N. Charoenkitkarn, M. Chignell, A.O. Mendelzon, and D. Modjeska. Browsing Local and Global Information. In *CASCON '95*, Toronto, ON, Canada, November 1995.

[48] L.S. Heath, D. Hix, L.T. Nowell, Wake W.C., G.A. Averboch, E. Labow, S.A. Guyer, Brueni D.J., R.K. France, K. Dalal, and E.A. Fox. Envision: A User-Centered Database of Computer Science Literature. *Communications of the ACM*, 38(4):52–53, April 1995.

[49] T.R. Henry. Interactive Graph Layout: The Exploration of Large Graphs. Technical Report 92–03, Dept. of Comp. Sci., U. of Arizona, June 1992. Ph.D. dissertation published as Technical Report.

[50] T.R. Henry and S.E. Hudson. Interactive Graph Layout. In *Proc. ACM UIST '91*, pages 55–64. ACM, 1991.

[51] J.G. Hollands. Presenting a Network Graphically: A Comparison of Performance Using Fisheye View and Scrolling. Master's thesis, Dept. of Psychology, U. of Guelph, Guelph, Ontario, Canada, 1988.

[52] J.G. Hollands, T.T. Carey, M.L. Matthews, and C.A. McCann. Presenting a Graphical Network: A Comparison of Performance Using Fisheye and Scrolling Views. In G. Salvendy and Smith M.J., editors, *Designing and Using Human-Computer Interfaces and Knowledge-Based Systems*, pages 313–320. Elsevier, 1989.

[53] Human Physical Mapping Project at the Whitehead Institute/MIT Genome Center, Release 8. Avail. via anon. `ftp` from `genome.wi.mit.edu`., September 1995.

[54] D. Jablonowski and Guarna A.Jr. GMB: A Tool for Manipulating and Animating Graph Data Structures. *Software–Practice and Experience*, 19(3):283–301, March 1989.

[55] V. Jain and B. Shneiderman. Data Structures for Dynamic Queries: An Analytical and Experimental Evaluation. Technical Report CAR-TR-715 CS-TR-3287, Dept. of Comp. Sci., U. of Maryland, July 1995.

[56] B. Johnson. TreeViz: Treemap Visualization of Hierarchically Structured Information. In *Proc. ACM CHI '92*, pages 369–370, Monterey, CA, May 1992. ACM.

[57] B. Johnson and B. Shneiderman. Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *IEEE Visualization '91*, pages 284–291, San Diego, CA, October 1991.

[58] J.G. Kalbfleisch. *Probability and Statistical Inference*, volume 1 of *Universitext*. Springer-Verlag, 1979.

[59] T. Kamada and S. Kawai. A General Framework for Visualizing Abstract Objects and Relations. *ACM Transactions on Graphics*, 10(1):1–39, January 1991.

[60] Karlis Kaugars, Juris Reinfelds, and Alvis Brazma. A Simple Algorithm for Drawing Large Graphs on Small Screens. In *Proc. of 1994 DIMACS Graph Drawing Workshop, Princeton, NJ*, number 894 in Lecture Notes in Computer Science, pages 278–281. Springer-Verlag, October 1994.

[61] A. M. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.

[62] S. Koenig and R. Paige. A Transformational Framework for the Automatic Control of Derived Data. In *Proc. of the 7th VLDB Conf.*, pages 306–318, 1981.

[63] H. Koike and H. Yoshihara. Fractal Approaches for Visualizing Huge Hierarchies. In *VL '93: IEEE Symposium on Visual Languages*, pages 55–60, Bergen, Norway, August 1993.

[64] H.P. Kumar, C. Plaisant, and B. Shneiderman. Browsing Hierarchical Data with Multi-Level Dynamic Queries and Pruning. Technical Report CAR-TR-772 CS-TR-3474 ISR-TR-95-53, Dept. of Comp. Sci., U. of Maryland, March 1995.

[65] H.P. Kumar, C. Plaisant, M. Teittinen, and B. Shneiderman. Visual Information Management for Network Configuration. Technical Report CAR-TR-716 CS-TR-3288 ISR-TR-94-45, Dept. of Comp. Sci., U. of Maryland, June 1994.

[66] H.S. Liao, M. Osada, and B. Shneiderman. Browsing Unix Directories with Dynamic Queries: An Evaluation of Three Information Display Techniques. Technical Report CAR-TR-605 CS-TR-2841, Dept. of Comp. Sci., U. of Maryland, February 1992.

[67] J.D. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics*, 5(2):110–141, April 1986.

[68] J.D. Mackinlay, G.G. Roberson, and S.K. Card. The Perspective Wall: Detail and Context Smoothly Integrated. In *ACM CHI '91*, pages 173–179. ACM, April 1991.

[69] K.C. Malcolm, S.E. Poltrock, and D. Schuler. Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise. In *Proc. ACM Hypertext '91*, pages 13–24, San Antonio, Tx, December 1991. ACM.

[70] Joe Marks. A Formal Specification Scheme For Network Diagrams That Facilitates Automated Design. *Journal of Visual Languages and Computing*, 2(4):395–414, December 1991.

[71] D. McCarthy and U. Dayal. The Architecture of an Active Database Management System. In *Proc. of the ACM-SIGMOD 1989 Intl. Conf. on Management of Data*, pages 215–224, 1989.

[72] K. Melhorn. *Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[73] E.B. Messinger, L.A. Rowe, and R.R. Henry. A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):1–12, 1991.

[74] K. Misue. D-ABDUCTOR 2.0 User Manual. Technical Report IIAS-RR-93-9E, Fujitsu Laboratories, May 1993.

