Challenges in Graph-Based Relational Data Visualization

Emanuel G. Noik
noik@db.toronto.edu

Department of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario, Canada M5S 1A4

Abstract

During recent years we have witnessed a growing trend toward the use of visual interfaces to view and query databases. The graph topovisual formalism is particularly well-suited for depicting relational data. The vertices of a directed graph represent a set of entities, while arcs represent relationships among the entities. This paper studies the functional requirements of a hypothetical graph visualization facility (GVF) by surveying past work in related areas and by describing challenging problems that must be addressed before such a system can be realized. Our contribution, therefore, is twofold: we present a unique holistic approach, which, in turn, gives rise to a number of interesting research problems.

1 Introduction

A glimpse at recent computer science literature reveals the wide-spread use and popularity of the term *visualization*. The availability of inexpensive high-resolution display devices has made it possible to present information in a variety of pictorial, diagrammatic, graphical, and otherwise nontextual ways.

Visualization is many things to many people: *visual programming* uses nontextual symbols to compose computer programs; *program visualization* shows the structure of programs and algorithms through pictures and animations; *scientific visualization* represents continuous data such as that encountered in weather analysis and prediction or in complex industrial processes; *quantitative data visualization* formats discrete numerical data into diagrams such as pie and bar charts commonly used to display data in business applications.

In this paper, we study the problems associated with the visualization of relational data. The following scenario illustrates our goals and motivates the discussion:

> *We are given a large relational database and we are told to analyze it (for example, the database may contain airline flight information — airlines, cities, flights between cities, flight times; or information about a large software system — modules, functions, variables, containment relations, calls relations). We can measure our success by how many* interesting *properties we can discover. Because the database is large and complex we realize that conventional text-based reports and SQL-style queries are insufficient; instead, we require some form of visualization facility to make our search more efficient. Because we know that graphs can be used to represent sets of entities and relationships among individual entities, we decide to exploit the graph topovisual formalism to ease our task. After some deliberation, we conclude that to carry out our assignment, we must perform the following tasks: extract a graph from the database, filter the database graph to obtain specific subgraphs, determine more compact* abstract *instances of large graphs, pose queries against the database using a visual or graphical language, save and retrieve graphs from secondary storage, obtain a nicely formatted drawing of a graph given its topology; and explore an abstracted graph to recover its hidden features.*

Having identified the relevant set of tasks, we observe that it may be beneficial to view these tasks as an integrated set of tools: a graph visualization facility. Having adopted this viewpoint we identify several constraints that could be exploited to simplify each of the tasks.

The paper proceeds as follows. Relevant theoretical background on common graph formalisms and relational databases is provided. A sample relational database that is cited in subsequent examples is introduced. Each of the graph visualization tasks is described and, where applicable, previous work is outlined. The paper is concluded with a partial list of open problems that cover issues specific to each task and issues that arise as a consequence of integration.

2 Background

2.1 Preliminaries

Over the years, numerous approaches for visualizing non-quantitative, structural, set-theoretical, and relational information have been proposed. Not surprisingly, the diagrammatic paradigms employed are essentially topological rather than geometrical in nature. The term *topovisual* has been used to describe these formalisms [26]. Two of the best known topovisual formalisms are the *graph* and the *Venn diagram*. Both have their roots in the work of Swiss mathematician Leonhard Euler (1707–1783).

A graph[1], in its most basic form, is simply a set of vertices connected by edges (undirected) or arcs (directed) and is commonly used to represent a (single) set of elements S and a binary relation R on S. Graphs can be modified to support several types of vertices and edges, each representing different types of elements and relationships.

Bondy and Murty provide an excellent introduction to graph theory [5]. The basic data structures and algorithms for manipulating graphs are outlined in [1, 45].

A general introduction to database theory can be obtained from [47]. A *data model* in database theory is a mathematical formalism that consists of two parts: a notation for describing data, and a set of operations used to manipulate that data. The relational data model, first described by Codd [11], is one of the most popular data models in use today. Another well-known data model is the entity-relationship (ER) model [9]. This model has the advantage over most other data models in that it allows the description of the conceptual scheme of an enterprise to be specified independently of efficiency considerations of physical database design. The design of the conceptual scheme is captured in an ER diagram that is usually transformed into a different data model (often the relational data model) upon which actual database systems are implemented.

2.2 Visualizing Databases

A natural application of data visualization occurs in the database area in which graphical views of data complement ordinary textual views. While the conceptual database scheme can be visualized as an ER diagram (drawing conventions and layout algorithms for ER diagrams are described in [3, 44]), the relational data consisting of entities and the relationships among them can be depicted by the vertices and edges, respectively, of a labeled graph. Edges are used to represent nondirectional relationships between two entities. For example, the *sibling* relationship can be shown graphically as an edge between the vertices that represent the two siblings. Directional relationships are represented by arcs. For example, the *parent* relationship is shown as an arc leaving the vertex that represents the parent and terminating at the vertex that represents the child. Multiple relationships can be represented within a single graph by displaying the name of each relationship as the label of an edge or arc, or by using different colours or line styles to render the edges or arcs associated with each relationship. For example, it is easy to show both the sibling and the parent relationships among a group of persons. Furthermore, multiple entity types can also be represented in a single graph. This can be accomplished by using appropriate labels (*e.g.,* `person(Bob)` instead of `Bob`), or by drawing vertices in a different fashion (for example, using different colours, sizes, or shapes). Each extension tends to introduce additional visual complexity that may diminish the overall effectiveness of the graph as a topovisual formalism.

Graphs have also been used to represent database queries: the G+ system [14, 15], for example. In this context, graphs form the basis of a visual query language. Graphlog is a query language that evolved from the G+ project and can be used to pose recursive queries in a natural intuitive fashion [12]. Several other systems allow queries for ER databases to be specified graphically [6, 16]: the graphical query is based on the ER diagram, however, the answer is displayed textually. In Graphlog, both the query and the answer are represented as a graph, but Graphlog does not incorporate an explicit ER schema. Work on the 4Thought toolset at IBM Canada [40, 13] combines these approaches by introducing an explicit ER schema into the G+/Graphlog framework.

Mackinlay has studied the problem of creating effective visualizations of relational and quantitative data, although his emphasis was on finding the most appropriate diagrammatic technique for a specific problem instance (for example, bar chart, pie chart, xy-plot) [30].

The use of three-dimensional graphics for visualizing complex databases has been investigated at Bellcore [7], MCC [19], and Xerox PARC [8, 10, 31, 38].

3 A Simple Example

Consider a relational database that contains three types of entities (*subjects, authors*, and *documents*) and two relationships (a document is *written_by* an author, and a subject is *discussed_in* a document). A listing of a sample database is given in the Appendix; its corresponding database graph is shown in Figure 1. The contents of this database reflect the articles cited in this paper. As demonstrated by the database graph, each of the documents addresses, at most, two subject areas. This fact lends further support to our claim that our approach, which integrates all of these areas,

[1] In this setting, the graph is regarded as a topovisual rather than a mathematical formalism.

