Interactive Visualization of Large Graphs

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
maandag 21 november 2005 om 16.00 uur

door

Franciscus Jacobus Johannes van Ham

geboren te Veldhoven
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. J.J. van Wijk
en
prof.dr.ir. J.F. Groote

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Ham, Frank van

Interactive visualization of large graphs / door Frank van Ham. -
Proefschrift. - ISBN 90-386-0704-0
NUR 980
Subject headings: computer visualisation / information presentation / image communication
CR Subject Classification (1998) : H.5.2, I.3.8
The work in this thesis was supported by NWO under grant number 612.000.101.

The work in this thesis has been carried out in the research school ASCI (Advanced School for Computing and Imaging).

© F.J.J. van Ham 2005. All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

Printing: Eindhoven University Press

Cover design: JWL producties, Eindhoven
# Contents

1 Introduction ................................................................. 1
   1.1 Motivation .................................................................. 2
   1.2 Objective .................................................................. 4
   1.3 Outline .................................................................. 4

2 Problem ............................................................................. 7
   2.1 Graphs ..................................................................... 7
      2.1.1 Simple graphs .................................................. 8
      2.1.2 Planar Graphs .................................................. 9
      2.1.3 Cyclic and acyclic graphs .................................... 9
      2.1.4 Unconnected and connected graphs ...................... 10
      2.1.5 Trees ................................................................ 10
      2.1.6 Weighted Graphs .............................................. 10
      2.1.7 Dense and sparse graphs .................................... 11
      2.1.8 Degree distributions .......................................... 11
      2.1.9 Graph size ....................................................... 12
      2.1.10 Graph attributes ............................................. 12
   2.2 Aim .......................................................................... 12
      2.2.1 Fuzzyness ......................................................... 13
      2.2.2 Scope ................................................................ 13
      2.2.3 Orientation ...................................................... 14
      2.2.4 Usage pattern ................................................... 14
   2.3 Context ...................................................................... 14
      2.3.1 Pre-existing mental maps .................................... 15
      2.3.2 User expertise .................................................. 15
      2.3.3 Hardware ......................................................... 15

3 Previous Work ..................................................................... 17
   3.1 Graph Drawing .......................................................... 17
      3.1.1 Aesthetics and conventions .................................. 18
      3.1.2 Rooted Trees ..................................................... 19
      3.1.3 Directed Acyclic Graphs ..................................... 19
7 Matrixzoom
7.1 Introduction ........................................... 91
7.2 Approach .............................................. 92
7.3 Matrix visualizations ..................................... 93
7.4 Visualization ............................................ 94
  7.4.1 Matrix subdivision ................................... 95
  7.4.2 Multilevel zooming .................................. 96
  7.4.3 Multidimensional Information ......................... 98
7.5 Results .................................................. 98
7.6 Memory issues .......................................... 101
  7.6.1 Architecture ......................................... 102
  7.6.2 Visualization modifications ........................... 104
  7.6.3 Application .......................................... 105
7.7 Conclusions ............................................. 109

8 Small World Graphs
8.1 Introduction ............................................ 111
8.2 Approach .............................................. 112
8.3 Force model ............................................. 113
8.4 Visualization ............................................ 116
  8.4.1 Visual Abstraction ................................... 116
  8.4.2 Clustering comparison ................................. 118
  8.4.3 Detail and Context ................................... 121
  8.4.4 Edges ................................................. 126
8.5 Conclusions ............................................. 128

9 Reflection
9.1 Comparison .............................................. 129
  9.1.1 Flexibility ........................................... 129
  9.1.2 Scalability .......................................... 132
9.2 Common design aspects ..................................... 133
  9.2.1 Representation ........................................ 135
  9.2.2 Abstraction .......................................... 136
  9.2.3 Navigation .......................................... 137
  9.2.4 Mapping ............................................. 138
9.3 Developing new visualizations ................................ 139
  9.3.1 Data ................................................. 139
  9.3.2 Aim .................................................. 140
  9.3.3 Context .............................................. 140

10 General conclusions ........................................ 143
10.1 On graph visualization .................................... 143
10.2 On evaluation ........................................... 144
10.3 Future work ............................................. 146
Bibliography

A Color Plates

B Statistics for beamtree user experiment
  B.1 Estimating the size of a node .................................. 177
  B.2 Estimating the depth of a node .................................. 178
  B.3 Estimating the number of levels in the tree .................. 178
  B.4 Finding the common ancestor of two nodes in the tree ...... 179
  B.5 Recalling the position of a previously indicated node ...... 180

Samenvatting

Nawoord
Chapter 1

Introduction

In recent decades, computer systems have allowed us to gather, store and manipulate ever increasing amounts of information with relative ease. Thanks to these advances in computer hard- and software, the internet now consists of several billion web pages (and is still growing), word-processor software now counts over a million lines of code and researchers have unravelled the basic structure of the 2.9 billion base pairs in the human genome.

However, as a direct result, the amount of information available to a single individual is getting both bigger and more complex, up to the point where it is nearly impossible for the average person to decide which information is relevant for him, in the short time that is typically available. As an example, currently a weekday edition of The New York Times contains more information than the average person was likely to come across in a lifetime in seventeenth-century England [183]. The ability to filter relevant information from this information flood quickly and accurately is an important asset for businesses and individuals in this increasingly information oriented world.

This 'information overload' problem is compounded by the fact that, in many cases, the user does not know exactly what specific information he is looking for. He might be interested in unknown trends, anomalies or clusters, but precisely because they are unknown, it is hard to specify the exact criteria a computer needs to retrieve them. And, due to the sheer size of the data, it is impractical for the user to sift through all of it manually. There is therefore a need for computer tools that allow users to manage information more effectively.

With the increase of the speed of graphics hardware over the last decade, millions of graphical primitives can now be rendered on screen at real time speeds. This has been an enabling technology for the research field of Visualization, which can be defined as 'The use of computer-supported, interactive, visual representations of data to amplify cognition' [37]. In other words, visualization concerns itself with the use of visual methods to gain insight into complex data sets. Visualization is a useful method to convey large amounts of information because:
Chapter 1 Introduction

It allows for fast pattern recognition since the visual system is the main source of input for a human, our brain has evolved to recognize certain properties immediately, without requiring conscious thought. These include amongst others color, shape and size (See figure 1.1).

- it has been a natural way to communicate both abstract and concrete ideas throughout history.
- it allows for fast pattern recognition. Since the visual system is the main source of input for a human, our brain has evolved to recognize certain properties immediately, without requiring conscious thought. These include amongst others color, shape and size (See figure 1.1).
- it offers a high bandwidth access directly into the brain. Verbally describing all but the most trivial pictures would indeed take more than a thousand words.
- it allows users to externalize part of their short term working memory. Early studies [122] have shown that humans can keep at most between 7 and 10 items in working memory at one time.

Research in visualization is traditionally split into two separate areas. One of these is Scientific Visualization, which concerns itself with the visualization of (often continuous) data that has an intrinsic representation in the real world. One can think of turbulent flow fields, medical imaging or stress visualizations in mechanical design. The other one is called Information Visualization (although technically speaking, Abstract Data Visualization might be a better name), which deals with the visualization of abstract discrete data for which no immediate spatial representation exists, such as large data tables, electronic document collections or software programs.

1.1 Motivation

In this thesis we focus on one specific type of abstract data, namely data that comes in the form of graphs. Such data is ever present in our society, from functional relationships between genes in biology to social networks in sociology and from component relations...
1.1 Motivation

The common denominator in these types of datasets is that they all consist of a number of distinct items (also called nodes), such as genes, persons or software components, and relations (or edges) between those nodes (i.e., gene X activates gene Y, person X knows person Y or software component X uses software component Y).

Since it is difficult for most humans to reason about a graph without a visual representation of that graph, graphs are often represented by node link diagrams, which represent the nodes as icons and the relations between those nodes as lines connecting two icons (figure 1.2a). One should not confuse this representation of a graph with the graph itself however, because an infinite number of possible drawings exists for a single graph. Dijkstra therefore reasoned that pictures of mathematical concepts such as graphs are undesirable in general because they are overspecific [46, 45, 47]. Indeed, we cannot draw fundamental conclusions on general graphs from a specific visual representation of a specific graph, yet understanding a specific graph is difficult without a visual representation. Insights gained on specific graphs can then be used to create new hypotheses on the entire class of graphs from a specific application area.

As drawing these node link diagrams by hand can get quite tedious, many methods and techniques for automated graph drawing have been developed and implemented in tools such as Neato and Dot [68]. Although these automatically generated diagrams look familiar and are easy to read, they fall short when the numbers of nodes is larger than several hundreds (figure 1.2b). The large number of overlapping edges makes it hard to make statements on connectivity. Another problem is that the generated diagrams are often static, or allow only limited interaction, and thus do not exploit the possibilities offered by current generation hardware.
1.2 Objective

The research question that we address in this thesis is therefore:

*How can we use computer supported visualization to help users gain insight in the structure and content of large graphs, with thousands or millions of nodes?*

Note that we do not confine ourselves to a specific application area. This means that we are dealing with a large problem space. There are also surprisingly few methods, techniques or tools that are able to visualize large graphs, so there are almost no best-practices known. Both of the above factors make the problem that we are facing hard. Since a single solution that provides an optimal visualization for any problem is not likely to exist, the nature of this thesis is exploratory.