[75] K. Misue and K. Sugiyama. Multi-viewpoint Perspective Display Methods: Formulation and Application to Compound Graphs. In *4th Intl. Conf. on Human-Computer Interaction*, volume 1, pages 834–838, Stuttgart, Germany, September 1991. Elsevier.

[76] D.A. Mitta. A Fisheye Presentation Strategy: Aircraft Maintenance Data. In *INTERACT '90*, pages 875–880. IFIP, Elsevier, August 1990.

[77] J. Nielsen. *Hypertext and Hypermedia*. Academic Press, San Diego, CA, 92101, 1990.

[78] E.G. Noik. Exploring Large Hyperdocuments: Fisheye Views of Nested Networks. In *Proc. ACM Hypertext '93*, pages 192–205, Seattle, WA, November 1993.

[79] E.G. Noik. Layout-independent Fisheye Views of Nested Graphs. In *VL '93: IEEE Symposium on Visual Languages*, pages 336–341, Bergen, Norway, August 1993.

[80] E.G. Noik. Encoding Presentation Emphasis Algorithms for Graphs. In *Graph Drawing (Proc. GD '94)*, number 894 in Lecture Notes in Computer Science, pages 428–435. Springer-Verlag, October 1994.

[81] E.G. Noik. A Space of Presentation Emphasis Techniques for Visualizing Graphs. In *GI '94: Graphics Interface 1994*, pages 225–234, Banff, AL, Canada, May 1994.

[82] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[83] X. Pintado. Selection and Exploration in an Object-Oriented Environment: The Affinity Browser. In D. Tsichritzis, editor, *Object Management*, pages 79–88. Centre Universitaire d'Informatique, University of Geneva, July 1990. See also `http://cuiwww.unige.ch/OSG`.

[84] J. Pitkow and K.A. Bharat. WebViz: A Tool for WWW Access Log Analysis. In *Proc. Second WWW Conference*, October 1994.

[85] C. Plaisant. Dynamic Queries on a Health Statistics Map. In *Proc. Conf. American Statistical Assoc.*, pages 18–23, Alexandria, VA, 1993.

[86] R. Ramakrishnan, K.A. Ross, D. Srivastava, and S. Sudarshan. Efficient Incremental Evaluation of Queries with Aggregation. In *Proc. ILPS '94*, pages 204–218, 1994.

[87] E. Reingold and J. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, 1981.

[88] S.P. Reiss. A Framework for Abstract 3D Visualization. In *VL '93: IEEE Symposium on Visual Languages*, pages 108–115, Bergen, Norway, August 1993.

[89] G.G. Robertson, J.D. Mackinlay, and S.K. Card. Cone Trees: Animated 3D Visualizations of Hierarchical Information. In *ACM CHI '91*, pages 189–194. ACM, April 1991.

[90] S.F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein. Interactive Graphic Design using Automatic Presentation Knowledge. In *Proc. ACM CHI '94*, pages 112–117, Boston, MA, April 1994. ACM.

[91] S.F. Roth and J. Mattis. Data Characterization for Intelligent Graphics Presentation. In *Proc. ACM CHI '90*, pages 193–200, Seattle, WA, April 1990. ACM.

[92] L.A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and Tuan A. A Browser for Directed Graphs. *Software–Practice and Experience*, 17(1):61–76, January 1987.

[93] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[94] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1989.

[95] D. Sankoff and J.B. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[96] M. Sarkar and M.H. Brown. Graphical Fisheye Views of Graphs. In *Proc. ACM CHI '92*, pages 83–91, Monterey, CA, May 1992. ACM.

[97] M. Sarkar and M.H. Brown. Graphical Fisheye Views of Graphs. Technical Report 84, DEC System Research Center, Palo Alto, CA, March 1992.

[98] M. Sarkar and S.P. Reiss. Manipulating Screen Space with *Stretch Tools:* Visualizing Large Structure on Small Screen. Technical Report CS-92-42, Dept. of Comp. Sci., Brown U., Providence, RI, September 1992.

[99] M. Sarkar, S.S. Snibbe, and S.P. Reiss. Stretching the Rubber Sheet: A Metaphor for Visualizing Large Structures on Small Screens. In *Proc. ACM UIST '93*, November 1993.

[100] D. Schaffer, Z. Zuo, L. Bartram, J. Dill, S. Dubs, S. Greenberg, and M. Roseman. Comparing Fisheye and Full-Zoom Techniques for Navigation of Hierarchically Clustered Networks. In *Graphics Interface '93*, pages 87–96, May 1993.

[101] M.H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In *DOOD '91*, pages 189–207, 1991.

[102] B. Shneiderman. Direct Manipulation: A Step Beyond Programming. *IEEE Computer*, 16(8):57–69, August 1983.

[103] B. Shneiderman. Tree visualization with tree-maps: A 2-D space filling approach. *ACM Transactions on Graphics*, 11(1):1–39, January 1992.

[104] B. Shneiderman. Dynamic Queries for Visual Information Seeking. Technical Report CAR-TR-655 CS-TR-3022 SCR-TR-93-3, Dept. of Comp. Sci., U. of Maryland, September 1993.

[105] B. Shneiderman. Dynamic Queries for Visual Information Seeking. *IEEE Software*, 11(6):70–77, November 1994.

[106] M.C. Stone, K. Fishkin, and E.A. Bier. The Movable Filter as a User Interface Tool. In *Proc. ACM CHI '94*, pages 306–312, Boston, MA, April 1994. ACM.