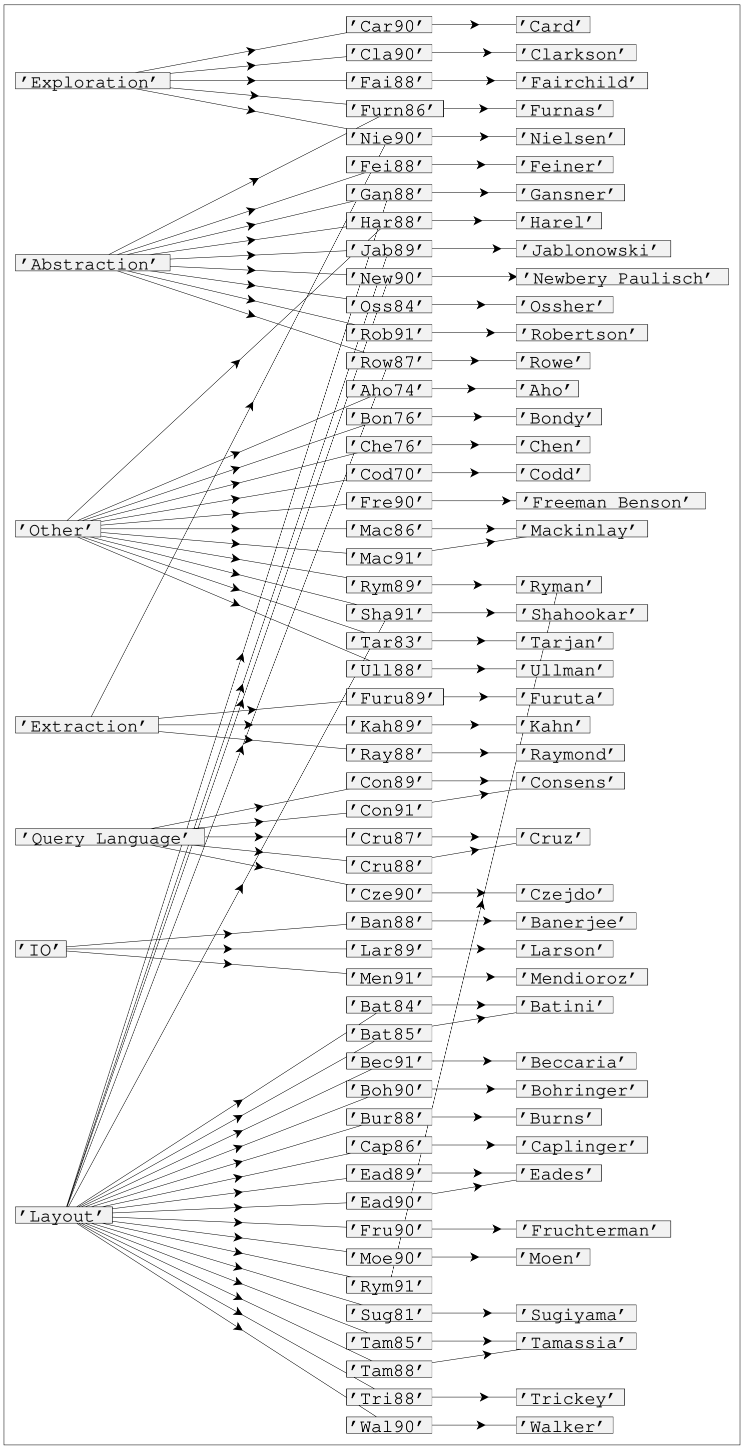

Figure 1: **Graph extraction.** The database graph extracted from the sample database. Edge labels are suppressed.

is unique.

4 Graph Visualization Facility

As suggested in the Introduction, a graph visualization facility is a system that integrates the following tasks for graphs: extraction and filtering, storage and retrieval, abstraction, queries, layout, and exploration. The G+ system [15] is one example, although it does not attempt to solve many of the problems that we discuss later. Recent work on graph browsers and editors also addresses several of these areas [39, 25, 27, 34].

In the remainder of this section, we define the aforementioned tasks.

4.1 Graph Filtering and Extraction

A **(simple) graph extractor** is given a database and a set of extraction criteria as input, and produces a database graph as output.

Example 4.1 *In the sample database,* subjects, documents, *and* authors *are represented as vertices. An arc from a document to a vertex represents an instance of the* written_by *relationship; an arc from a subject to a document represents an instance of the* discussed_in *relationship. Figure 1 shows the database graph extracted from the sample database.*

A **graph filter** is given a graph and a set of filtering criteria as input, and produces a subgraph that satisfies the criteria as output. Graph filtering is used to extract specific graph elements from the source graph.

Example 4.2 *Figure 2 shows a graph that specifies a simple filtering criterion and a graph that is the result of applying this filter to the database graph.*

While often the entire database graph is sought[2], graph extraction can also incorporate a set of filtering criteria to select a subset of the database graph. A naive, inefficient, but simple approach is to extract the entire database graph and then to filter it with the required criteria[3].

Most modern databases contain several types of metainformation in addition to data. An **extended graph extractor** is given a database, a database schema, and a set of extraction criteria as input and produces a database graph and an ER graph as output.

[2] This filtering operation returns the set of all vertices and arcs defined by the extraction criteria.

[3] If we assume that the complete database graph must be extracted at least once, we can subsequently rely on this graph and use only filtering operations.

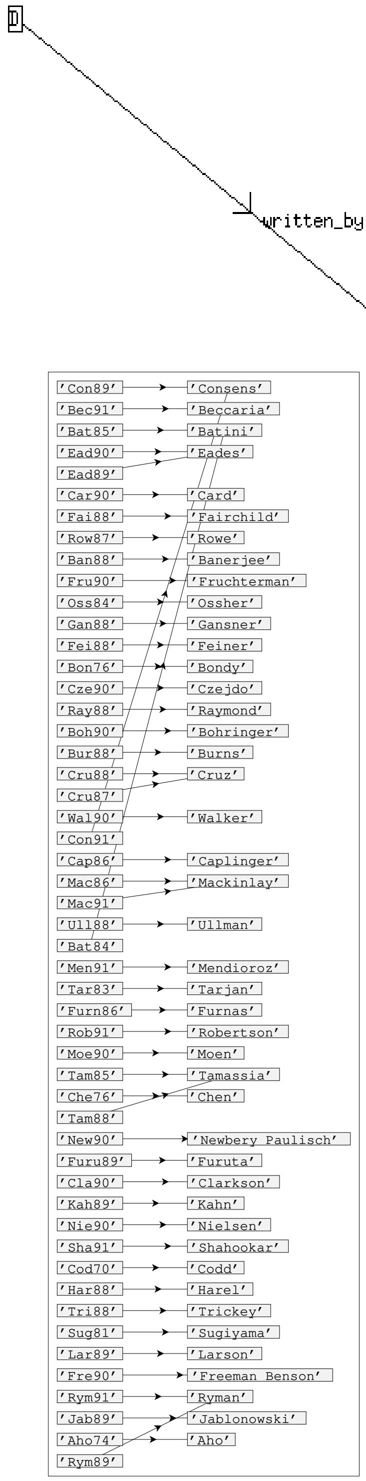

Figure 2: **Graph Filtering.** A graph that specifies a simple filtering criterion and the result of applying this filter on the sample database graph. Edge labels are suppressed.

Database structural design information such as the set of functional dependencies can be used in the other graph visualization tasks. For example, functional dependencies can be used in graph layout to improve the quality of the resulting layouts by creating additional constraints based on the nature of the dependencies. An extended graph extractor as defined previously should extract this type of information.

A problem that is related to graph extraction is the automatic construction of hypertext networks from existing text. While *objective links* can be derived from the structure of the text, *subjective links* must be added manually [28, 24]. The Oxford English Dictionary is one of the largest texts to be converted to hypertext format [37].

4.2 Graph Storage and Retrieval

A **graph server** provides two services: it stores a large graph in secondary memory and it retrieves portions of a previously saved graph. We assume that the graphs in question are too large to fit in primary memory and must be retrieved in parts. Graph retrieval consists of a sequence of graph requests, each one loading a portion of the graph. We can study the properties of different retrieval strategies by studying the corresponding sequences of retrieval requests. In general, we would like to minimize the number of secondary memory accesses required to retrieve the graph in the average case. Because only a portion of the graph can be held in primary memory, a request may cause a previously loaded portion to be *paged out* of memory; sequences of requests that give rise to many page faults can cause *thrashing*: the situation in which portions of the graph are continually swapped in and out of memory. Thus, the design goals of a graph server can be defined as follows: *to determine how a large graph should be saved in secondary memory to minimize the number of secondary memory accesses required for retrieval and to reduce the probability of thrashing*. Mendioroz provides a survey of past work on this problem and one solution [32].