In exploring the field, we do not attempt to answer the above research question analytically, but instead sample the space of possible solutions. We first determine a number of aspects that influence the resulting visualization. We then select a number of instances for the valuation of these aspects and construct and evaluate (formally and/or informally) a suitable visualization. Afterwards we compare the results and extract a set of fundamental problems that arise in any graph visualization. Finally, we come to a set of recommendations that one can adhere to when faced with the problem of constructing a suitable visualization of a specific large graph.

1.3 Outline

In the next chapter, we look at the problem in more detail and define a number of input variables that influence our final visualization. We subdivide these into data-related, task-related, and contextual factors.

In Chapter 3, we discuss related work. We present the most well-known graph layout and clustering algorithms from the area of graph drawing and novel representations and interaction techniques from the area of information visualization. In Chapter 4 we choose four different valuations for the factors we described in Chapter 2 resulting in four different graph visualization problems. We then construct visualizations for these problems in chapters 5 to 8.

Chapter 5 presents a graph visualization that is specifically tuned to quickly finding the largest elements in a general tree. It is based on the concept of treemaps, a space filling visualization technique. We improved on treemaps by using overlap as a visual cue to show the hierarchical structure of the tree. We performed a formal user test to quantify the improvement. This work was previously published [79, 80, 179].

In Chapter 6, we present a novel method to visualize state transition graphs. A state transition graph is an elementary representation of a computer process, in which nodes represent memory states and edges represent transitions from one state to another. The size of state transition graphs can range from a few dozen nodes to billions of nodes. To be able to deal with these sizes we developed a fast technique that clusters individual
1.3 Outline

states together based on path branching. We designed the visualization to explicitly show symmetries that are often present in these types of graphs. Parts of this chapter were previously published [77, 78, 71].

In Chapter 7, we tackle a real world software engineering problem. In many cases the software architect designs a system, after which an implementation is created by a larger number of programmers. The architect has no way of knowing to what extent this implementation corresponds to his original design. We show code deviations of the original design by using a hierarchical matrix representation of a graph, which allows the user to zoom from architectural level to code level. The visualization is also able to visualize graphs whose edge set does not fit in RAM. This chapter was previously published in [76]. Section 7.6 was previously published [3], with both authors contributing equally.

Chapter 8 deals with graphs in which every node can be reached from any other node in a small number of steps, also known as small world graphs. We use a layout algorithm to create an embedding that represents the cluster structure present in the graph. The visualization uses a novel browsing scheme that allows the user to see detailed structure in an area he or she is interested in, with the higher level contextual structure displayed in the periphery. This work has been previously published in [81].

In Chapter 9 we evaluate and compare these techniques and based on the findings we extract four core problems that will surface in any large graph visualization. We end this chapter with a number of practical guidelines that one can adhere to when faced with the problem of visualizing an unknown graph. Finally, in Chapter 10 we present conclusions and give suggestions for future work.
Chapter 2

Problem

Faced with the problem of designing a good visualization, we first define what the external factors that influence this design are. Based on the definition of the term visualization in the previous chapter we can state that a visualization is “A computer-supported, interactive visual representation of data used to amplify cognition”. Besides the representation that we are trying to construct, the two important words in this definition are data and cognition.

Obviously, the actual data that we are trying to visualize has a great influence on our final design. Although we confined ourselves to graphs in general, there are many specific types of graphs when we take a closer look at their structure. Also, the amount of additional information given with a graph differs between application areas.

The cognition in the definition refers to the cognitive task that the user is trying to accomplish, or the aim of the user. This task can be very specific, in the sense that the user is trying to answer a well-defined question about the data. Alternatively, tasks might be very general in the sense that the user does not know enough about the data to formulate a more well-defined question in the first place or because he is not able to formally specify the question.

A third factor that is of influence when designing a visualization is the context. Contextual factors can be subdivided into human factors, such as user expertise or existing mental maps of the data, and machine factors, such as amount of available RAM or screen space and hardware architecture.

We shall elaborate on each of these three sets of properties in the next sections.

2.1 Graphs

As we stated in the previous chapter, the data that we want to visualize are (finite) graphs. A finite graph consists of a finite set of nodes (also known as vertices) \( V \) and a finite set of edges \( E \). An edge \( e \in E \) connecting the node pair \( x \) and \( y \) will be denoted with \( e_{xy} \). Graphs can be classified as undirected or directed based on their interpretation of direction.
Figure 2.1 Factors influencing visualization design: data, aim and context

Undirected graphs do not take the ordering in the node pair into account, that is, $E \subseteq 2^V$, where $2^V$ is the set of all 2-subsets of $V$. Since each edge is modelled as an unordered pair, the notation $e_{xy}$ is equivalent with $e_{yx}$. Nodes $x$ and $y$ are endpoints of $e_{xy}$ and $e_{yx}$ is both incident to $x$ and to $y$. We can also say that $x$ is a neighbor of $y$ and vice versa. The total number edges incident to a single node $x$ is called the degree of $x$. If no further qualification is given, the term graph is generally assumed to mean an undirected graph.

In the case of directed graphs (or digraphs), $E$ is a subset of $V \times V$, and a directed edge (also known as an arc) is an ordered pair of nodes. Directed graphs interpret a directed edge $e_{xy}$ as a connection from the tail $x$ to the head $y$. That is, $y$ can be reached from $x$ but $x$ cannot be reached from $y$. The set of edges that have $x$ as tail are the out-edges of $x$. Similarly, the set of edges that have $x$ as head constitutes the in-edges of $x$. The cardinality of these sets denote the indegree and outdegree of $x$, respectively.

Visualization of directed graphs is more complex in general because of their asymmetry. Besides the fact that we need to visualize the fact that two nodes are related, we also need to visualize the direction of this relation. In node link diagrams this is commonly solved by adding an arrowhead indicating the direction, but this adds extra visual complexity to the visualization, especially if we are dealing with thousands of edges.

2.1.1 Simple graphs

An edge $e_{xx}$ is called a self-edge or loop. An edge $e_{xy}$ is called multiple if there exists another edge $e'_{xy}$ with the same endpoints. The number of edges in the graph with endpoints
2.1 Graphs

Figure 2.2 (a) A tree with a path of length 3 in bold; (b) A planar directed acyclic graph

$x$ and $y$ is called the *multiplicity* of edge $e_{xy}$ or $mult(e_{xy})$. A graph is called *simple* if it contains no loops or multiple edges.

For visualization purposes we often consider only simple graphs. Any graph can be made simple by removing loops and aggregating multiple edges into a single edge with integer multiplicity. If desired we can easily add loops and multiple edges back into the image after the visualization is constructed.

### 2.1.2 Planar Graphs

A graph is planar if we can construct a two dimensional drawing of the graph such that no edges cross. Planarity testing for a graph can be done in linear time [90]. A drawing of a graph can be made planar by introducing dummy nodes at the edge crossings.

Planar graphs can offer very pleasing visualizations, but unfortunately in real world data sets they are rarely seen.

### 2.1.3 Cyclic and acyclic graphs

A walk of length $N$ is defined as a sequence of edges $(e_{x_0x_1}, e_{x_1x_2}, e_{x_2x_3}, \ldots, e_{x_{N-1}x_N})$. A walk is called a *path* if all vertices $x_i$ ($0 \leq i \leq N$) are distinct. A walk is called a *trail* if all edges in the walk are distinct. A *cycle* is a non-empty trail in which the start and endpoint are the same, i.e., $x_1 = x_N$. Paths and cycles are defined for both directed and undirected graphs and a path on a digraph is also called a *dipath*. The graph theoretic distance between two nodes $i$ and $j$ is equal to the length of the shortest path connecting $i$ and $j$.

A connected graph is a graph in which there exists a path from any node to any other node. A digraph in which there exists a dipath from any node to any other node is called *strongly connected*. A digraph whose underlying undirected graph is connected is called *weakly connected*. All other graphs are called *unconnected*.

Weakly connected finite acyclic directed graphs, are also referred to as *Directed Acyclic Graphs*, or DAGs. In a DAG we can always identify at least one node with no in-edges.
(a source) and at least one node with no out-edges (a sink). This makes the layout of that graph easier because we can assign a depth (or layer) to each node which is equal to the distance from that node to a source. Mapping that depth to a display dimension gives a good impression of the node ordering of the graph. For example, if the depth of \( y \) is larger than the depth of \( x \), we can state that no dipaths in which node \( y \) occurs before node \( x \) exist.

### 2.1.4 Unconnected and connected graphs

Some graph algorithms do not work well on unconnected graphs. Apart from that, visualizing an unconnected graph as a whole is not very useful because there are no relations between individual components. The relative positioning of individual components in the visualization has therefore no immediate meaning. Instead of wasting computational resources on creating a layout of an unconnected graph as a whole, it is often better to first detect the connected components in a graph. Such a test can be implemented in \( O(|V| + |E|) \) time by a depth first search algorithm or in \( O(E) \) by using a variant of Tarjan Union Find [162]. In the case of multiple connected components we can visualize each component separately, thereby reducing the complexity of the visualization.