[107] M. Stonebreaker. Implementation of Integrity Constraints and Views by Query Modification. In *Proc. of ACM Conf. on Management of Data*, May 1975.

[108] K. Sugiyama and K. Misue. 'Good' Graphic Interfaces for 'Good' Idea Organizers. In *Human-Computer Interaction – INTERACT '90*, pages 521–526. IFIP, Elsevier, August 1990.

[109] K. Sugiyama and K. Misue. Visualization of Structural Information: Automatic Drawing of Compound Digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, 1991.

[110] K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[111] E.R. Tufte. *Envisioning Information*. Graphics Press, P.O. Box 430, Cheshire, Connecticut, 06410, 1990.

[112] D. Turo and B. Johnson. Improving the Visualization of Hierarchies with Treemaps: Design Issues and Experimentation. In *IEEE Visualization '92*, pages 124–131, Boston, MA, 1992.

[113] A. Tversky and I. Gati. Studies of Similarity. In E. Rosch and B.B. Lloyd, editors, *Cognition and Categorization*, pages 79–98. Erlbaum, 1978.

[114] J.D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1803 Research Boulevard, Rockville, MD, 20850, 1988.

[115] J.D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 2. Computer Science Press, 1803 Research Boulevard, Rockville, MD, 20850, 1989.

[116] T. Urpi and A. Olive. A Method for Change Computation in Deductive Databases. In *Proc. of the 18th VLDB Conf.*, pages 225–237, Vancouver, BC, 1992.

[117] K. Utting and N. Yankelovich. Context and Orientation in Hypermedia Networks. *ACM Transactions on Information Systems*, 7(1):58–84, January 1989.

[118] C. Wetherell and A. Shannon. Tidy Drawings of Trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, 1979.

[119] F. Widom and S.J. Finkelstein. Set-oriented Production Rules in Relational Database Systems. In *Proc. ACM SIGMOD '90*, pages 259–270, May 1990.

[120] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing Set-oriented Production Rules as an Extension to Starburst. In *Proc. of the 17th VLDB Conf.*, pages 275–285, Barcelona, Spain, 1991.

[121] C. Williamson and B. Shneiderman. The Dynamic HomeFinder: Evaluating Dynamic Queries in a Real Estate Information Exploration System. In *Proc. ACM SIGIR '92*, pages 338–346. ACM, June 1992.

Appendix A

Graphite System Description

A.1 Introduction

The Graphite graph visualization system consists of a number of key integrated components as shown in Figure A.1. All components except for the graph editor, have been implemented by roughly 22 thousand lines of C++ [28] code. TCL (Tool Control Language) [82] was used to integrate these components by providing a consistent command-oriented interpreted front-end to the system. The graph editor was implemented by a combination of C++ and roughly 2600 lines of TCL/TK [82] – the language of the user interface tool kit built on top of TCL.

This design decision permits both users or programs to interact with the TCL interpreter to create and manage different types of objects. The system provides commands to create, destroy, configure, draw, and loop or aggregate over sets of objects, and a set of math functions that can participate in TCL mathematical expressions (see Appendix A.4).

In the following, we summarize each component; we discuss the command interface and graph layout components in more detail in the following sections.

1. *Generic utilities:*

   - 5500 lines of C++ code in 36 source files
   - includes generic C++ classes such as: dynamic searchable arrays, dictionaries, linked-lists, coordinates, dynamic strings, trees, stacks, numeric series, colours, quadtrees, etc.

2. *Hygraph/Container classes:*

   - 3500 lines of C++ code in 18 source files
   - C++ classes to represent and manipulate hygraphs and two- or three-dimensional containers

3. *Hygraph Layout Engine:*

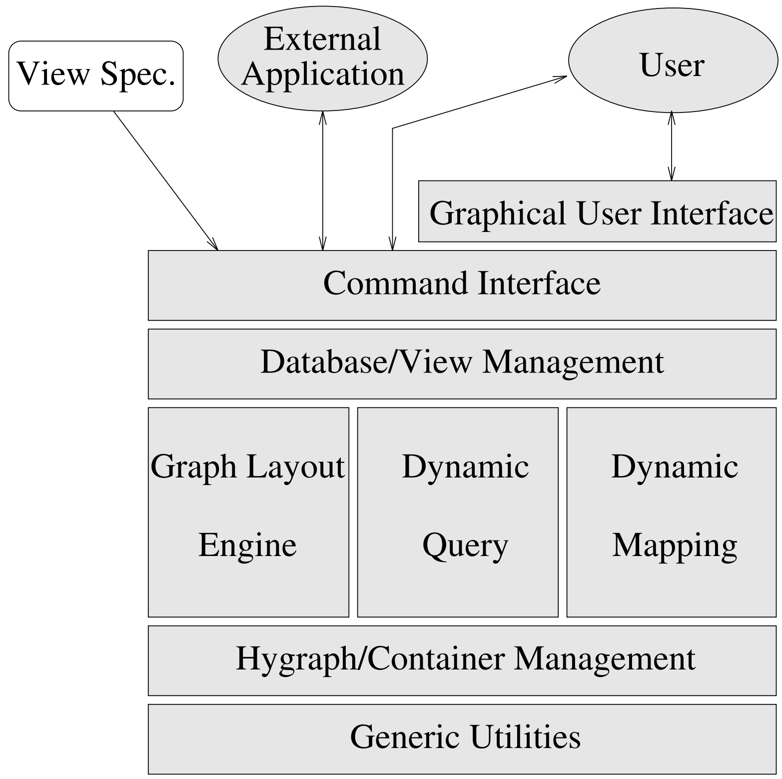


Figure A.1: The high-level architecture of the Graphite visualization system.