4.3 Graph Abstraction

The purpose of graph abstraction is to reduce the size (number of vertices) and the complexity (number of edges, edge crossings, and so on) of a graph in a way that minimizes the loss of specific graph characteristics, or that ensures that the resulting abstracted graph exhibits specific characteristics. This section is devoted to a discussion of the previous work in this area.

4.3.1 Clustering

A common abstraction technique is *clustering*: the set of vertices is partitioned into subsets or *clusters*, and each cluster is replaced by a single vertex in the resulting graph. The representative vertex was called a *proxy* in an early paper by Sugiyama [43]. The vertex partitioning strategies define the type of clustering to be performed. Two common partitioning strategies in graph theory are the blocks and the (strongly) connected components of the graph.

4.3.2 The Grid

As the size and complexity of computer programs grows, so does the need for effective software structuring techniques to aid their designers and implementors. A software structuring mechanism called the *grid* is described by Ossher [36]. While standard techniques such as *static nesting* and *visibility control lists* can be used to specify hierarchical structures and interactions between modules in an amorphous collection, the grid is especially suited for describing the structure of *layered, object-oriented* programs. Layered graphs provide a natural and convenient model for representing the structure of such programs. In a layered graph, each layer corresponds to an abstraction level that contains a view of a collection of objects from a single point of view; a slice through the layered graph is a view of a single object from many points of view. The grid mechanism, based on this model, can specify arbitrary layered graph structures. Four techniques are used to reduce the complexity of layered graphs: *factorization, clustering, deviation*, and *approximation*. The techniques of factorization and clustering involve collecting related nodes and then treating all nodes in the collection uniformly. The interactions between individual nodes are replaced by interactions between collections. Deviation is the explicit specification of the manner in which individual nodes deviate from the assumed uniform behaviour of collections. Using factorization and clustering, a clear, simple overview can be presented. Deviation provides the specification of how the actual structure differs from the overview. The technique of approximation allows the specifications of deviations to be ignored when not needed, and gives rise to approximate specifications of structure that are simpler yet sufficiently accurate.

4.3.3 Graph Browser

The need for graph abstraction facilities has also been recognized in several articles describing graph browsers and editors. In an article describing Graph Browser (GRAB) [39], the authors observed a property common in many *function call graphs* [4]: generic utility functions are called by many functions. The associated graph, therefore, has several vertices whose in-degree is proportionally much higher than that of other vertices. The same point is noted

[4] Each vertex represents a function; an arc is drawn from vertex X to vertex Y if function X calls function Y.

in an article describing DAG [25]. However, in both situations the authors propose only an ad-hoc solution: they suggest that such vertices should be suppressed explicitly rather than identified automatically by a graph abstraction facility.

4.3.4 Graph Manager/Browser

Developed at the University of Illinois at Urbana-Champaign, Graph Manager/Browser (GMB) comprises two components: the Graph Manager that implements an abstract graph data type, and the Graph Browser that handles the details of displaying a subgraph of a graph created through the Graph Manager [27]. The Graph Browser displays graph *views* (analogous to views in the database setting), a graph view being a subgraph of its parent graph. Several graph views can be defined for a single parent graph. The abstract graph data type comprises a set of nodes, a set of arcs, node groups, states, graph annotations, and auxiliary data. In addition, an arc can be one of three types: directed, nondirected, and ordered. A *node group* is a set of nodes that is treated as a single entity by the application; node groups are recursive in that a member of a node group can be a node group. Each node and arc in a graph maintains an associated state that serves two purposes: to represent a user-definable characteristic, and to determine *view attributes* (including visibility). These mechanisms are used to control the level of graph abstraction; however, the issues of how node groups are selected and how visibility is controlled are not addressed.

4.3.5 Extensible Directed Graph Editor

Extensible Directed Graph Editor (EDGE) is an editor kernel for the direct and visual manipulation of graphs [34]. Two techniques are proposed to create graph abstractions: multilevel subgraph abstraction and edge concentration. A *multilevel subgraph abstraction* is used to group a subset of the nodes and edges. It may be viewed in three ways: black-box view (display the entire subgraph as a single, small icon); grey-box view (display the subgraph, but terminate external edges at the subgraph's bounding-box); white-box view (draw a bounding box around the subgraph). Subgraphs can contain subgraphs; EDGE calculates the layout of each subgraph independently by ignoring edges that have an external end point. While the multilevel subgraph abstraction is used to reduce the number of visible nodes, *edge concentration* reduces the number of edges. Edge concentration is based on a clustering technique that finds sets of directed edges in which all source nodes are connected to all target nodes. Each set (edge concentration) is then replaced by a special node whose fan-in is the common set of source nodes and whose fan-out is the common set of target nodes. Indirection is used to reduce the number of edges without sacrificing connectivity information.

4.4 Graph Query Language

A graph query language can be used to pose queries on graphs. For example, we may want to find all instances of a particular type of subgraph in a graph. Graphlog is a visual query language whose queries are expressed as labeled multidigraphs [12]. Graphlog expressions are graphs whose edges are labeled by regular expressions that must be satisfied by paths in the source graph. The answer to a Graphlog query is the set of all subgraphs of the source graph that match the patterns specified by the query.

Example 4.3 *Figure 3 shows a simple graph query and the result of applying it on the example database graph.*

4.5 Graph Layout

The standard two-dimensional graph layout problem is defined as follows: for graph $G(V, E)$, assign a Cartesian coordinate (x, y) to each vertex $v \in G(V)$. In general, the quality of a layout is directly related to the amount of time that an algorithm is given to compute it. Most graph layout algorithms can be divided into two groups: algorithms that emphasize performance and algorithms that emphasize layout quality. In the context of a graph visualization facility (an interactive system), we are interested primarily in algorithms that emphasize performance.

In their survey paper, Eades and Tamassia discuss graph-drawing approaches; each approach can be characterized by the classes of graphs that it can draw and the layout conventions that it implements [18].

Example 4.4 *The importance of effective graph layout algorithms is illustrated in Figures 4 and 5, in which the sample database graph shown in Figure 1 is drawn using very primitive layout algorithms.*

The remainder of this section describes some common graph layout algorithms.

4.5.1 Drawing Trees

The layout of **trees** has received much attention, probably because of the many applications of trees. Trees are drawn either as **rooted trees** or as **free trees**. Although rooted trees are one of the simplest classes of graphs to draw, the layout problem for them is highly nontrivial: many subtleties must be considered. For example, several well-known algorithms fail to position interior subtrees correctly; an improved solution is described in a recent paper by Walker [48]. Free