### 2.1.5 Trees

Trees are connected acyclic simple graphs. Rooted trees are trees that have an unique root(node). Rooted trees (also known as hierarchies) can also be modelled by a connected acyclic simple digraph, in which all arcs point away from the root and no node has more than one in-edge. The nodes in the tree that have degree 1 (excluding the root) are called leaves. Nodes that are not leaves are called internal nodes. The depth of a node \( x \) in a rooted tree is given by the length of the path between the root and \( x \). The neighbor of \( x \) with lower depth than \( x \) is called the parent of \( x \). The neighbors of \( x \) with higher depth are the children of \( x \). Every node in a rooted tree has a parent except for the root. The transitive closure of both these relations are commonly denoted by ancestor and descendant respectively.

Because trees contain no cycles they are generally easier to visualize than general graphs. Common visualizations use a top down (or left-right) layout ordered by depth (figure 3.1.2). Layout algorithms can compute such a layout in \( O(N) \) time. However, for trees we can also construct visualizations that use containment [98] or overlap [80] as a means to visually establish a relationship between a parent and its child nodes.

### 2.1.6 Weighted Graphs

In a weighted graph each edge is assigned a (usually non-negative, non-zero) weight. This weight can for example indicate the relative strength of a relation between two nodes. Weighted graphs can also be thought of as continuous versions of a discrete graph. Instead of a connection either being there or not, a connection can exist with relative weight \( w \). We indicate the weight of an edge \( e_{ij} \) by \( w_{ij} \) and \( w_{ij} = 0 \) if there is no edge between \( i \) and
Any arbitrary graph can be made weighted by supplying \( w_{ij} = 1 \) for each edge. The additional information present in weighted graphs can be used to improve the performance of clustering and layout algorithms.

### 2.1.7 Dense and sparse graphs

Informally we can define dense graphs as graphs with a relatively large number of edges and sparse graphs as graphs with a relatively small number of edges. However, these notions of large and small are fairly arbitrary. To be more precise, in this thesis we call a class of graphs with \( N \) nodes and more than \( N \log(N) \) edges dense. Similarly, graphs in which the number of edges is below \( N \log(N) \) are called sparse.

The total density of a graph can be expressed by \( \frac{|E|}{N(N-1)} \). The diameter of a graph is defined by the maximal shortest path length over all node pairs in the graph.

Because of their higher density, dense graphs usually have a smaller diameter. This makes traditional node link visualization very compact and hard to read. For dense graphs we have to come up with different solutions.

### 2.1.8 Degree distributions

With the degree distribution we mean the distribution of node degrees over the node set. Degree distributions can be calculated quickly from the initial graph and can give us valuable information on the underlying basic structure of the graph. For random graphs we can also predict if the graph will consist of one giant component or multiple smaller disconnected components by looking at the degree distribution [10].

Graphs in which two nodes are connected completely at random are known as random graphs [54] or Erdős-Renyi graphs. Random graphs have a Poisson degree distribution. If the probability that an edge occurs between two arbitrary nodes is \( p \), then the probability \( p_k \) that a node has degree exactly \( k \) will be:

\[
p_k = \binom{N}{k} p^k (1-p)^{N-k}
\]

which limits to the Poisson distribution for large \( N \). Large random graphs are rare in information visualization because they contain no structure by definition and visualizing a random graph to be able to make statements on its structure is not very useful.

Recent research has found that in many real world graphs, such as the Internet [105] or telecommunications call graphs [2] the node degrees exhibit a power law degree distribution in which the number of nodes with degree \( k \) is proportional to \( k^{-\beta} \) with \( \beta > 0 \) (usually 2 or 3). Power law distributed graphs therefore exhibit a small number of nodes with high degree (hubs) and a large number of nodes with low degree. This information can be used in the design of a visualization.

Grid-like graphs have a spiked degree distribution, that is, the variation in node degrees is small. These graphs usually also have a mesh like structure, examples include finite element graphs and meshes from 3D object scans.
2.1.9 Graph size

Another important aspect of the input data is the actual number of nodes and edges of the graph. We consider size in terms of the number of nodes, with the number of edges determined by the graph’s density. We can roughly divide graphs into three size classes based on the amount of RAM they take up [2]. The first class are graphs whose vertex and edge sets fit in the system RAM. The second one are so called semi-external graphs, whose node set fits in RAM but whose edge set does not. Finally, fully external graphs have a node set that is too big to fit in RAM. Note that with the ever increasing size of RAM even the first class can contain graphs that are large. Assuming memory footprints of 100 bytes per node and 20 bytes per edge and a (high) graph density of 0.5, we can still store a graph with 10,000 nodes in 1 GB of RAM.

The size of the graph also poses limits on the visualization techniques that we can use. Techniques that are developed for graphs up to 100 nodes will almost certainly fail when confronted with a graph of 10,000 nodes or more. Sometimes this breakdown occurs because of technical or algorithmic reasons (i.e. the visualization cannot interactively display 10,000 items at once or uses an $O(N^2)$ layout algorithm), but more often because of human visual overload or lack of screen space. Possible solutions to the size problem are for example grouping nodes together into clusters or showing only a small part of a large graph at once. However, both of these solutions make it harder for the user to maintain context when navigating.

2.1.10 Graph attributes

With graph attributes we mean all extra data that is stored with nodes and edges. Mathematically speaking graphs do not need to have any attributes associated with their nodes and edges, but because we are visualizing real world data sets, they often do. We split the set of attributes in node attributes, such as the size or class of the item the node represents, and edge attributes such as the person responsible for making that connection or a weight.

Having attributes available greatly enhances the usage potential of a visualization, making it easier to extract new insights. We can try and map different properties on the structure of the graph to see how well they correspond. For example, we can try to find two separate components that each have a different value for a specific property. Attributes also allow us to better evaluate the appropriateness of a specific visualization, because we can use domain knowledge to judge whether new information on the graph structure is correct.

2.2 Aim

Another variable that has a significant impact on the resulting visualization is the aim of the user. What is he or she trying to accomplish by using the visualization? In this section we define three independent dimensions that govern the type of questions asked. Table 2.1 gives a sample question for each category.
2.2 Aim

2.2.1 Fuzzyness

Potential questions a user might have about a graph data set range from the very concrete to the very fuzzy. An example of a very specific question might be “How many sink nodes does this digraph have?”. Although we can easily answer these types of questions with visualization, for example by highlighting all sink nodes in a graph, a dedicated algorithm is able to provide an exact numerical answer much faster, especially when the number of sink nodes is very large.

Visualization is really helpful, in answering fuzzy, hard to specify questions such as “Is there any cluster structure present in the graph”. It is often the case that the user does not have enough understanding of the structure of the graph he is interested in to formulate a concrete question that improves his insight. Since we often cannot give a concrete answer to a fuzzy question, the best we can do is return a less fuzzy answer, which in turn can be used to formulate a new question.

A visualization can serve as a useful tool to provide answers to fuzzy questions that often pop up in the first phases of analysis of an unknown object. At the same time we can also use the visualization to show the answers to concrete questions, helping the user to better understand their relationships or find previously unknown relationships.

2.2.2 Scope

Overview questions are related to global structure and often involve comparing all items in the graph with respect to each other. Examples of global questions are for example “What are the leaf nodes with the largest sizes?” or “What is the global structure of my graph?”. Detail questions are questions about a specific item in the data set, such as “What are the direct neighbors of node X?” or “What does node X represent?”. In general, overview questions tend to have fuzzy question characteristics, while detail questions are more specific.

The scope also determines the general type of use of a visualization. Static visualizations that only focus on overview questions are usually informative, meaning that their main purpose is to relay information to the user. Function plots are a prime example of informative visualizations: they give the user a general overview of a function’s characteristics, but offer no possibility to read exact function values from the graph. Visualizations that facilitate both overview and detail questions are used in an explorative way. Typically, the user will query the overview to obtain detailed information on different items that are visible in the visualization. Explorative visualizations are generally more interactive.

One of the main advantages of visualization in general is that it can answer both detail and overview questions and also show the relations between them. This combination makes it easier for the user to mentally ‘absorb’ the data and may lead to new hypotheses on the structure of the data set.
2.2.3 Orientation

Questions from a user can be either structure oriented, data oriented or a combination of both. Structure oriented questions pertain only to the structural properties of the graph, such as node degree or cluster structure. Data oriented questions are questions relating to the attribute data attached to nodes and edges. Note that data oriented questions might also be answered by querying an appropriate database. The real power of graph visualization however, lies in its ability to answer questions that are both structure and data related. Examples of this type of questions include “Are there any clusters of nodes that share a common property?” or “What are the relations between nodes with property A and nodes with property B?”.

2.2.4 Usage pattern

Another important factor to consider is the usage pattern of the visualization. Will the visualization primarily be used to answer one specific type of question or should it be kept general enough to answer any question the user can possibly come up with during use? In other words, will the general use fall in one specific category of questions (i.e., fuzzy, global, data oriented questions) or is the usage pattern a priori unknown?

Both approaches have their advantages: customized solutions exploit the potential of visualization techniques to the maximum but often lack the flexibility to effectively deal with other types of questions. On the other hand, very general solutions are potentially able to answer different questions but suffer in performance. A good example of a customized visualization method are treemaps [98], which are very good at showing a numerical node property but perform less admirably for displaying node structure.