- 4300 lines of C++ in 21 source files
- C++ graph layout classes (see Section A.3)

4. *Dynamic Query:*

   - 500 lines of C++ code in 1 source file
   - code for evaluating dynamic selection queries

5. *Dynamic Mapping:*

   - 500 lines of C++ code in 1 source file
   - code for computing dynamic mapping settings

6. *Database/View management:*

   - 2500 lines of C++ code in 13 source files
   - code for maintaining main-memory databases [36] consisting of multiple multidimensional hygraphs
   - includes code used to maintain (insert, delete, add, remove, modify) base sets of objects (vertices, arcs, blobs) and dynamic sets of objects (focal points, path sources, and path sinks)

7. *Command Interface:*

   - 3500 lines of C++ code in 19 source files
   - TCL command-level interface – commands are used to access the underlying components

8. *Graphical User Interface:*

   - 2000 lines of C++ code in 3 source files
   - TK canvas extensions
   - 2600 lines of TCL/TK code in 9 source files
   - graphical user interface as described in Chapter 2
   - graph editor can be used to edit and browse node-and-link diagrams including hygraphs with nested blobs

A.2 Command Interface

A.2.1 Objects

The Graphite system is used to manipulate sets of the following types of objects:

1.	*hygraph*	a hygraph $h = \langle V, A, B \rangle$ [16]
2.	*vertex*	a vertex $v \in V$
3.	*arc*	an arc $a \in A \subseteq V \times V$
4.	*blob*	a blob $b \in B \subseteq V \times 2^V$
5.	*layout*	a layout object
6.	*focal point*	a focal point $f \in F \subseteq V$
7.	*path*	an object used to maintain shortest paths information
8.	*source*	a path source $s \in S \subseteq V$
9.	*sink*	a path sink $s \in S' \subseteq V$

A.2.2 Commands

Creating Objects

Objects are created with the `grCreate` command. Path objects are identified by an index value starting at 0, while the other objects are identified by unique strings. Note that `LayoutSpec` below refers to a layout algorithm specification (see Section A.3).

grCreate h Id	create a new hygraph or make existing hygraph current
grCreate v Id Label	add a vertex to current hygraph
grCreate a Id FromId ToId Label	add an arc to current hygraph
grCreate l Id LayoutSpec	create a layout object
grCreate f Id Mag	make vertex Id a focal point with magnification Mag
grCreate (s|e) Id	make vertex Id a start or end of a path
grCreate p Index (forward|reverse|undirected) (A|B) VArg AArg VExpr AExpr	compute shortest path info for current graph

Destroying Objects

Objects are destroyed with the `grDestroy` command.

grDestroy (h|v|a|b|l|f|p|s|e|i) Id	destroy object

Configuring Objects

Objects are configured with the `grConfigure` command. Note that vertices and arcs can have arbitrary numbers of numeric attributes. The current version of Graphite does not support nominal or string attributes.

grConfigure (h|b) Id layout R L	assign layout L to region R of object
grConfigure (h|v|b) Id bbox R X Y U V	set bounding box for region R to X Y U V
grConfigure (v|a) Id attr I Expr	set attribute I of object to Expr
grConfigure (v|a) I regexp (i|l) RegExpr	set attribute I to 1 if element's id or label matches regular expression RegExpr, else set to 0
grConfigure a Id arrow Mode	set link arrow mode; Mode: none, first, last, both

Object-generating Commands

The following commands provide access to objects through (pointer) navigation.

grFrom $a	the tail (vertex) of arc $a
grTo $a	the head (vertex) of arc $a
grPB $v	the parent blob of vertex $v
grPV $b	the parent vertex of blob $b

Set-generating Commands

The following commands provide access to set-valued fields.

grV	vertices
grA	arcs
grB	blobs
grF	focal points
grS	path sources
grE	path sinks
grIA $v	incoming arcs into vertex $v
grOA $v	outgoing arcs from vertex $v
grCB $v	child blobs of vertex $v
grCV $b	child vertices of blob $b
grTC $v (forward|reverse|undirected) (A|B)	transitive closure rooted at vertex $v in graph or nesting hierarchy

Relational Operators

The following commands implement the schema-preserving relational operations with object-preserving semantics.

grSelect Set Condition	selection
grUnion Set1 Set2	set union
grInters Set1 Set2	set intersection
grDiff Set1 Set2	set difference

Aggregate Operators

The following commands implement the standard aggregate operations.

grCnt Set	count
grMin Set Expr	minimum
grMax Set Expr	maximum
grSum Set Expr	sum
grAvg Set Expr	average
grProd Set Expr	product

Loops

Graphite provides a general looping mechanism. The primary use of the `grLoop` command is to compute derived expressions when used in conjunction with the `grSet` command.

grLoop Var Rel Body	loop over each element executing Body
grSet $o I Expr	set attribute I of object $o to Expr

Although *Body* may consist of an arbitrary number of TCL commands, as stated above, the primary use of the looping mechanism is to compute derived attributes with the `grSet` command. Thus it is reasonable to restrict *Body* to contain only `grLoop` and `grSet` commands. For example, the following program sets the value of vertex attribute 3 to the sum of the values of attributes 0 and 1 divided by the value of attribute 2.

```
grLoop v V {
   grSet $v 3 [expr ( a($v,0) + a($v,1) ) / a($v,2)]
}
```

Node Attributes	Link Attributes
position: relative x	link width
position: relative y	hue
position: relative z	saturation
position: absolute x	value
position: absolute y	
size: relative	
size: absolute x	
size: absolute y	
hue	
saturation	
value	
fill percentage	

Table A.1: Graphite dynamic mapping graphic properties.