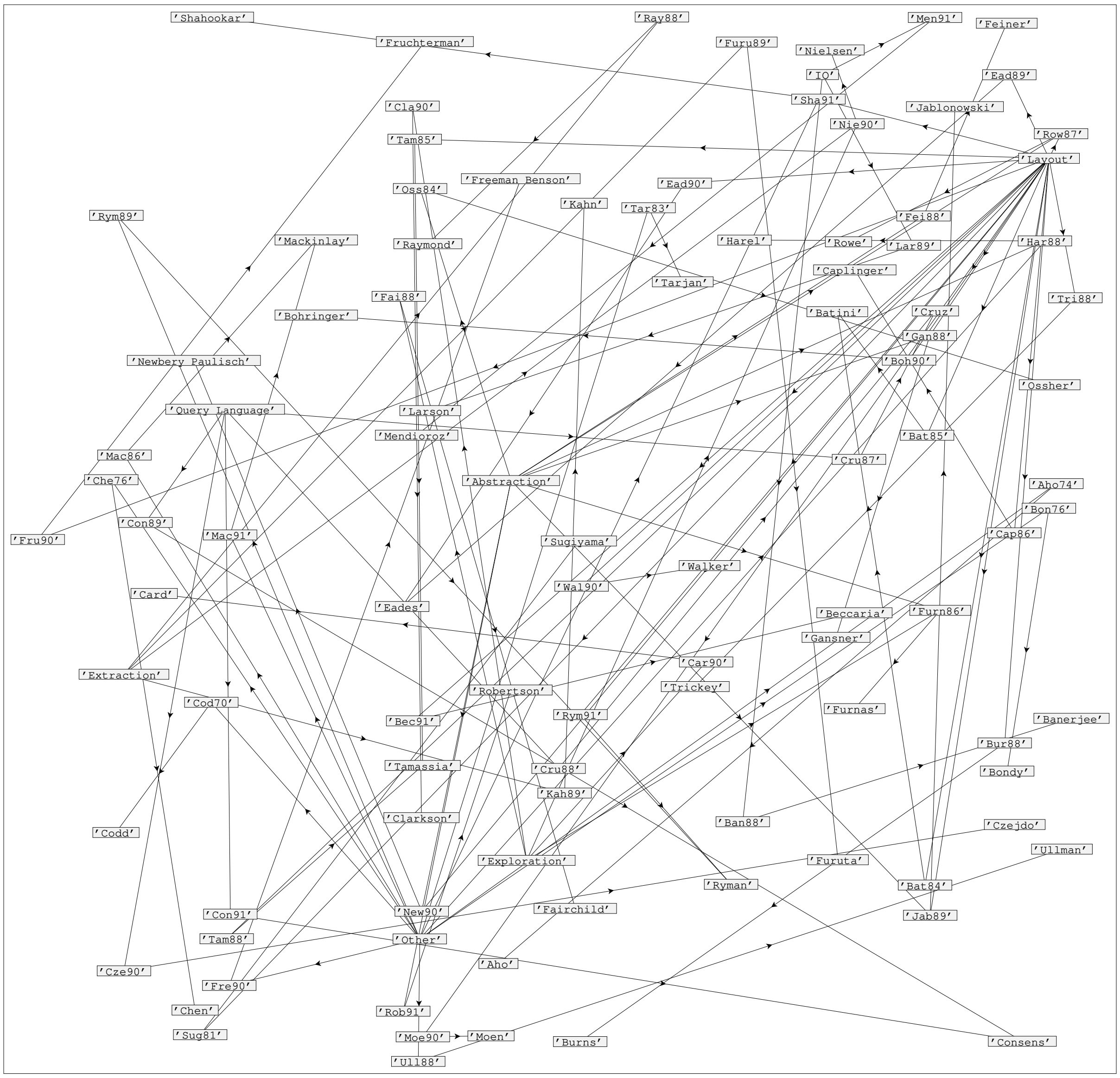

Figure 4: **Graph layout.** The sample database graph drawn using a random layout algorithm. Edge labels are suppressed.

trees are commonly shown as **radial drawings**. In this approach, vertices are arranged on concentric circles about the graph-theoretic center of the tree.

4.5.2 Drawing Planar Graphs

A graph is *planar* if it admits a planar drawing (a drawing in the plane that has no edge crossings). The **planar graph** plays an important role in graph layout: planar drawings are aesthetically pleasing. By adding a dummy vertex at each crossing, the problem of drawing a nonplanar graph with specified edge crossings can be reduced to drawing a planar graph. For these reasons, several planarity testing and layout algorithms have been devised. Two common graphic standards for drawing planar graphs are the **straight-line drawing** (each edge is a line segment) and the **orthogonal grid drawing** (each edge is mapped onto a chain of horizontal and vertical line segments). A **convex drawing** is a planar straight-line drawing in which every face is drawn as a convex polygon.

4.5.3 Drawing Directed Graphs

Digraphs are often classified into **directed acyclic graphs** (DAGs) and **general** or **cyclic digraphs**. While early algorithms were typically restricted to the class of DAGs, newer

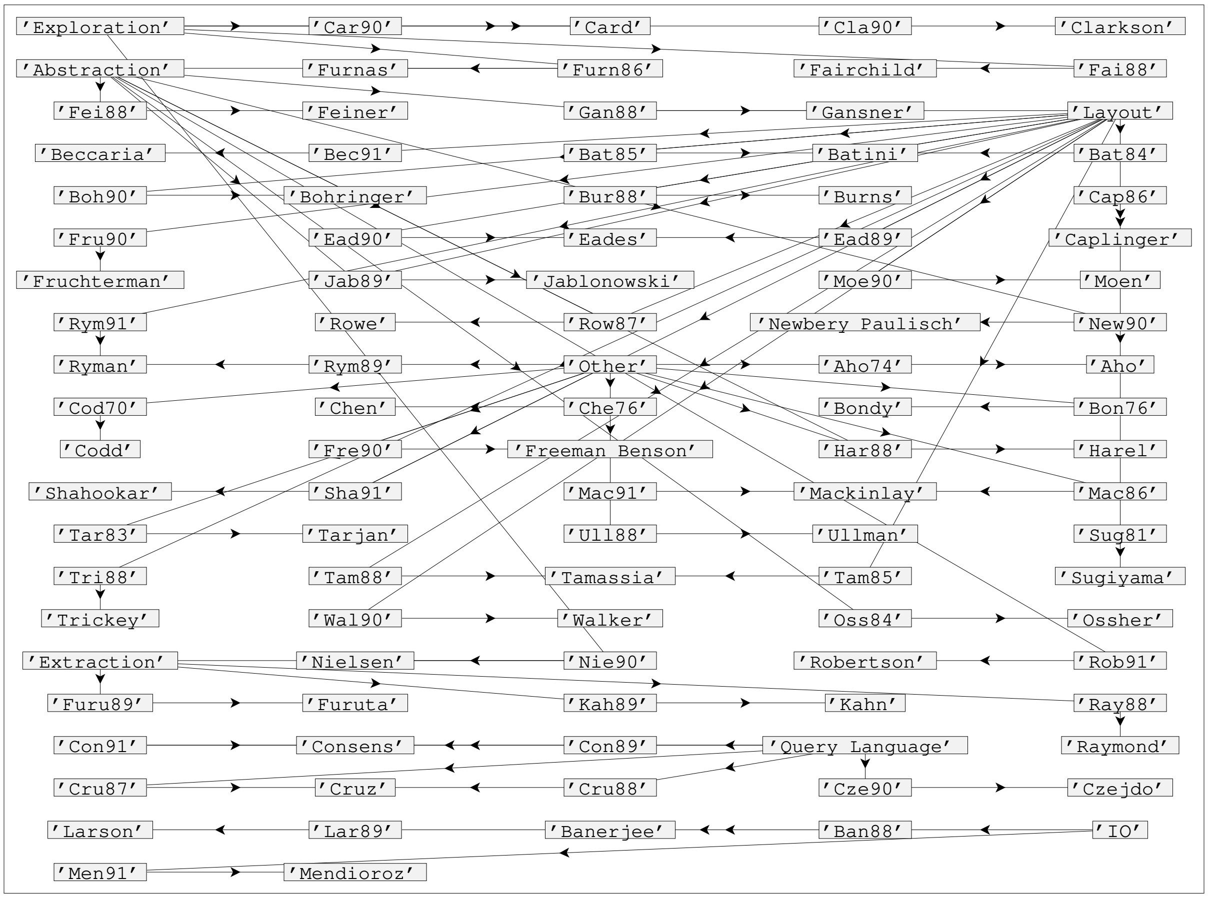

Figure 5: **Graph layout.** The sample database graph drawn using a grid layout algorithm that uses connected components to order vertices. Edge labels are suppressed.

algorithms can handle both types of digraphs. A practical and general method for drawing directed graphs based on certain aesthetic criteria is outlined in [17]. The criteria are viewed as the goals for solving optimization problems; each step of the general method aims to achieve one of the criteria by solving these optimization problems. Four criteria are outlined: arcs pointing upward should be avoided; nodes should be distributed evenly over the page; there should be as few arc crossings as possible; and arcs should be as straight as possible. The general method uses a *successive refinement* technique that refines the layout with respect to each criterion in turn. Several layout algorithms for directed graphs are shown to be instances of this general method.