2.3 Context

With context we mean all side constraints that influence our visualization design in some way. Generally, these fall into two categories, namely user and hardware.
2.3 Context

2.3.1 Pre-existing mental maps

Mental maps are the brain’s equivalent of physical maps, and make up a user’s internal model of an abstract data set. A mental map is created by browsing an externalization (in this case a visualization) of the data. Over time, these browsing activities lead to an established mental map of that data. These mental maps are surprisingly resistant to change [153], because that would require the user to unlearn what he has learned during previous sessions. Imagine what would happen if all map makers suddenly decided to map the magnetic east west direction to the up down axis on all maps. Although all maps would still be valid most people would have a very hard time getting used to the new ones.

For this reason it is important that we take care not to destroy any existing mental maps that the user has acquired, but instead create a visualization that leaves existing maps intact. This will generally make it easier for users to accept a new visualization. For example, a visualization of the relations in a software system should preferably have a correspondence with the image of the (smaller) basic architectural structure of that system.

Generally accepted visual notations also fall under mental maps in some sense. In some cases a specific visual cue has been in use for so long that it is automatically associated with a particular object or property. For example, in UML diagrams the generally accepted notation for nodes is a rectangle, in medical visualizations the color red is not very suitable for marking anything since it is commonly associated with tissue oxygenation, and artificial lighting usually comes from above. Although these issues are usually corrected easily, they can play an important role in the acceptance of a visualization.

2.3.2 User expertise

With user expertise we mean both the knowledge a user has on the data that we are visualizing and the experience a user has in working with visualizations in general. If the proposed user(group) has very little general knowledge on the subject area of the visualization, an informative visualization with minimal interaction possibilities might suffice. Expert users generally want the ability to explore and filter the data, perform what-if analyses and use different visualizations to explore different aspects of the data.

Of course, it is also possible to create a single visualization that offers both modes of operation. Novice users can use a restricted set of functions to gain familiarity with the data set and then explore it using more complex functionality.

2.3.3 Hardware

One of the most obvious side constraints we have to work with is the hardware that we are using to create the visualization. If we disregard the main processor speed, which has been steadily increasing over the years, we are dealing with three possible limits in practice. The first is the screen size. Normally, visualizations operate on a standard desktop configuration meaning that we have about 1,000,000 pixels to work with. It is however possible that we have a projection wall available that consists of a number of
Chapter 2 Problem

these desktops with linked displays [112], offering us much more freedom in what we can show. On the other hand, the visualization might have to run on a small hand-held device with a much smaller two-color low resolution screen, severely limiting our display options.

Another limiting hardware factor is the amount of physical RAM available for storage and computation. We briefly touched upon this problem in section 2.1.9. Clearly we have to use a different visualization design when our graph does not fit in RAM as a whole. In these cases incremental browsing techniques (see section 3.3.2) might be more appropriate. Besides graph storage, we have to take the memory complexity of computations into account. Algorithms that use $O(N^2)$ memory are unsuitable when visualizing larger graphs. For example, the memory requirements for a $O(N^2)$ memory algorithm on a graph of 5,000 nodes, assuming a node footprint of 128 bytes are already 3 GB.

Finally, the last potential bottleneck is the speed with which we can draw objects in memory to a computer screen. The higher this speed, the more items we can interactively show at the same time. Hardware accelerated rendering on consumer PC’s has been possible since the mid 1990’s and has seen a staggering increase in performance over the last decade, enabling us to render more than half a billion shaded polygons per second on current consumer level hardware. We therefore do not expect rendering speed to be much of a bottleneck in the future, given the still increasing rendering speeds and decreasing prices.

In this chapter we gave an overview of the different variables that play an important role in designing a visualization. Generally speaking, the characteristics of the input graphs are different for each specific application domain, and aims may also vary over an application domain. However suitable layouts exist for specific types of graphs, such as trees and DAGs, independent of the application domain. A number of techniques has also been developed to deal with general problems such as scale and scope. In the next chapter we give of an overview of existing graph layout, clustering and visualization techniques.
Chapter 3

Previous Work

Previous work in graph visualization is substantial. Visualization itself is a multidisciplinary area and relevant research is spread out over different, sometimes overlapping disciplines (such as user interfaces, perception and visualization). Another major source of knowledge are visualizations that come from application areas themselves and which are often published in other venues.

Graph theory has been around ever since Euler published his analysis of the Königsberger bridges problem [55] in 1736. It goes far beyond the scope of this thesis to offer an overview of all graph related research since then, so we confine ourselves to the areas of graph drawing and graph clustering, which are most relevant for graph visualization. Graph drawing is an area that attempts to construct optimal drawings for graphs. The aim of graph clustering is to find optimal clusterings of nodes. In the next sections we give an overview of previous work in graph drawing, graph clustering and visualization research.

3.1 Graph Drawing

The origination of graph theory is widely accredited to Euler, yet strangely enough, he did not make use of graphical representations of graphs\(^1\). In fact the first schematic drawing of the Königsberger bridges problem did not appear until 1892 [16].

That is not to say that graph drawings did not exist before this time; samples have been found that date as far back as the 11th century. Graph drawings in the Middle Ages were mainly used in genealogy, categorization, and logic. In the early 19th century, drawings of graphs were making their way from mathematics into other application areas, such as molecular notation in chemistry. In [108] a comprehensive illustrated overview of early (manual) graph drawing is given.

Graph drawing as a discipline flourished with the realization that computers could be used to automatically generate drawings of graphs, as opposed to the tedious process of

\(^1\)as a side note, Euler wrote almost half of his work when he was blind, so he probably did not need visualizations.
drawing a node-link diagram by hand. It is mainly concerned with creating node link layouts of graphs up to a few thousand nodes. We give an overview of some of the most important ideas and algorithms. A more in-depth treatment can be found in [21].

3.1.1 Aesthetics and conventions

Since graph drawing deals with laying out a graph in an effective manner, we first have to define what we mean with ‘effective’. Unfortunately there is no single requirement, and over the years a large number of them have surfaced. The most important ones are:

- Minimize number of edge crossings: edge crossings make edges harder to track visually;

- Minimize number of edge bends: edges with a large number of bends are harder to track visually;

- Maximize path continuity: discontinuous paths are harder to track visually;

- Maximize number of orthogonal edges: allowing only strictly horizontal or vertical edges leads to a cleaner image;

- Minimize node overlap: overlap makes nodes harder to distinguish visually;

- Minimize edge length: long edges and paths are harder to track visually;

- Minimize total drawing area: inefficient use of drawing area is undesirable;

- Optimize aspect ratio: very elongated diagrams do not fit well on a computer screen;

- Maximize symmetry: a clear display of symmetrical parts in a graph facilitates the creation of a mental map;

- Minimize edge length variance: uniform edge lengths create a cleaner image.

Optimizing all of the above aesthetics simultaneously is impossible because they often conflict with each other. Apart from that, there are many domain specific criteria that may govern the final layout of the graph. Several experiments to determine which aesthetics are most important have been conducted [135, 176]. Edge length, path continuity and edge crossing have been found to have the most influence on the cognitive cost of evaluating a graph. Many heuristics and algorithms have been designed to optimize the above factors and since not all of them are suitable for all types of graphs and we will discuss them by graph class.
3.1 Graph Drawing

3.1.2 Rooted Trees

Some of the earliest algorithms focused on trees, since the absence of cycles allows for elegant recursive algorithms. The use of manually drawn trees throughout history has led to a number of generally accepted aesthetics for a rooted tree drawing:

- The layout exhibits a layering of the nodes, that is, the vertical or horizontal distance from a node to the root is equal to its depth (continuity).
- Edges are rendered as straight lines and do not cross (crossings, bends).
- A parent should be centered over its children (symmetry, length).
- There exists a fixed minimal horizontal and vertical distance between nodes (overlap, length).
- Isomorphic subtrees have identical visual representations (symmetry).

One of the earliest algorithms is by Knuth [106], using only text characters, since graphical displays didn’t exist at that time. An efficient and effective algorithm to draw trees that tries to minimize the total area of the drawing and conforms to the aesthetics above was given by Reingold and Tilford [140]. Their algorithm runs in $O(N)$ time and performs reasonably for most trees, but in the worst case takes up $O(N^2)$ area. It is shown to be NP-complete [160] to minimize the area of a tree drawing if all of the above aesthetic criteria are obeyed.

Another notable tree drawing convention is the use of radial layouts [30, 50]. Radial layouts map the depth of the node to a set of concentric circles and position the root of the tree at the center of this set. Subtrees are recursively drawn within a subsection (wedge) of the circles.

Less well known layouts are orthogonal grid layouts, in which a node can only be positioned on a gridpoint directly adjacent to its parent. Depending on the number of gridpoints used, orthogonal grid layouts can be classified as hv-layouts, T-layouts or H-trees. With a maximum of four gridpoints available we can easily see that the maximum degree of our tree can be no more than four, if we wish to render nodes as single points. This makes these layouts practically unusable for information visualization.

We can apply tree drawing algorithms to general graphs by extracting a spanning tree first. A spanning tree is a subgraph of $G$ that contains all nodes of $G$ and does not contain cycles. We can then layout the spanning tree of the graph and use the resulting node positions for displaying the entire graph.