Drawing Graphs

Graphs are drawn with the `grDraw` command.

grDraw v	position vertices (node layout) using previously assigned layout object(s)
grDraw a Sep	perform arc routing (link layout)

Dynamic Mapping

The following commands allow a mapping to be established dynamically between objects' numeric attributes and their corresponding presentation attributes. Binding commands are typically invoked interactively by the user, but can also be placed in a Graphite program to assign default mappings that could capture domain-specific properties of a visualization application. The Graphite graphic properties available for dynamic mapping are listed in Table A.1.

<table>
<tr><td>grMap (v|a) I A</td><td>map object attribute I to presentation attribute A</td></tr>
<tr><td>grMap (v|a) -1 A</td><td>reset previous mapping to presentation attribute A</td></tr>
</table>

Dynamic Queries

The following commands are invoked by the graphical user interface component in response to user actions related to dynamic queries and attributes-based linking.

<table>
<tr><td>grUpdate (v|a|b) I W H N</td><td>update histogram associated with attribute I; optionally set width to W height to H and number of histogram bars to N</td></tr>
<tr><td>grLink (v|a|b) Id (0|1)</td><td>link or unlink histogram bars associated with vertex, arc, or blob Id</td></tr>
<tr><td>grPaint (v|a|b) H B (0|1)</td><td>link or unlink all objects associated with bar B of histogram H</td></tr>
<tr><td>grMask (v|a|b) H B (0|1|2)</td><td>mask or unmask objects associated with bars of histogram H; 0- bar B only, 1- all bars <= B, 2- all bars >= B</td></tr>
</table>

`mag($f)`	magnification of focal point $f
`sp($u,$v,$i)`	shortest path length between $u and $v in path $i
`a($o,$i)`	attribute $i of object $o
`na($o,$i)`	normalized value of attribute $i of object $o
`x($v)`	horizontal position of vertex $v
`y($v)`	vertical position of vertex $v
`w($v)`	width of vertex $v
`h($v)`	height of vertex $v
`w()`	graph width
`h()`	graph height
`onP($o,$i)`	no. of shortest paths in path object $i that $o is on

Table A.2: Math functions added in Graphite TCL/TK extension.

Miscellaneous

The `grInfo` command can be used to query miscellaneous interpreter state information.

grInfo version	returns a string containing current version information

A.2.3 Math Functions

Additional mathematical functions are available through the Graphite TCL/TK extended interpreter as shown in Table A.2. The standard TCL math functions are listed in Section A.4.

A.3 Layout Engine

A.3.1 Architecture

The Graphite layout engine [79] generates drawings of hygraphs [16]. The engine can compute both two and three dimensional layouts, can draw both graphs (hygraphs without blobs) as well as full-fledged hygraphs, and can create drawings by employing arbitrary combinations of simple layout methods (composite graph layout).

In order to provide more flexible and powerful drawing facilities, every Graphite layout algorithm has been designed around a generic container model: a container may recursively contain (sub)containers; containers may optionally be linked by uni-directed, bi-directed or undirected arcs. The hygraph formalism is mapped onto this model as shown in Figure A.2.

In drawing hygraphs, topological blob nesting is depicted by visual containment. Given a hygraph container hierarchy *C*, a drawing of a hygraph is generated by the following simple recursive bottom-up algorithm:

```
HygraphLayout(C)
    foreach c in C
        HygraphLayout(c)
    C.position(all c in C)
```

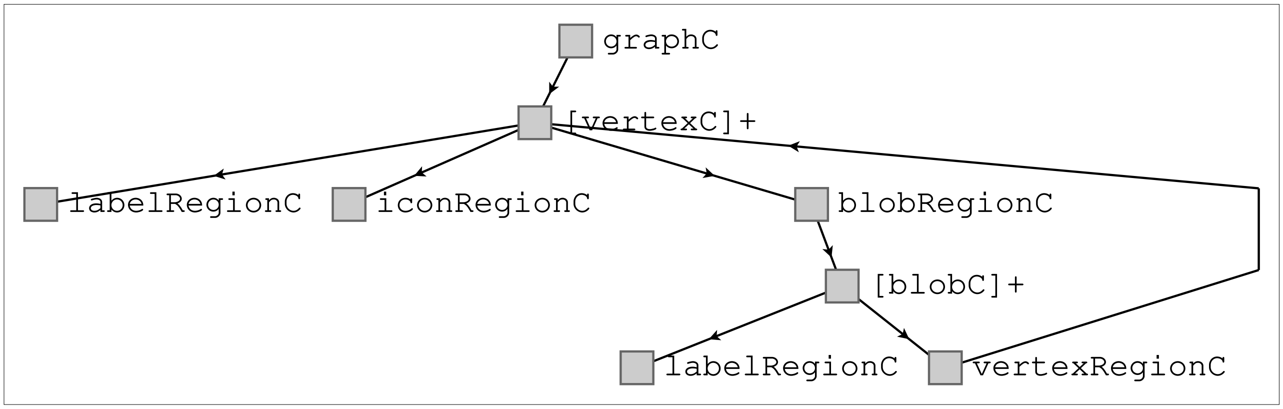


Figure A.2: Mapping the hygraph formalism onto the Graphite layout container model.

Line 4 suggests that each container *C* has an associated layout algorithm that positions its subcontainers relative to one another and computes its bounding box. Except for the top-level container, each container then becomes a subcontainer that is positioned by its parent's layout algorithm. For a leaf container (*i.e.*,, an icon or a label), the layout algorithm sets the container's bounding box to the dimensions of the associated icon or label.