4.5.4 Drawing General Undirected Graphs

Several aesthetics for drawing **general undirected graphs** have been identified. These include: display symmetry, edge crossing minimization, (edge) bend minimization, uniform edge length, and uniform distribution of vertices. General undirected graphs are most often shown as straight-line and polyline drawings. Several algorithms are based on physical models: vertices are replaced by rings and edges are replaced by springs; springs attract distant rings and repel proximate ones [22]

4.5.5 Dynamic Layout Algorithms

Dynamic layout algorithms support the insertion and deletion of vertices and edges, and can compute efficiently a partially updated layout (rather than recomputing the entire layout from scratch). Ideally, a graph update should cause only small local changes in the updated layout; a layout algorithm is *unstable* if a small change in graph topology can cause a large change in the resulting drawing. The problem of drawing dynamic trees has been studied by Moen [33].

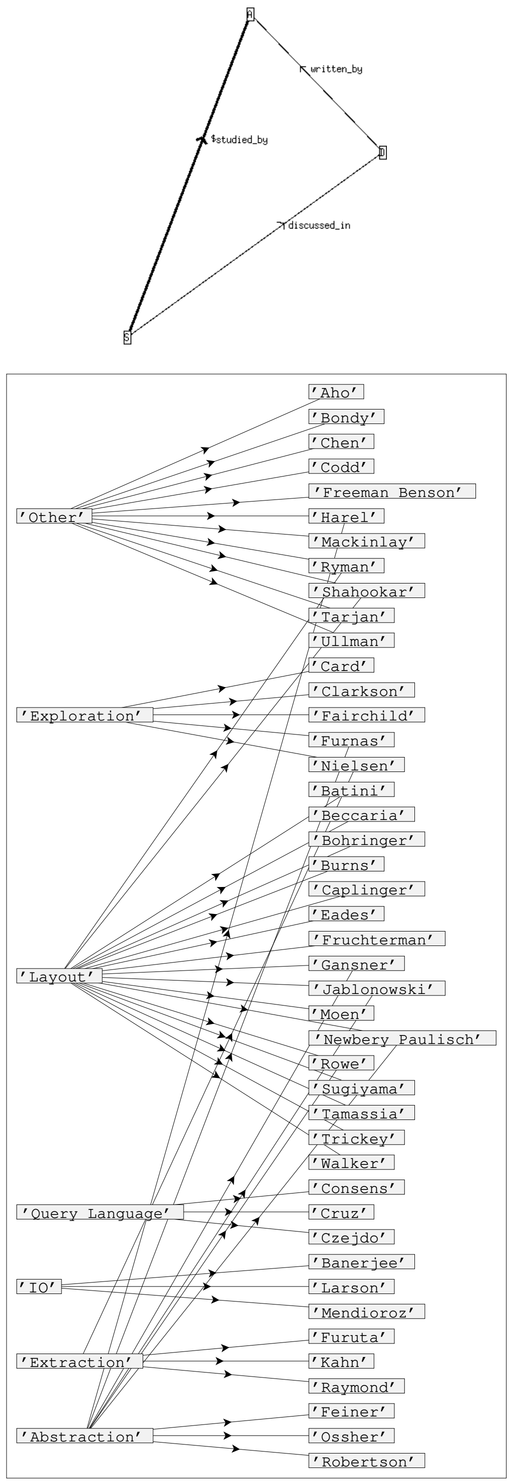

Figure 3: **Graph query.** A graph that specifies a Graphlog query and the result of applying this query on the sample database graph. Edge labels are suppressed.

4.5.6 Constraint Satisfaction Techniques in Graph Layout

Constraint satisfaction programming has gained popularity in recent years as a useful programming paradigm for expressing many complex optimization problems. A survey of the state of the art with an emphasis on incremental constraint satisfaction algorithms can be found in [21]. Recent work has shown that constraints can be incorporated in graph layout to improve the quality of the layouts and to increase layout stability in the presence of graph updates [4, 34].

4.5.7 VLSI Component Placement

An area of research that is closely related to graph layout is VLSI component placement. Computer-aided design tools have made it possible to automate the entire layout process that follows the circuit design phase in VLSI design. This came about for mainly two reasons: the use of gate array and standard cell design styles, and the creation of efficient software packages for automatic placement and routing. Module placement is an NP-complete problem and, therefore, cannot be solved exactly in polynomial time. To search through many candidate placement configurations efficiently, a heuristic algorithm must be used. A recent survey paper of past and current work in this area is in [42]. Unfortunately, most of the work in the area cannot be directly applied in interactive graph layout: in the cell placement task, layout quality is more important than runtime performance, because the quality of the layout directly affects the space-time performance of the resulting chip. Because many chips are usually manufactured from a single blueprint, it is wise to invest more time to obtain a superior layout. This quality-performance trade-off was observed elsewhere [39, 46].

4.6 Graph Exploration

Graph exploration is the interactive process that the user undertakes to examine or recover the underlying structure of the data. Exploration consists of several related activities: *change of perspective*, *navigation*, and *abstraction control*. The differences between these are illustrated in the following camera analogy[5]:

> *Liken the process of exploring a graph to the process of taking photographs with a still camera. You are vacationing in southwestern United States and find yourself at the Grand Canyon [*this is the large graph you will be exploring*]. Faced with the awesome scale of this*

[5]The camera analogy illustrates the differences between several exploration techniques — it does not, nor was it intended to encompass all possible or effective techniques.

*natural wonder, you quickly realize that your camera cannot capture the entire Canyon in a single photo no matter how wide your wide-angle lens may happen to be [*this is the limitation of the capacity of the human cognitive and visual systems, and the amount of information that can be displayed on a computer console*]. To photograph the entire canyon, you are therefore forced to take a sequence of photos, each aimed at adjacent portions of the canyon [*this is navigation*]. A busybody tourist points out that the Grand Canyon offers widely differing views depending on the viewer's location. You decide to rent a car and drive the length of both the South and the North Rim to obtain the greatest variety of views [*this is change in perspective*]. Having used up several rolls of film, you realize that the Grand Canyon can be appreciated on many scales, not only for its tremendous size, but also for the beauty of the plant and animal life found on its steep walls and deep valleys. You rush to the nearest souvenir shop, purchase another half dozen overpriced rolls of colour film, mount your zoom lens, and make your way down into the canyon, anticipation written all over your perspiring face [*this is a reduction in the level of abstraction*]."*

Although to date there has been little work on graph exploration, the navigation problem in large hypertext and hypermedia networks has been receiving growing attention [35].

One of the most common navigation facilities is the *backtrack*. The *history list* gives a user direct access to previously visited nodes. The *bookmark* is a user-defined place-holder that specifies that a node may be returned to later.

A variety of *overview diagrams* are used to view the information space at various levels of detail. For example, global and local overviews used simultaneously provide the user with two perspectives of the information space. An alternative is to provide *zoom* and *pan* facilities that allow the user to choose the amount of detail to be shown in a particular locale of the information space. A *helicopter* metaphor is used to navigate three-dimensional overview diagrams [19].

The *fisheye view* is a technique that can be used to view the entire space on a single overview diagram in varying levels of detail [23]. Information close to the user's current location is displayed in great detail; the level of detail diminishes with increasing distance from the current location. Figure 6 shows a fisheye view of the original drawing of the sample database graph shown in Figure 1.

Link inheritance is a structured hypertext mechanism that serves both as an abstraction facility and a navigational aid [20]. Link inheritance replaces individual links between nodes in an overview diagram with links that connect clusters of nodes, thus simplifying the diagram and easing navigation.