3.1.3 Directed Acyclic Graphs

Directed Acyclic Graphs lend themselves well to a layered layout. This method was pioneered by Sugiyama et al. [159] and involves a three step process:

- Layering: each node is assigned to a horizontal layer. A straightforward layering is longest path layering, in which each node is assigned a layer that is equal to the
Figure 3.1 Different tree layout methods: (a) Reingold-Tilford; (b) Radial; (c) Orthogonal.

longest path length from a source to that node (continuity, length), although other layerings are also possible [43]. Edges that cross multiple layers are broken up by inserting dummy nodes along the edge.

• Crossing reduction: all (dummy) nodes in a layer are reordered such that the number of edge crossings between two layers is minimized (crossings). The problem of finding an ordering such that the number of crossings is minimal is NP-complete. As a result a number of heuristics have been developed, see [21] for an overview.

• Coordinate assignment: y-coordinates can simply be assigned based on the layers. We need to compute x-coordinates for each vertex, such that the ordering computed in the previous step is preserved and arcs can be drawn with minimal bends (bends, overlap). This constraint can be expressed as a quadratic programming problem and subsequently minimized.

Layered layouts offer a clear view of the intrinsic node ordering and can be computed efficiently. They are especially attractive if vertices share an ordered attribute (for example time). They can also be employed to visualize arbitrary digraphs by reversing the direction of a number of arcs to create an acyclic graph, laying out the graph using the above method and subsequently reversing the arcs back. However, finding the minimal set of such arcs is again NP-complete, so a number of heuristics have been developed [138, 29].

3.1.4 General Graphs

Force Directed Methods

Force directed layout methods model the graph as a collection of repulsive particles (nodes) that are connected by a set of attracting springs (edges). This approach is popular because it is easy to implement and modify and results look pleasant for a variety of graphs.
3.1 Graph Drawing

One of the earliest force directed methods (and one of the earliest graph drawing methods in fact) is the Barycenter method from Tutte [169, 170], who only defined attractive forces. To avoid the trivial solution, in which all nodes are placed at the same position, the position of a minimum of three nodes is fixed on the corners of a convex polygon. The position of the remaining nodes can then be calculated analytically. In practice this amounts to positioning unplaced nodes at the average position (or barycenter) of their already placed neighbors.

Force directed algorithms initially made their way from multidimensional scaling and VLSI layouts to graph drawing. Multidimensional scaling maps a set of high dimensional points to a set of lower dimensional points, while attempting to preserve the distances between points. Early algorithms [109] (rediscovered later [101]) follow the same approach and map graph theoretic distance to Euclidean distance. In the case of unconnected nodes the graph theoretic distance is set to a large number.

The first spring embedder that used both an attractive and a repulsive force model was presented by Eades [49]. Edges exert an attractive force on node pairs that is proportional to the log of the edge length (length) and nodes repulse each other with an inverse square law (overlap). The total energy in the entire system is then minimized by initially assigning random positions to all nodes and iteratively moving nodes in the direction of the force exerted upon them. Note that there is no guarantee that this minimum will be the global minimum. However, as a result of the optimization process the resulting embeddings often display symmetries clearly (symmetry).

Variations on this basic scheme have been proposed, amongst others by Fruchterman and Reingold [62] who proposed a square attractive force and an repulsive force that is inversely proportional to the distance between nodes. They also introduce a cooling schedule that limits the maximum possible displacement of a node as a function of the number of iterations performed. A related optimization technique, simulated annealing,
was employed in [44], resulting in better minima at the expense of a longer running time.

Other models tried to capture various aesthetics in the force model by including terms that capture the number of edge crossings [44, 167], global edge orientation [158], orthogonal edges [95] or node position constraints [87].

One problem with force directed layout methods is their stability. Since the layout is the result of an iterative optimization procedure, small changes in the initial state (such as a different initialization or a few extra edges) can lead to radically different visualizations. This makes it hard for a user to construct and maintain a consistent mental map of the entire structure.

Another problem with classical force directed layout models is that they require $O(N^2)$ time per iteration due to the node-node repulsions. Since the total number of iterations is estimated to be proportional to the size of the graph, this amounts to a total running time of $O(N^3)$. Improvements have been made by using methods from N-body simulations [20, 168, 137], decreasing the time per iteration to $O(N \log N)$ without notably affecting the quality of the layout.

Other approaches [84, 173, 65] use an algebraic multilevel approach to recursively coarsen the graph into a smaller graph and use the layout from the coarsened graph to compute a layout for the original graph. These methods are significantly faster than doing a layout of the whole input graph at once.

Despite the problems mentioned above, force directed layout methods are widely used and are flexible enough to deal with most kind of graphs. This makes them an often used choice in information visualization, where the structural properties on the input graph are not known a priori.
Topology-Shape-Metrics Algorithms

Topology-Shape-Metrics based algorithms use a three step algorithm to lay-out general graphs in an orthogonal way:

1. **Planarisation**: The graph is transformed to a planar graph by determining a set of edge crossings and substituting them with dummy nodes. Since determining such a minimal set is NP-complete, current algorithms extract and draw a maximal planar subgraph \([100]\) and then rout the remaining edges around that graph \([104]\).

2. **Orthogonalisation**: The resulting drawing is made orthogonal by converting each edge to a set of horizontal and vertical line segments. Algorithms usually try to minimize the number of bends \([60]\).

3. **Compaction**: Final vertex positions are assigned based on a specific metric. The most common metrics are minimal total edge length or drawing area.

Orthogonal drawings are widely used in diagramming and schematics, making Topology-Shape-Metrics based approaches very suitable for laying out UML diagrams or database schemas, for example. However, the complexity of the planarization step makes their application in large scale graph drawing difficult.

Spectral Graph Drawing

A different approach is taken by spectral graph drawing algorithms. Denote the *Laplacian* matrix \(L\) of a weighted graph \(G\) as

\[
L_{ij} = \begin{cases} 
\sum_{k=1}^{n} w_{ik} & \text{if } i = j \\
-w_{ij} & \text{if } i \neq j
\end{cases}
\]

where \(w_{ij}\) represents the weight of edge \(e_{ij}\). Denote the x-coordinates that are assigned to a node \(n_i\) as \(x_i\). Minimizing the total squared weighted edge length in a one-dimensional drawing then corresponds to minimizing:

\[
\frac{1}{2} \sum_{i,j=1}^{n} (x_i - x_j)^2 w_{ij} = \sum_{i,j=1}^{n} L_{ij} x_i x_j = x^T L x
\]

To avoid the trivial solution in which \(Lx = 0\), the extra constraint \(|x|^2 = 1\) (or, equivalently, \(x^T x = 1\)) is added. Using Lagrange multipliers this minimization can be reformulated as \(x^T Lx - \lambda x^T x\). Taking the derivative and setting this to 0, gives

\[
Lx = \lambda x
\]

which is an eigenvalue problem with solutions \(\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_d\). Hall \([74]\) showed that assigning the eigenvector \(\mu_2\) corresponding to the second smallest eigenvalue \(\lambda_2\) to \(x\), results in an optimal continuous one-dimensional placement with minimal total squared
Table 3.1 Overview of the most important graph layout algorithms

<table>
<thead>
<tr>
<th>Method</th>
<th>Graphs</th>
<th>Aesthetics</th>
<th>Complexity (WC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruchterman-Reingold</td>
<td>Trees</td>
<td>Cross, Bends, Cont. Sym., Overl., Var., Length</td>
<td>$O(N^2)$ - $O(N^3)$</td>
</tr>
<tr>
<td>Radial layouts</td>
<td>Trees</td>
<td>Bends, Cont. Sym., Overl., Var., Length</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>hv-layouts</td>
<td>Trees</td>
<td>Cross, Bends, Var., Ortho., Length</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Sugiyama</td>
<td>DAGs</td>
<td>Cross, Bends, Var., Length</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>Naive Force-Directed</td>
<td>All</td>
<td>Length, Overl., Sym.</td>
<td>$O(N^3)$</td>
</tr>
<tr>
<td>Barnes-Hut Force-Directed</td>
<td>All</td>
<td>Length, Overl., Sym.</td>
<td>$O(N^2 \log(N))$</td>
</tr>
<tr>
<td>Multiscale</td>
<td>All (especially gridlike)</td>
<td>Length, Overl., Sym.</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>High dimensional embedding</td>
<td>All (especially gridlike)</td>
<td>Length, Overl., Bends, Ortho</td>
<td>$O(N) - O(N^2)$</td>
</tr>
<tr>
<td>TSM-Orthogonal</td>
<td>All</td>
<td>Length</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>Spectral</td>
<td>All (especially gridlike)</td>
<td>Length</td>
<td>$O(N^2)$</td>
</tr>
</tbody>
</table>

edge length $\lambda_2$. The third smallest (and orthogonal) eigenvector ($\lambda_3$) represents the next-best solution and we can assign it to the nodes’ y-coordinates to obtain a two dimensional embedding.

In terms of force directed placement we can view spectral graph drawing as a method that optimizes edge lengths, under the assumption of a global repulsive force acting from the center of the drawing. For gridlike graphs the results are quite good, but the results on trees are worse because there is no local repulsive force keeping the branches apart. This limits their application in information visualization, although a couple of approaches seem to provide reasonable results [35, 107]. On the other hand, spectral methods can be fast on sparse graphs (when using for example a Lanczos approximation to find the eigenvectors) and have a solid mathematical background.