This layout algorithm per container approach means that the Graphite layout engine is capable of performing composite layout at a much finer level of granularity than previous efforts [Henry91,Messinger91]. In the basic (non-composite) layout scenario, only the top-level container (*graphC*) is assigned a layout algorithm. In a composite layout scenario, in addition to the top-level container, a (possibly) unique layout algorithm can be associated with each node, blob, nodes region, and blobs region container, using the `grConfigure` command.

A.3.2 Layout Algorithms

The current version of the graph layout engine is capable of producing a drawing using arbitrary combinations of the following simple parametric layout algorithms. Note that in the following, we use %d to refer to an integer-valued constant and %f to refer to a real-valued constant.

1. average: move each container towards the average location of its neighbours. Parameters:
   - n %d – number of iterations
   - s %f – ratio of distance moved towards average
2. circle: circular layout. Containers are positioned along the perimeter of a circle; containers are ordered to reduce (but not necessarily minimize) the number of arc crossings.
3. cluster: size-based clustering layout. Containers are divided into clusters according to size and shape similarity. Each cluster is laid-out independently and then resulting drawings are combined to form a

composite. This algorithm is conceptually similar to the technique used by Henry's interactive graph layout approach [50], except that the graph is partitioned automatically by the algorithm rather than interactively by the user. Parameters:

- proportion %f – maximum tolerated ratio between largest and smallest subcontainer
- minaspect – minimize aspect ratio, not area

4. fill: space filling layout. This algorithm selects grid or stack depending on which layout produces the most compact drawing.

5. grid: grid layout. Containers are positioned at the intersections of an $m \times n$ grid. Containers are placed, one-connected component at a time, in top to bottom left to right order; containers are ordered to reduce (but not necessarily minimize) the number of arc crossings. The values m and n are computed by the algorithm to produce a drawing that is as nearly square as possible given the sizes and shapes of the individual containers.

6. hier: hierarchical layout. Initially, the algorithm computes a depth-first spanning forest F of the input graph, and places containers by following tree arcs with respect to F [4] (and ignoring forward, cross, back, and external arcs). As such, the algorithm is similar to several past tree drawing approaches [87, 118], but has been extended to perform hierarchical layout of general graphs. Containers are assigned to layers by the longest path layering technique [25]. Graphs with multiple roots and cycles can be drawn; dummy middle points are inserted in links that span multiple layers; limited crossing minimization [25, 35, 54, 73, 92, 110] is performed. Parameters:

   - x|y – make a left to right or top to bottom drawing
   - ls %f – layer separation as proportion of layer dimension

7. nop: no layout. Containers' positions are not altered. This method can be used to provide a form of incremental layout capability. A container's size is changed to fully enclose its subcontainers. Parameters:

   - center %d – center containers?

8. overlap: overlap elimination. Containers are repositioned to eliminate overlaps in a manner similar to the horizontal shuffle algorithm [24], except that containers are shuffled along both the x and y axes (and z for 3-D layouts). This practice preserves orthogonal ordering and is more successful at preserving the user's mental map [24] than the horizontal shuffle.

9. pack: quadtree-based two dimensional bin packing. Containers are placed into a small enclosing region by an incremental quadtree growing approach. Note that bin packing is NP-complete [37], so this solution necessarily involves the use of heuristics. Parameters:

   - compact %d – try compacting?

- aspect %f – desired aspect ratio (x versus y axis).

10. random: random layout. Containers are positioned randomly. Parameters:

    - scale %f – scale default bounding box

11. spring: force-based layout. Containers are positioned according to the following analogy to a physical system: containers are treated as charged bodies that repel one another; arcs are treated as springs attached to pairs of containers that pull each member of the pair towards the other. The simulation is continued until a low-energy state is reached resulting in the final layout. This algorithm is similar to several approaches described in the literature [23, 33, 59]. Parameters:

    - t %f – starting temperature
    - n %d – number of iterations
    - af %f – anchor force
    - av %f %f %f – anchor vector

12. stack: stack layout. Containers are positioned along the axis which results in the most square (cubical) or compact drawing.

13. sugi: hierarchical layout. A simple version of the Sugiyama layout algorithm [110]. Parameters:

    - x|y – make a left to right or top to bottom drawing
    - ls %f – layer separation as proportion of layer dimension
    - n %d – number of sweeps
    - sw %d – sweep direction: 1-up 2-down 3-both
    - bc %d – barycenter: same as above

14. 1d: one-dimensional layout. Containers are positioned along one of the specified axes. Parameters:

    - x|y|z – axis along which to position nodes
    - center %d – center nodes?
    - alternate %d – alternate between x and y axes?

In addition, all layout algorithms support the following generic parameters.

- 3d %d – 3-D drawings or 2-D drawings?
- fixedsize %d – preserve original sizes? (used to achieve stability in incremental drawing)
- normalize %d – normalize containers?