5 Open Problems in Graph Visualization

In this section, is a partial list of problems that must be solved before a graph visualization facility can be constructed.

5.1 Graph Extraction

Problem 1 *What types of metainformation should be captured during extraction? How can this information be used during the latter stages of graph visualization?*

The main challenge in graph extraction is to determine the type of metainformation that should be extracted, in addition to the graph, and how this information could be used most effectively during the latter stages of graph visualization. For example, it may be possible to derive the properties of the extracted graphs from the properties of the underlying data (for example, minimum or maximum vertex degrees, cyclicity, and so on).

5.2 Graph Storage and Retrieval

Problem 2 *To visualize large graphs, a graph server must be integrated with graph abstraction and graph exploration tools. How should a graph server incorporate domain-specific abstraction information to facilitate graph exploration and abstraction control?*

In most graph storage and retrieval strategies, connectivity is the most important feature of graphs, and closely connected clusters are stored contiguously [2, 29]. While this approach is suitable for simple graph navigation (where the user follows edges from a vertex to one of its neighbours), it makes abstraction more difficult, because vertices in an abstracted graph tend to be distributed throughout the original graph. To address this problem, graph server algorithms will have to incorporate domain-specific abstraction information.

5.3 Graph Abstraction

Problem 3 *What are the primitive graph abstraction concepts that could be used to describe domain-specific abstractions?*

Graph abstraction is by and large a domain-driven activity: very few general or universal abstraction principles appear to exist. It may be possible, however, to define a set of primitive abstraction operations that can be combined to describe more complex domain-specific abstractions.

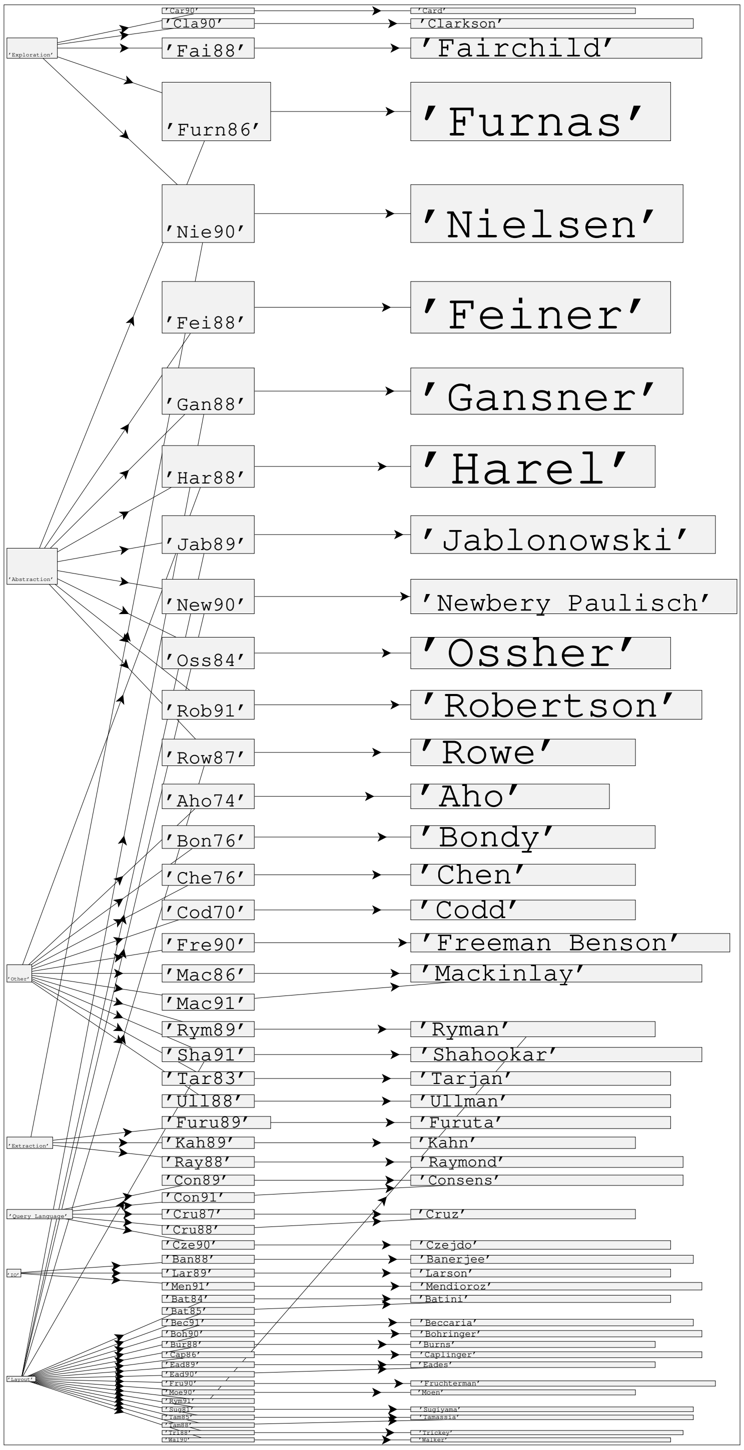

Figure 6: **Graph Exploration.** A fisheye-view of the sample database graph. In a fitting gesture, the fisheye's focus has been centered on node 'Furnas'. Edge labels are suppressed.

Problem 4 *For a set of graph abstraction operations A, can we determine a sequence of graph abstraction operations S so that the result of applying S is the same as that of applying A, and the cost of applying S is minimized?*

Assuming that a graph abstraction operation can be expressed as a set[6] of more primitive abstraction operations, it is interesting to consider the effect of removing, reordering, combining, and replacing of individual operations on the result of the original operation. Specifically, we are interested in optimizing the abstraction operation to reduce the cost of applying it. We would like to determine whether it is possible to reduce the problem of optimizing a graph abstraction operation into a (database) query optimization problem.

5.4 Graph Query Language

Problem 5 *Can a graphical query language be used to pose queries and metaqueries about the graph-theoretic properties of graphs?*

Although most database queries are queries about the data, in many situations it is useful to be able to pose metaqueries about the database schema. Similarly, a graph query language should be capable of solving general queries about the structure of the graphs. This capability could be used to pose queries about the layout of a specific graph or a family of graphs. For example, one may ask: *what are all the vertices that are located to the left of and below vertex X?* Although this type of query is more meaningful within the context of a specific domain-specific layout scenario (for example, a set of cities positioned according to their location on a map), other uses include the study of layout algorithms (for example, what is the effect of certain constraints on layout appearance and stability?). Applications to the other tasks in the graph visualization facility include the study of graph filters, graph abstraction strategies, graph retrieval patterns, and so on.

Problem 6 *Can a graph query language be extended to a full graph-abstraction language? What features must be added and how?*

A graph query language such as Graphlog can be viewed as a subset of a graph abstraction language. Graphlog is used to compose new relationships from existing ones by describing subgraph patterns, treating them as queries, and finding all occurrences of the matching subgraphs in the database graph. The set of matching subgraphs can also be thought of as a filter of the original database while the graph query represents the filtering criteria. Each new relationship defined is really a simple graph abstraction. To extend Graphlog to a more general graph abstraction language, it may be necessary to add some new options (for example, the ability to describe and refer to clusters).

[6] The set may be unordered, partially ordered, or fully ordered.

5.5 Graph Layout

Problem 7 *How can the database schema information be used to derive layout constraints automatically?*

Although there has been some preliminary work to incorporate user-supplied constraints in graph layout, this work suffers several shortcomings. First, in the absence of further mechanisms, constraints must be specified explicitly by the user rather than inferred automatically by the layout algorithm. Second, constraints refer to specific instances of graphs rather than entire graph families and therefore cannot be reused easily. In the context of visualizing relational data as graphs, however, the situation may not be as severe: there may be several possibilities to improve on this aspect of graph layout.