**High dimensional Embedding**

Another recent promising approach introduced by Harel and Koren [86] first creates a high dimensional embedding by picking $n$ (say $n=50$) pivot nodes that cover the graph as well as possible. Each node in the graph is then mapped to to an $n$-dimensional vector that describes the graph theoretic distances to every pivot node. This $n$-dimensional collection of points is projected to two dimensions by using principal component analysis. The method is fast and also allows for zooming by recomputing the PCA for the selected subset. Results on regular graphs can be quite good but unfortunately the method does not provide very good results on the more diverse class of real world graphs.

Table 3.1 shows that all but the most elementary graph drawing algorithms have more than linear complexity, which makes application of these methods to large graphs a costly affair. And even if it were possible to create layouts in linear time, images of graphs of several thousands of nodes are impossible to comprehend. Another often used approach therefore is to first reduce the size of the graph by clustering. Graph clustering is a technique that is relevant for a wide variety of application areas and has received a large amount of attention. In the next section we give a comprehensive overview.

---

1Since these methods consist of multiple algorithmic steps, the overall complexity depends on the worst-case complexity of its individual steps. Some variants can provide better results at the expense of higher computational complexity.
3.2 Graph clustering

Graph clustering concerns itself with finding an optimal clustering (or grouping) of the nodes of the graph and is applied in many different scientific contexts. As such, prior work is substantial and spread out over different scientific disciplines, from mathematics [41] and physics [126] to electronics [147] and sociology [34]. Currently there are two major research subfields in graph clustering, which differ in the amount of information they presuppose and in how they define ‘optimal clustering’.

*Graph partitioning* is an area that assumes the number of target clusters is given and defines optimal as ‘having a minimal number of edges between clusters’. Graph partitioning has its roots in VLSI design and parallel computations, in which the objective is to assign nodes to a given number of entities (circuits or computers) such that the communication needed between nodes is minimal. The noun *partition* implies that resulting clusters are disjoint and that their union covers the complete graph.

![Figure 3.4](Image) How many natural clusters are there in this image? Image from [48]

The much younger field of *natural graph clustering*, defines ‘optimal’ as ‘representing natural groups’. The problem in this definition lies in the word ‘natural’, which is hard to define mathematically. In fact, even humans have problems identifying natural clusters as is witnessed by figure 3.4. The number of clusters is not known in advance, and depends on the scale of observation. Natural graph clustering is of greater interest to information visualization because it does not require a target number of clusters as input.

In the next two sections we give an overview of techniques and methods from both areas. An extensive survey on partitioning can be found in [13].
3.2.1 Graph partitioning

Problem formulation

We can formulate the graph partitioning problem as follows: compute a partition $P_k$ of the nodes of the graph into $k$ disjoint clusters $C_1, C_2, \ldots, C_k$, such that the number of edges between subsets is minimal. To formalize this, we first define the cutsize $\text{Cut}(C_k)$ of a cluster $C_k$ as

$$\text{Cut}(C_k) = |\{ e_{xy} \in E : (x \in C_k \land y \not\in C_k) \lor (x \not\in C_k \land y \in C_k) \}|$$

or, in words, the number of edges running between $C_k$ and any other cluster. The total cutsize for a partition $P$ can then be expressed as

$$\text{Cut}(P_k) = \sum_{i=1}^{k} \text{Cut}(C_i)$$

In its simplest form ($k = 2$) the Min-Cut problem amounts to minimizing $\text{Cut}(P_2) = \text{Cut}(C_1) + \text{Cut}(C_2)$ such that $C_1 \neq \emptyset$ and $C_2 \neq \emptyset$. Solutions can be computed efficiently [83], but the sizes of the clusters are often very unbalanced. A formulation that keeps the sizes of cluster approximately equal is Min-Cut Bisection that minimizes $\text{Cut}(P_2)$ such that $\text{Abs}(|C_1| - |C_2|) \leq \varepsilon$, but has been shown to be NP-complete. Other popular bi-partitioning formulations that keep the size of the clusters balanced (but not necessarily equal) are Size constrained Min-Cut Bipartitioning that minimizes $\text{Cut}(P_2)$ such that $L \leq |C_i| \leq U$ for $i = 1, 2$ and Minimum Ratio Cut Bipartitioning that minimizes the ratio $\text{Cut}(C_1)/|C_1||C_2|$.

All of these formulations can be generalized to $k$-way partitions in a straightforward fashion, but optimizing them is again NP-complete. A number of optimization heuristics have been proposed instead, which we discuss next.

Iterative optimization

One of the first iterative optimization techniques for bisections is attributed to Kernighan and Lin [103]. The Kernighan-Lin algorithm performs a number of passes starting from a random balanced bisection. Each pass returns a bisection that has lower cost than the starting bisection by greedily swapping a pair of nodes. This bisection then serves as input for the next pass and the algorithm ends when such a bisection can no longer be found.

During a single pass, the algorithm evaluates all node pairs $(i, j)$ (with $i \in C_1$ and $j \in C_2$) that have not been swapped yet, and swaps the one with the highest gain. The gain of a swap is defined as the difference between the values of the objective function before and after the swap. The resulting value of the objective function is stored and this process is repeated until all nodes have been swapped at most once. The partition with the lowest observed value of the objective function is then returned.

A straightforward implementation of Kernighan-Lin requires $O(N^3)$ time. Fiduccia and Mattheyses [58] improved on this by using specialized data structures, reducing the
time per pass to \( O(E) \). Sanchis [144] extended their algorithm to multi-way partitions, and is currently the most widely used multiway partitioning algorithm. Another often used way of computing multi-way partitions is by recursive application of a bisection algorithm.

### Spectral clustering

Suppose we encode a bipartitioning of a weighted simple graph as a vector \( \mathbf{p} \), with \( p_i = 0 \) if \( n_i \in C_1 \) and \( p_i = 1 \) if \( n_i \in C_2 \). Let \( w_{ij} \) represent the weight of an edge \( e_{ij} \), \( w_{ij} \) is equal to 1 for unweighed graphs. We can then formulate the Min-Cut problem as a minimization of:

\[
\frac{1}{2} \sum_{i,j=1}^{n} (p_i - p_j)^2 w_{ij}
\]

Note that this is the exact same formulation as presented in section 3.1.4 on spectral graph drawing above. The optimal solution is the second smallest eigenvector \( \mu_2 \) of the Laplacian, representing optimal one dimensional coordinates. These can be transformed to a bipartitioning with sizes \( |C_1| \) and \( |C_2| \) by assigning the \( |C_1| \) nodes with the highest coordinates to \( C_1 \) and the rest to \( C_2 \) (or vice versa).

This approach can be extended to k-way partitioning by also incorporating \( \lambda_3, \lambda_4, ..., \lambda_{k+1} \). These eigenvectors can then be mapped directly to \( k \) clusters [19] or used to create a \( k \)-dimensional embedding which can then be clustered using existing geometric algorithms [12], such as k-means. An alternative method is Recursive Spectral Bisection [150] which uses the result from the initial bisectioning as input for a new run.

### Agglomerative clustering

Instead of recursively applying partitioning algorithms to a dataset, as is the case in Recursive Spectral Bisectioning, we can also recursively aggregate nodes into clusters [99]. If we start with a clustering in which each node is assigned to a single cluster, we can then select the two clusters that are closest according a distance measure between clusters and merge them into a new cluster. Iteration of this process results in a binary tree of clusters, also known as a dendrogram. Various cluster distance measures can be used here. Given two sets of nodes \( C_1 \) and \( C_2 \), define the set \( D \) as

\[
D(C_1, C_2) = \{ d(i, j) : i \in C_1, j \in C_2 \}
\]

where \( d(i, j) \) denotes the (for example Euclidean) distance between nodes \( i \) and \( j \). For the distance between \( C_1 \) and \( C_2 \), we can then either use the minimum value in \( D \) (minimum link clustering), the maximum value in \( D \) (maximum link or complete link clustering) or the average value of \( D \) (average link clustering). Minimum link clustering has the tendency to form long chains of clusters, while maximum link clustering is sensitive to cluster outliers. Average link clustering provides a good compromise between the two. Agglomerative clustering can provide good results, but selecting an optimal clustering from the dendrogram can be difficult.
Multilevel implementations

Multilevel partitioning implementations (see also 3.1.4) operate in three phases:

1. First of all the original graph $G_0$ is coarsened into a sequence of smaller graphs $G_1, G_2, \ldots, G_L$.

2. The smallest graph $G_L$ is then partitioned into $k$ parts.

3. These parts are then projected back from $G_j$ to $G_{j-1}$ to obtain a partition of $G_0$.

Coarsening is usually done by finding a set of edges to be contracted, such that no two edges are incident to the same vertex. Such a set is also called a matching. A maximal matching is a matching that cannot be extended by adding an edge. Each matching reduces the size of the graph by a factor of at most $1/2$.

A bipartitioning algorithm, such as Fiduccia-Mattheyses [58] or spectral bipartitioning, is then applied to $G_L$. The result of this step is then projected back (uncoarsened) into $G_{L-1}$ and a limited number of iterations of the Kernighan-Lin algorithm is performed on the boundary vertices to further optimize this cut. This uncoarsening process is then repeated until we obtain a cut of the original graph.