<table>
<tr><td><code>- ~ !</code></td><td>Unary minus, bit-wise NOT, logical NOT. Bit-wise NOT may be applied only to integers.</td></tr>
<tr><td><code>* / %</code></td><td>Multiply, divide, remainder. Remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.</td></tr>
<tr><td><code>+ -</code></td><td>Add and subtract.</td></tr>
<tr><td><code><< >></code></td><td>Left and right shift. Valid for integer operands only.</td></tr>
<tr><td><code>< > <= >=</code></td><td>Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise.</td></tr>
<tr><td><code>== !=</code></td><td>Boolean equal and not equal. Each operator produces a zero/one result.</td></tr>
<tr><td><code>&</code></td><td>Bit-wise AND. Valid for integer operands only.</td></tr>
<tr><td><code>^</code></td><td>Bit-wise exclusive OR. Valid for integer operands only.</td></tr>
<tr><td><code>|</code></td><td>Bit-wise OR. Valid for integer operands only.</td></tr>
<tr><td><code>&&</code></td><td>Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise.</td></tr>
<tr><td><code>||</code></td><td>Logical OR. Produces a 0 result if both operands are zero, 1 otherwise.</td></tr>
<tr><td><code>x ? y : z</code></td><td>If-then-else, as in C. If x evaluates to non-zero, then the result is the value of y. Otherwise the result is the value of z.</td></tr>
</table>

Table A.3: Standard TCL math operators, grouped in decreasing order of precedence.

- reorder %d – reorder nodes?
- retainlabels %d – use supplied label dimensions?
- minsep %f %f %f – minimum separation between containers
- border %f %f %f – border size around container
- fixedunits %d – are above units fixed (absolute) or proportional?
- charsize %f %f %f – default character size in labels
- elimoverlap %d – perform overlap elimination?

A.4 TCL Math Operators and Built-in Functions

The TCL math operators are listed in Table A.3, grouped in decreasing order of precedence. All of the binary operators group left-to-right within the same precedence level. For example, the command `4 * 2 < 7` returns 0.

The standard TCL mathematical functions are listed in Table A.4. TCL also implements a number of functions for conversion between integers and floating-point numbers as shown in Table A.5.

`sin`	sine	`hypot`	convert rectangular coordinates to polar
`cos`	cosine	`sqrt`	square root
`tan`	tangent	`pow`	power
`asin`	arc sine	`log`	natural logarithm
`acos`	arc cosine	`log10`	logarithm
`atan`	arc tangent	`exp`	exponential
`atan2`	polar arc tangent	`ceil`	ceiling
`sinh`	hyperbolic sine	`floor`	floor
`cosh`	hyperbolic cosine	`fmod`	floating point remainder
`tanh`	hyperbolic tangent		

Table A.4: Standard TCL math functions.

`abs(arg)`	Returns the absolute value of `arg`. `Arg` may be either integer or floating-point, and the result is returned in the same form.
`double(arg)`	If `arg` is a floating value, returns `arg`, otherwise converts `arg` to floating-point and returns the converted value.
`int(arg)`	If `arg` is an integer value, returns `arg`, otherwise converts `arg` to integer by truncation and returns the converted value.
`round(arg)`	If `arg` is an integer value, returns `arg`, otherwise converts `arg` to integer by rounding and returns the converted value.

Table A.5: TCL math conversion functions.

Appendix B

Selected Algorithms

B.1 All-Pairs-Shortest-Paths

We have modified the APSP [4, 31] to take into account vertex costs in addition to arc costs, path direction, and the possibility of multiple arcs between pairs of vertices.

```
% Input:  sets V of vertices and A of arcs, an array CV of vertex costs,
%         and an array CA of arc costs.
% Output: |V| x |V| matrix C of minimum arc costs, |V| x |V| matrix PL of
%         shortest path lengths, |V| x |V| matrix PV giving a vertex in
%         the middle of each shortest path, |V| x |V| matrix PA giving an
%         arc in the middle of each shortest path.

% reset path lengths and minimum arc costs
foreach u in V
   foreach v in V
      PL[u,v] := infinity
      C[u,v] := infinity

% compute minimum arc costs and remember arcs
foreach v in A
   if CA[a] < C[head(a),tail(a)]
      if dir = forward OR dir = undirected
         C[head(a),tail(a)] := CA[a]
         PA[head(a),tail(a)] := a
      if dir = reverse OR dir = undirected
         C[tail(a),head(a)] := CA[a]
         PA[tail(a),head(a)] := a

% initialize path lengths
foreach u in V
   foreach v in V
      PL[u,v] := CA[u,v] + CV[u] + CV[v]
      PV[u,v] := -1
   PL[u,u] := CV[u]

% compute shortest path lengths
foreach w in V
   foreach u in V
      foreach v in V
         if PL[u,w] + PL[w,v] - CV[w] < PL[u,v]
            PL[u,v] := PL[u,w] + PL[w,v] - CV[w]
            PV[u,v] := w
```

B.2 Incremental Path Maintenance

The `UpdateSource` and `UpdateSink` procedures update the `npaths` count for each vertex or arc that is affected by an addition or removal of a path source or sink.

```
% Inputs:  incr = 1 -> adding source/sink; incr = -1 -> removing source/sink

UpdateSource( u, incr )
   if incr = 1
      S := S + {u}
   else
      S := S - {u}

   foreach path p
      foreach v in E
         if PL[u,v] < infinity
            UpdatePath( u, v, p, incr )
            npaths[p](u) += incr     % UpdatePath misses initial source

UpdateSink( v, incr )
   if incr = 1
      E := E + {u}
   else
      E := E - {u}

   foreach path p
      foreach u in S
         if PL[u,v] < infinity
            UpdatePath( u, v, p, incr )
            npaths[p](u) += incr     % UpdatePath misses initial source

UpdatePath( u, v, p, incr )
   w = PV[u,v]
   if w = -1     % Direct path: u -> v; mark a[u,v] and v
      a := PA[u,v]
      npaths[p](a) += incr;
      npaths[p](v) += incr;
   else          % Intermediate path: u -> w -> v
      UpdatePath( u, w, p, incr )
      UpdatePath( w, v, p, incr )
```