Problem 8 *What are the principles governing the layout of graphs that contain multiple relationships?*

By incorporating arc labels, it is possible to represent multiple relationships among the set of entities (vertices). The subgraph induced by the subset of arcs that defines each relationship can be a tree, a DAG, or a general graph. Each relationship may suggest a specific layout convention [41], as illustrated by the following example.

Example 5.1 *Consider a graph whose vertices represent the entities:* directories, files, *and* functions, *and whose edges represent the relationships: directory* contains *directory or file, file* includes *file, and function* calls *function. The first relationship induces a tree subgraph on directories and files, the second a DAG layout on files, and the third a general graph layout on functions.*

Problem 9 *Can this problem be reduced to a minimization or maximization problem? What is the computational complexity of this problem?*

In most situations, we may be able to provide a partial priority ordering of the constituent relationships (that fixes some relationships as more important than others). The individual subgraph preferences could be a set of weighted constraints, while the ordering could define an equation that measures the degree to which a particular layout satisfies the constraints. The problem could then be treated as a maximization problem.

Problem 10 *Can 2D graph layout be extended to take advantage of three or more dimensions?*

Problem 11 *Can we build graph layout managers that adapt to the available display technology? In essence, can we describe display-independent layout algorithms?*

Visualization of graphs in spaces other than the standard 2D Cartesian plane is a mostly unexplored possibility. There has been some work in $2\frac{1}{2}$D or pseudo-3D graph layout [27, 34], and more recently on full 3D graph layout [38]. The leading-edge technology of virtual reality and cyberspace user interfaces may provide even more powerful visualization facilities, enabling the user to visualize even larger and more complex graphs.

5.6 Graph Exploration

Problem 12 *How can we integrate graph exploration with graph abstraction and graph layout?*

The graphs that are explored in the context of graph visualization are abstracted graphs. An abstracted graph is derived from an original graph by applying a sequence of abstraction operations. Similarly, the graph exploration operation can be treated as a sequence of graph abstraction operations used to vary the amount of detail exposed within various subgraphs of the graph. The *back-end* of the exploration task, then, is simply a set of graph abstraction operations, each applied to some subgraph $S \subseteq G$. The *front-end* of the exploration task is mostly a human-computer interaction (HCI) problem, and is not discussed in this paper.

Problem 13 *How can we simplify the (multiple) graph layout update problem by using our knowledge of the original graph and the types of graph abstraction operations that are encountered in graph exploration?*

When a graph that has been previously laid-out is modified by adding or deleting vertices or edges, we encounter the graph layout update problem: *how does the layout of the graph change to reflect the updates?* Graph exploration operations give rise to multiple graph layout updates. The update problem is difficult in the general case because we cannot anticipate graph updates; however, in the graph exploration scenario, we have additional information in the form of the original graph and the possible abstraction operations that may be applied during exploration.

6 Summary and Conclusions

Visualization tools can ease the analysis of large relational databases. A facility that relies on the graph topovisual formalism to visualize relational data must be capable of performing the following tasks: graph extraction and filtering, graph storage and retrieval, graph abstraction, graph-based queries, graph layout, and graph exploration. We have shown that by integrating this diverse set of tasks, we may benefit from the derived constraints and additional sources of information. Our holistic approach is important for two reasons: it establishes the essential ingredients comprising a GVF, and it identifies certain problems that must be solved before a GVF is realized.

Acknowledgements

I am indebted to several colleagues whose assistance helped to improve the quality of the paper. Mariano Consens, Isabel Cruz, Alberto Mendelzon, Dimitra Vista, and the anonymous referees offered many comments on the paper's content, while the editor made a "few" suggestions regarding its presentation.

References

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

[2] J. Banerjee, W. Kim, S.J. Kim, and F. Garza. Clustering a DAG for CAD Databases. *IEEE Transactions on Software Engineering*, 14(11), November 1988.

[3] C. Batini, M. Talamo, and R. Tamassia. Computer Aided Layout of Entity Relationship Diagrams. *Journal of Systems and Software*, 4(2/3):163–173, July 1984.

[4] K.F. Bohringer and F. Newbery Paulisch. Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms. In *ACM CHI '90*, pages 43–51, Seattle, WA, April 1990. ACM.

[5] J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications*. North Holland, Amsterdam, 1976.

[6] L. Burns, J. Archibald, and A. Malhotra. A Graphical Entity-Relationship Database Browser. In *21st Annual Hawaiian Intl. Conf. on System Sciences*, pages II:694–704, January 1988.

[7] M. Caplinger. Graphical Database Browsing. *ACM SIGOIS Bulletin*, 7(2–3):113–121, Summer-Fall 1986.

[8] S.K. Card, G.G. Roberson, and J.D. Mackinlay. The Information Visualizer: An Information Workspace. In *ACM CHI '91*, pages 181–188. ACM, April 1991.

[9] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[10] M.A. Clarkson. An Easier Interface. *Byte*, pages 277–282, February 1991.

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

[12] M.P. Consens. Graphlog: “Real Life” Recursive Queries using graphs. Master’s thesis, Department of Computer Science, University of Toronto, 1989.

[13] M.P. Consens, A.O. Mendelzon, and A.G. Ryman. Visualizing and Querying Software Structures. In *Proc. of the 1991 CAS Conf.*, pages 17–35, Toronto, Ontario, Canada, October 28–30 1991.

[14] I.F. Cruz. G+: A Graphical system for recursive queries. Master’s thesis, Department of Computer Science, University of Toronto, 1987.

[15] I.F. Cruz, A.O. Mendelzon, and P.T Wood. Recursive Queries Without Recursion. In *2nd Intl. Conf. on Expert Database Systems*, pages 355–368. ACM, 1988.

[16] B. Czejdo, R. Elmasri, M. Rusinkiewicz, and D.W. Embley. A Graphical Data Manipulation Language for an Extended Entity-Relationship Model. *IEEE Computer*, 23(3):26–36, March 1990.

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

[18] P. Eades and R. Tamassia. Algorithms for Drawing Graphs: an Annotated Bibliography. Technical Report CS-89-09, Department of Computer Science, Brown University, 1989.

[19] 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.

[20] S. Feiner. Seeing the Forest for the Trees: Hierarchical Display of Hypertext Structure. In *ACM OIS ’88*, pages 205–212. ACM, March 23–25 1988.

[21] B.N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[22] T. Fruchterman and E. Reingold. Graph Drawing By Force-Directed Placement. Technical Report UIUCDCS-R-90-1609, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1990.

[23] G.W. Furnas. Generalized Fisheye Views. In *ACM CHI ’86*, pages 16–23, Boston, MA, April 13–17 1986. ACM.

[24] R. Furuta, C. Plaisant, and B. Shneiderman. A Spectrum of Automatic Hypertext Constructions. *Hypermedia*, 1(2):179–195, 1989.

[25] 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.

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

[27] 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.

[28] P. Kahn. Linking Together Books: Experiments in Adapting Published Material into Hypertext. *Hypermedia*, 1(2):111–145, 1989.

[29] P. Larson and V. Deshpande. A File Structure Supporting Traversal Recursion. In *ACM MOD ’89*, pages 243–252. ACM, 1989.

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

[31] 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.

[32] C.G. Mendioroz. Graph Storage and Manipulation in Secondary Memory. Master’s thesis, Department of Computer Science, University of Toronto, 1991.

[33] S. Moen. Drawing Dynamic Trees. *IEEE Software*, 7(4):21–28, July 1990.

[34] F. Newbery Paulisch and W.F. Tichy. EDGE: An Extensible Graph Editor. *Software–Practice and Experience*, 20(S1):63–88, June 1990.

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

[36] H.L. Ossher. A New Program Structuring Mechanism Based on Layered Graphs. Technical Report STAN-CS-85-1078, Stanford University, December 1984. Ph.D. Thesis.

[37] D.R. Raymond and F.Wm. Tompa. Hypertext and the Oxford English Dictionary. *Communications of the ACM*, 31(7):871–879, July 1988.