Multilevel algorithms such as [102, 18] can provide decent partitionings of graphs of tens of thousands of nodes in seconds. The implementations involved are usually complex, but luckily a number of partitioning packages are readily available, the most well-known being CHACO [88], JOSTLE [152] and MeTiS [102].

3.2.2 Natural graph clustering

The definition of the problem in natural graph clustering is much harder since there is no exact definition of the concept of a ‘natural’ cluster. There are however intuitive (and similar) notions of what the properties of a natural cluster in a graph $G$ are [48]:

- **Path count**: The number of paths between vertices $i$ and $j$ that have length greater than 1 is large if $i$ and $j$ lie in the same dense cluster and small if $i$ and $j$ belong to different clusters.

- **Probabilistic**: A random walk in $G$ that visits a dense cluster will likely not leave the cluster until many of its vertices have been visited.

- **Edge frequency**: Considering all shortest paths between all pairs of vertices in $G$, links between different clusters are likely to be in many shortest paths.

All of the formulations above give rise to different clustering algorithms, which we discuss below.
3.2 Graph clustering

Sample graph $G$

\[
\begin{pmatrix}
3 & 4 & 7 & 8 \\
1 & 2 & 5 & 6
\end{pmatrix}
\]

\[
\begin{pmatrix}
4 & 4 & 4 & 1 & 0 & 1 & 0 \\
4 & 5 & 4 & 4 & 2 & 1 & 2 & 0 \\
4 & 4 & 4 & 5 & 2 & 1 & 2 & 1 \\
1 & 2 & 1 & 2 & 4 & 3 & 3 & 2 \\
0 & 1 & 0 & 1 & 3 & 4 & 4 & 3 \\
1 & 2 & 1 & 2 & 3 & 4 & 5 & 3 \\
0 & 0 & 0 & 1 & 2 & 3 & 3 & 3
\end{pmatrix}
\]

$G$'s adjacency matrix $A_G$

\[
\begin{pmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{pmatrix}
\]

Resulting clustering with $m = 3$

\[
\begin{pmatrix}
3 & 4 & 7 & 8 \\
1 & 2 & 5 & 6
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 4 & 4 & 1 & 0 & 1 & 0 \\
4 & 5 & 4 & 4 & 2 & 1 & 2 & 0 \\
4 & 4 & 4 & 5 & 2 & 1 & 2 & 1 \\
1 & 2 & 1 & 2 & 4 & 3 & 3 & 2 \\
0 & 1 & 0 & 1 & 3 & 4 & 4 & 3 \\
1 & 2 & 1 & 2 & 3 & 4 & 5 & 3 \\
0 & 0 & 0 & 1 & 2 & 3 & 3 & 3
\end{pmatrix}
\]

$G$'s adjacency matrix $A_G$

\[
(I + A_G)^2
\]

Path count

An early algorithm [119, 161] based on path count is an extension of single-link clustering known as $k$-path clustering. In the adjacency matrix $A_G$ of a simple unweighted graph $G$, entry $a_{ij}$ gives us the number of paths with length 1 between two nodes $i$ and $j$. In general, the number of paths with length $k$ is given by $A_G^k$ (the $k$th power of $A$).

Commonly $(I + A_G)^k$ instead of $A_G^k$ is used, to avoid issues with the parity of $k$. This corresponds to adding a self loop to each vertex in the graph. The matrix $(I + A_G)^k$ is then converted to a clustering by defining a minimum desired number of paths $m$ and setting all off-diagonal entries of $(I + A_G)^k$ smaller than $m$ to 0. Connected components can then be computed on the resulting matrix (see figure 3.5).

This basic approach has a number of drawbacks however, the most notable being that the matrix $(I + A_G)^k$ fills up very quickly, making iterative multiplication a costly affair in terms of both memory and computation. Another drawback is that smaller clusters get ‘swallowed’ by bigger clusters too quickly.

A more recent method called Markov Clustering or MCL [48] avoids these pitfalls. The basic principle remains the same, but instead of iteratively multiplying adjacency matrices, Markov matrices are multiplied. Assuming a random walk on the graph, the entry $m_{ij}$ of a Markov matrix $M$ denotes the probability that a path to $i$ is taken given that the current position in the graph is node $j$. $M$ can be computed directly from $A_G$ by normalizing $A_G$'s columns, that is $m_{ij} = a_{ij} / \sum_k a_{kj}$.

Each multiplication is followed by an inflation operation that further boosts the probability of higher-probability neighbors. Algebraically speaking this amounts to raising each element in $M$ to a power $p$ and then renormalizing the matrix columns, i.e. $m_{ij}$ is replaced by $m_{ij}^p / \sum_k m_{kj}^p$. In most cases alternating multiplications and inflations leads
to a stable matrix from which we can deduce a (potentially overlapping) clustering (figure 3.6). A higher clustering granularity can be obtained by increasing $p$. To avoid matrices growing dense, entries smaller than $\varepsilon$ are pruned and the column is subsequently renormalized. Pruning schemes keep the matrix multiplication sparse and make the application of MCL to graphs of thousands of nodes feasible.

The main advantages of MCL are the simplicity and elegance of its basic formulation, its dependance on only one free parameter ($p$) that governs the resulting cluster granularity and its scalability. In practice, MCL is applied in different contexts, most notably in gene clustering packages [53, 113].

**Probabilistic methods**

Probabilistic methods are based on the observation that if a random walk originating at node $i$ shares many points with a random walk originating at node $j$, nodes $i$ and $j$ are likely to be in the same cluster. A random walk is a path generated by a process that starts at a node $i$ and iteratively moves to a random adjacent node $j$, with each adjacent node having equal probability. One of the first approaches [73] computes a random walk of length $|N|^2$ and extracts all the cycles in the walk. Next, a node similarity measure $S(i, j)$
is computed that is based on the commonality of the sets of nodes that are visited in cycles originating at \(i\) and \(j\) respectively. Nodes with \(S(i, j) > 0\) are then merged into a single cluster. This method is computationally expensive due to the explicit simulation of the random walk.

The authors of [85] take a similar approach but compute probabilities deterministically. They define \(P(k)(i)\) as the vector whose \(j\)th element contains the probability that a random walk originating at \(i\) visits node \(j\) in at most \(k\) steps. Two nodes \(i\) and \(j\) have similar neighborhoods if \(P(k)(i)\) and \(P(k)(j)\) are correlated and for \(S(i, j)\) the correlation coefficient is used.

The resulting similarities \(S(i, j)\) are mapped to the edges of the graph and edges with low similarity are subsequently removed. The resulting connected components form a cluster structure. This method is quite fast because the size of neighborhoods is limited to a small constant \(k\), meaning that not all probabilities have to be calculated. On the downside, this also imposes constraints on the maximum diameter of detected clusters.

**Structural methods**

Structural methods are based on the observation that edges connecting clusters will be in a large number of shortest paths because they are few in number. The total number of shortest paths an edge is in is represented by the *edge betweenness* measure in social network analysis. Edge betweenness can be computed in \(O(NE)\) by performing an \(O(E)\) breadth first search for every node in the graph. A simple hierarchical clustering algorithm [69, 126] then computes the betweenness for all edges, removes the edge with highest betweenness, recalculates the betweenness for the other edges and repeats these steps until all edges are gone. The resulting connected components form the clusters of the graph. The method provides good results and is hierarchical, but the betweenness recalculiation after each edge removal results in a total worst-case complexity of \(O(E^2N)\) making application to larger examples infeasible.

Another algorithm [125] that can be classified as both structural and probabilistic uses a different approach. Given an undirected graph \(G = (V, E)\) and suppose its vertices are partitioned into \(n\) clusters. The fraction of edges connecting nodes in cluster \(i\) to nodes in cluster \(j\) is given by \(f_{ij}(0 \leq f_{ij} \leq 1)\):

\[
f_{ij} = \frac{|\{e_{xy} \in E : x \in i \wedge y \in j\}|}{2|E|}
\]

The total fraction of edges connected to nodes in cluster \(i\) is then given by \(g_i = \sum_{j=1}^{n} f_{ij}\). Note that \(\sum_{i=1}^{n} g_i = 1\).

If the edges in a graph are distributed randomly over the graph, the expected fraction \(F_{ij}\) of edges running between \(i\) and \(j\) is equal to \(g_ig_j\). The quality of a cluster \(i\) can then be measured by looking at the difference between the actual number of edges that fall within \(i\) and the number of edges that should fall within \(i\) if the distribution were random:

\[
Q(i) = f_{ii} - F_{ii} = f_{ii} - g_i^2.
\]

Define the partition quality \(Q_G\) of the graph as the sum of those
differences over all clusters

\[ Q_G = \sum_{i=1}^{n} Q(i) = \sum_{i=1}^{n} (f_{ii} - g_i^2) \]

The optimal partition is the partition with the highest value of \( Q_G \). Because evaluation of all possible partitions is computationally impossible, the author of [125] suggests using a greedy agglomerative clustering scheme that selects the edge with the highest change in \( Q_G \) and merges the two clusters \( i \) and \( j \) connected by this edge. In this case

\[ \Delta Q_G = Q(j \cup i) - (Q(i) + Q(j)) = f_{ji} + f_{ij} - 2g_i g_j = 2(f_{ij} - g_i g_j) \]

This creates a dendrogram of clusters, in which the partition with the highest occurring value of \( Q \) is optimal. The whole algorithm runs in time \( O((E + N)N) \). In [125] results on a number of real world graphs are provided, suggesting that this method may be well-suited for information visualization.