B.3 Longest-path Layer Assignment

The following algorithm computes a longest-path layering of a graph [72] (page 45).

```
% Input:  graph G, root vertex r
% Output: longest-path layer assignment l(v) for each v in G(V)
LongestPathLayer(G,r)
   TopSort(G,r)
   foreach v in G(V)
      l(v) := 0
   Sort G(V) in increasing rt(v) order   % reverse topological order
   foreach v in G(V)
      foreach u in IA(v)
         l(u) := max(l(u),l(v)+1)
```

B.4 Topological Sort

The following algorithm assigns the reverse topological sort number to each vertex in a DAG [4].

```
% Input:  Directed acyclic graph G, root vertex r
% Output: a reverse topological sort number rt(v) for each v in G(V)
TopSort(G,r)
   n=0
   foreach v in G(V)
      v.visited := false
   TopSort(r)

TopSort(v)
   v.visited := true
   foreach w in OA(v)
      if not w.visited
         TopSort(w)
   rt(v) = n++
```

B.5 Incremental Evaluation of Aggregate Expressions

Given an aggregate expression $\mathcal{F}$ AgOp $R(A_i)\,(R)$, it is desirable to be able to compute the value of the expression incrementally as tuples are added or deleted from R, or the values of attribute $R(A_i)$ are modified. In what follows, we use $AgOp_0$ to refer to the value of the aggregate expression prior to the update, as in min_0, and v to the value of attribute $R(A_i)$ in the tuple that is being inserted or deleted. Initially, sum, $prod$, avg, and cnt expressions are set to zero, min to some system-defined maximum positive floating point value, and max to a system-defined maximum negative floating point value. For a more general discussion of incremental evaluation of queries with aggregation see [86].

To insert r into R:

- $min \leftarrow \begin{cases} \text{if } v < min_0 \text{ then} & v \\ \text{else} & min_0 \end{cases}$
- $max \leftarrow \begin{cases} \text{if } v > max_0 \text{ then} & v \\ \text{else} & max_0 \end{cases}$
- $sum \leftarrow sum_0 + v$
- $prod \leftarrow prod_0 \cdot v$
- $cnt \leftarrow cnt_0 + 1$
- $avg \leftarrow ((avg_0 \cdot cnt_0) + v) \div (cnt_0 + 1)$

To delete r from R:

- $min \leftarrow \begin{cases} \text{if } v = min_0 \text{ then} & MIN(R(A_i)) \\ \text{else} & min_0 \end{cases}$
- $max \leftarrow \begin{cases} \text{if } v = max_0 \text{ then} & MAX(R(A_i)) \\ \text{else} & max_0 \end{cases}$

Operator	*Insert*	*Delete*	*Modify*
MIN	1	$1, n$	$1, n$
MAX	1	$1, n$	$1, n$
CNT	1	1	1
SUM	1	1	1
$PROD$	1	$1, n$	$1, n$
AVG	1	1	1

Table B.1: Time complexity of incremental updates to aggregate functions; where two figures are given, the first corresponds to average-case and the second to worst-case.

- $sum \leftarrow sum_0 - v$
- $prod \leftarrow \begin{cases} \text{if } v = 0 \text{ then} & PROD(R(A_i)) \\ \text{else} & prod_0 \div v \end{cases}$
- $cnt \leftarrow cnt_0 - 1$
- $avg \leftarrow ((avg_0 \cdot cnt_0) - v) \div (cnt_0 - 1)$

Finally, to modify the value $R(A_i)$ for some $r \in R$, we apply the above deletion operation to r followed by an insertion of r with $R(A_i)$ set to the modified value.

Table B.1 summarizes the time complexity of evaluating updates to aggregate functions incrementally. All updates with the exception of deletions against a MIN, MAX, or $PROD$ aggregate expression can be incrementally evaluated in $O(1)$ time. The latter can also be evaluated in constant time when the deleted value is not the actual extremum (or zero for $PROD$); in the situations where it is, however, the new extremum (or $PROD$) must be recomputed from scratch requiring linear time. Note, however, that the situation where an actual extremum is deleted should arise far less frequently than a deletion of an intermediate value. Similarly, we might assume that in the situations where $PROD$ is used, attribute values will typically be non-zero.

B.6 Incremental Evaluation of Set-Valued Expressions

Here we show that any relational algebra expression involving the schema-preserving operators: union, intersection, difference, and selection, has a simple incremental form. That is, given two input sets, there is a simple expression for updating the result following an insertion or deletion into either input set. We assume that sets are maintained using a data structure such as a balanced binary tree, that can perform membership tests in $O(\log n)$ time.

Union. Given $T \leftarrow R \cup S$:

- Insert r into R: $T \leftarrow T \cup \{r\}$
- Delete r from R: $T \leftarrow T - (\{r\} - S)$

Intersection. Given $T \leftarrow R \cap S$:

- Insert r into R: $T \leftarrow T \cup (S \cap \{r\})$

- Delete r from R: $T \leftarrow T - \{r\}$

Difference. Given $T \leftarrow R - S$:

- Insert r into R: $T \leftarrow T \cup (\{r\} - S)$
- Delete r from R: $T \leftarrow T - \{r\}$
- Insert s into S: $T \leftarrow T - \{s\}$
- Delete s from S: $T \leftarrow T \cup (R \cap \{s\})$

Selection. Given $T \leftarrow \sigma_F(R)$:

- Insert r into R: $T \leftarrow T \cup \sigma_F(\{r\})$
- Delete r from R: $T \leftarrow T - \{r\}$

It can be easily seen that since each of the above incremental update operations performs at most two membership tests, each operation executes in logarithmic time.