[38] 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.

[39] 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.

[40] A.G. Ryman. The Theory-Model Paradigm in Software Design. Technical Report 74.048, IBM Canada, October 11 1989.

[41] A.G. Ryman. Personal communication, June 1991. Layout of multi-relation graphs.

[42] K. Shahookar and P. Mazumder. VLSI Cell Placement Techniques. *ACM Computing Surveys*, 23(2):143–220, June 1991.

[43] 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.

[44] R. Tamassia. New Layout Techniques for Entity-Relationship Diagrams. In *IEEE Proc. of the 4th Intl. Conf. on Entity-Relationship Approach*, pages 304–311, Chicago, IL, October 28–30 1985. IEEE Computer Society, IEEE Computer Society Press, New York, NY.

[45] R.E. Tarjan. *Data Structures and Network Algorithms*. Number 44 in CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, 1983.

[46] H. Trickey. DRAG: A Graph Drawing System. In *Proc. of the Intl. Conf. on Electronic Publishing, Document Manipulation, and Typography*, pages 171–182, Nice, France, April 20–22 1988. Cambridge University Press, Cambridge, UK.

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

[48] J.Q. II Walker. A Node-positioning Algorithm for General Trees. *Software–Practice and Experience*, 20(7):685–705, July 1990.

Appendix A — The Sample Database

```
%
%  subject(Subject).
%
subject('Abstraction').
subject('Exploration').
subject('Extraction').
subject('IO').
subject('Layout').
subject('Other').
subject('Query Language').

%
%  author(Author).
%
author('Aho').
author('Banerjee').
author('Batini').
author('Beccaria').
author('Bohringer').
author('Bondy').
author('Burns').
author('Caplinger').
author('Card').
author('Chen').
author('Clarkson').
author('Codd').
author('Consens').
author('Cruz').
author('Czejdo').
author('Eades').
author('Fairchild').
author('Feiner').
author('Freeman Benson').
author('Fruchterman').
author('Furnas').
author('Furuta').
author('Gansner').
author('Harel').
author('Jablonowski').
author('Kahn').
author('Larson').
author('Mackinlay').
author('Mendioroz').
author('Moen').
author('Newbery Paulisch').
author('Nielsen').
author('Ossher').
author('Raymond').
author('Robertson').
author('Rowe').
author('Ryman').
author('Shahookar').
author('Sugiyama').
author('Tamassia').
author('Tarjan').
author('Trickey').
author('Ullman').
author('Walker').
```

```
%
%  document(Document).
%
document('Aho74').
document('Ban88').
document('Bat84').
document('Bat85').
document('Bec91').
document('Boh90').
document('Bon76').
document('Bur88').
document('Cap86').
document('Car90').
document('Che76').
document('Cla90').
document('Cod70').
document('Con89').
document('Con91').
document('Cru87').
document('Cru88').
document('Cze90').
document('Ead89').
document('Ead90').
document('Fai88').
document('Fei88').
document('Fre90').
document('Fru90').
document('Furn86').
document('Furu89').
document('Gan88').
document('Har88').
document('Jab89').
document('Kah89').
document('Lar89').
document('Mac86').
document('Mac91').
document('Men91').
document('Moe90').
document('New90').
document('Nie90').
document('Oss84').
document('Ray88').
document('Rob91').
document('Row87').
document('Rym89').
document('Rym91').
document('Sha91').
document('Sug81').
document('Tam85').
document('Tam88').
document('Tar83').
document('Tri88').
document('Ull88').
document('Wal90').

%
%  written_by(Document,Author).
%
written_by('Aho74','Aho').
written_by('Ban88','Banerjee').
written_by('Bat84','Batini').
written_by('Bat85','Batini').
written_by('Bec91','Beccaria').
written_by('Boh90','Bohringer').
written_by('Bon76','Bondy').
written_by('Bur88','Burns').
written_by('Cap86','Caplinger').
written_by('Car90','Card').
written_by('Che76','Chen').
written_by('Cla90','Clarkson').
written_by('Cod70','Codd').
written_by('Con89','Consens').
written_by('Con91','Consens').
written_by('Cru87','Cruz').
written_by('Cru88','Cruz').
written_by('Cze90','Czejdo').
written_by('Ead89','Eades').
written_by('Ead90','Eades').
written_by('Fai88','Fairchild').
written_by('Fei88','Feiner').
written_by('Fre90','Freeman Benson').
written_by('Fru90','Fruchterman').
written_by('Furn86','Furnas').
written_by('Furu89','Furuta').
written_by('Gan88','Gansner').
written_by('Har88','Harel').
written_by('Jab89','Jablonowski').
written_by('Kah89','Kahn').
written_by('Lar89','Larson').
written_by('Mac86','Mackinlay').
written_by('Mac91','Mackinlay').
written_by('Men91','Mendioroz').
written_by('Moe90','Moen').
written_by('New90','Newbery Paulisch').
written_by('Nie90','Nielsen').
written_by('Oss84','Ossher').
written_by('Ray88','Raymond').
written_by('Rob91','Robertson').
written_by('Row87','Rowe').
written_by('Rym89','Ryman').
written_by('Rym91','Ryman').
written_by('Sha91','Shahookar').
written_by('Sug81','Sugiyama').
written_by('Tam85','Tamassia').
written_by('Tam88','Tamassia').
written_by('Tar83','Tarjan').
written_by('Tri88','Trickey').
written_by('Ull88','Ullman').
written_by('Wal90','Walker').

%
%  discussed_in(Subject,Document).
%
discussed_in('Abstraction','Fei88').
discussed_in('Abstraction','Furn86').
discussed_in('Abstraction','Gan88').
discussed_in('Abstraction','Har88').
discussed_in('Abstraction','Jab89').
discussed_in('Abstraction','New90').
discussed_in('Abstraction','Oss84').
discussed_in('Abstraction','Rob91').
discussed_in('Abstraction','Row87').
discussed_in('Exploration','Car90').
discussed_in('Exploration','Cla90').
discussed_in('Exploration','Fai88').
discussed_in('Exploration','Furn86').
discussed_in('Exploration','Nie90').
discussed_in('Extraction','Furu89').
discussed_in('Extraction','Kah89').
discussed_in('Extraction','Nie90').
discussed_in('Extraction','Ray88').
discussed_in('IO','Ban88').
discussed_in('IO','Lar89').
```

```
discussed_in('IO','Men91').
discussed_in('Layout','Bat84').
discussed_in('Layout','Bat85').
discussed_in('Layout','Bec91').
discussed_in('Layout','Boh90').
discussed_in('Layout','Bur88').
discussed_in('Layout','Cap86').
discussed_in('Layout','Ead89').
discussed_in('Layout','Ead90').
discussed_in('Layout','Fru90').
discussed_in('Layout','Gan88').
discussed_in('Layout','Jab89').
discussed_in('Layout','Moe90').
discussed_in('Layout','New90').
discussed_in('Layout','Row87').
discussed_in('Layout','Rym91').
discussed_in('Layout','Sha91').
discussed_in('Layout','Sug81').
discussed_in('Layout','Tam85').
discussed_in('Layout','Tam88').
discussed_in('Layout','Tri88').
discussed_in('Layout','Wal90').
discussed_in('Other','Aho74').
discussed_in('Other','Bon76').
discussed_in('Other','Che76').
discussed_in('Other','Cod70').
discussed_in('Other','Fre90').
discussed_in('Other','Har88').
discussed_in('Other','Mac86').
discussed_in('Other','Mac91').
discussed_in('Other','Rym89').
discussed_in('Other','Sha91').
discussed_in('Other','Tar83').
discussed_in('Other','Ull88').
discussed_in('Query Language','Con89').
discussed_in('Query Language','Con91').
discussed_in('Query Language','Cru87').
discussed_in('Query Language','Cru88').
discussed_in('Query Language','Cze90').
```