Although clustering methods can transform big problems into a number of smaller ones, they also introduce new problems. Without proper navigation mechanisms the user quickly loses context when exploring a clustered dataset and the lack of available screen space also poses problems. Both these issues fall in the area of information visualization, which we explore in the next section.
3.3 Information Visualization

One of the main user interface problems in visualizing large graphs is that there is not enough display space available to accommodate all the nodes in a node-link layout of large graphs. Obvious solutions are using the screen more efficiently or increasing the amount of screen space by using geometrical spaces that offer more room, such as hyperbolic or 3D. Another (orthogonal) approach is to use interaction techniques to offload some of the complexity to the time dimension and only present the user with a smaller subset of the data at any given time. In the next sections we give an overview of techniques in both of these categories, as well as a comprehensive list of graph visualization tools currently available. For further information we refer the reader to the survey by Herman et al. [89].

3.3.1 Increasing display space

A number of approaches exist to deal with the scarcity of display space. One approach is to use the display space that is available more effectively. Node link diagrams use display space inefficiently, as they contain a lot of white space. Treemaps are the main proponent of this approach. Another way to increase the amount of display space available is to use hyperbolic space. Finally, a popular method is to use a virtual 3D space instead of 2D. We discuss these three approaches below.

Treemaps

Treemaps, invented by Johnson and Shneiderman in 1991 [98], are rectangular displays of rooted trees that use the available visualization space very efficiently. They also encode the value of one quantitative leafnode attribute $A$ in a node’s screen size. We assume that the value of $A$ for an internal node is the sum of the values of its childnodes.

The basic treemap algorithm is straightforward: An arbitrary rectangle representing the root node of the tree is horizontally partitioned into smaller rectangles, where each smaller rectangle represents a child $c_i$ of the root node. The area of the smaller rectangle is proportional to $A(c_i)$. This process is then repeated recursively, and the direction of the subdivision alternates between horizontal and vertical (see figure 3.3.1).

Treemaps are especially useful for the display of large hierarchical datasets in which the attribute $A$ of a node is important. As the size of each rectangle is proportional to this attribute, one can easily spot the largest nodes in the tree. Because they use every available pixel on the display, treemaps scale very well, and have been applied to trees with a million leaf nodes [57].

However, standard treemap representations have two problems. Firstly, the method often leads to high aspect ratios (i.e. long thin rectangles). These make it more difficult to compare sizes and lead to interaction problems. Secondly, since the leaf nodes take up all of the visualization space, no space remains for the internal nodes of the tree. This makes it difficult to reconstruct the hierarchical information from the treemap, especially when dealing with large, deep trees.
Recent research has tried to provide answers for both of these problems. New subdivision schemes have been developed by [36, 171], producing better aspect ratios but worsening layout stability. With stability we mean the visual representation’s resistance to small changes in the tree structure, such as increasing the size of a single node. Other algorithms [148, 27] improve the aspect ratio to a lesser extent, but offer greater stability and a consistent ordering of the items in the treemap.

Quantum treemaps [27] offer a quantized version of any subdivision scheme. Instead of an attribute value $A$, each leaf node is assigned a list of items of fixed size. The algorithm then computes a size for each rectangle such that it can at least display the specified number of items. A bubblemap [24] is an extension of Quantum treemaps that is not limited to rectangular shapes.

Methods to improve the display of hierarchical information include nesting [98, 171], line width, color and, more recently, shading [180]. However, these methods still require a significant cognitive effort to extract the actual tree structure from the picture, especially when compared to the traditional node-link visualization. The use of nesting also affects the area of nodes deeper in the hierarchy since more space has to be reserved for the borders. This makes it more difficult to accurately estimate the value of $A$ for a node.

Treemaps have been used successfully in real-world applications to visualize the stock market [151], hard disk usage [163] and image collections [24].
Hyperbolic Layouts

Another way to increase the amount of display space available is by using hyperbolic geometry instead of traditional Euclidean geometry. Hyperbolic geometry is similar to Euclidean geometry, but without its fifth postulate, which states that if a line $l$ does not intersect a point $p$ there exists only one (parallel) line that intersect $p$ and does not intersect $l$. In hyperbolic geometry there are infinitely many of such lines.

Since we cannot display hyperbolic space directly (in our Euclidean world) a number of mappings from hyperbolic to Euclidean space have been developed. The most well known are the Klein-Beltrami and Poincaré models, which map the entire hyperbolic space to a unit disk (or a sphere in 3D) with the perimeter of the disc representing infinity. Infinite lines in hyperbolic space are represented by chords of the disc. The Klein-Beltrami model is projective and preserves straight lines, the Poincaré model is conformal and preserves angles between lines.

In hyperbolic space the circumference of a circle expands exponentially with its radius, instead of linearly. This makes hyperbolic spaces naturally suited for displaying trees, where the number of items generally increases exponentially with depth.

Hyperbolic graph layout algorithms operate by applying a graph layout algorithm in hyperbolic space and then mapping the results back to Euclidean space. Lamping et al. [110] visualize large trees by first applying a radial layout and subsequently mapping the results back using the Poincaré model. Munzner [123, 124] uses a variation of cone-trees (see 3D layouts in the next section) to layout a spanning tree of a graph in hyperbolic space and maps the result using the Klein model. Figure 3.10 shows examples of these methods.

Another advantage of hyperbolic projections is that they compress the space farther
Figure 3.9 Mapping hyperbolic space to Euclidean space: (a) Klein-Beltrami model (b) Poincaré model. All shown lines are considered hyperbolically parallel, except for m and l.

Figure 3.10 Hyperbolic graph layouts: (a) 2D hyperbolic layout of an organization tree (b) 3D hyperbolic layout of a software call graph. Images from [110] and [123], respectively.
away from the origin. This gives the added bonus of a focus and context display (see section 3.3.2), allowing the user to see the area near the origin of the hyperbolic space in closeup. Navigation can easily be implemented by translating the graph in hyperbolic space and re-projecting the structure.

3D layouts

Another widely used method in visualization to increase the amount of display space is the use of interactive 3D graphics. The use of a three-dimensional instead of a two-dimensional visualization space seems to have a number of advantages:

- 3D offers more visualization space. By adding an extra dimension to the visualization space, we can space items farther apart or use the extra dimension to display additional information.
- The world surrounding us is three dimensional and we are used to create mental maps of three dimensional spaces. Why not make use of this ability and create 3D visualizations?
- 3D graphics look attractive and interesting. Users will initially be more inclined to work with a ‘sexy’ looking interface.

Apart from these conceptual advantages, 3D visualizations are often also easy to implement because most 2D graph layout methods can be easily extended to 3D and the implementor is protected from the low-level 3D implementation issues by well-structured 3D API’s coupled with powerful computer graphics hardware.

An early successful example of a 3D visualisation are conetrees [143] in which the childnodes of a parent $p$ are distributed equally over the base of a virtual cone with its apex at $p$. Figure 3.11a shows a medium sized example of an organization hierarchy. Distant cones can be clicked and the visualization rotates them into view. Great attention has been paid to details such as the ground plane (gives the user a better sense of space), shadows (provides hints for the 3D structure) and animation (helps to maintain the mental map of the structure during rotation).

Other 3D visualizations are generalizations of existing 2D visualizations. Narcissus [88] is an example of a graph visualization system that extends force directed methods to 3D. Other methods include Information Cubes [141] and NV3D [130] that extend nesting to 3D by using translucent boxes.

Another option is to use the extra dimension to display additional node attributes or extra information. SGI’s file system navigator [146] renders the filesystem as a 2D tree and uses the third dimension to display file size. Steptree [32] is an application that extends treemaps to 3D by using an extra dimension to encode file depth, similar to beamtrees [76]. Information Landscapes [182] is a technique that uses the extra dimension to indicate the density of points in a 2D layout, forming a 3D landscape (Fig. 3.11b).

Much of the pioneering work on 3D visualization was done in the 1990’s but since then 3D has lost some of its original appeal due to some intrinsic difficulties:
Figure 3.11 3D visualizations: (a) Conetrees use a 3D arrangement of cones to display hierarchical data [143] (b) ThemeView from PNNL uses a landscape metaphor to indicate prominent themes [129].

- We can only render 2D projections of 3D scenes. This makes it difficult to assess the structure of the object without additional cues, such as motion.

- It is difficult to optimize aesthetics such as edge crossings, because of the infinite number of possible viewing directions. For the same reason it is very easy for the user to lose orientation when navigating.

- Because most common input devices are 2D, navigating in 3D space is more difficult.

- Our retina is inherently a 2D sensor [174], and is more susceptible to left-right and up-down cues than front-back cues.

- 3D images do not print well on paper, making it harder to communicate images from a visualization.

Because of these reasons some researchers [174] suggest that 2D, or $2\frac{1}{2}$D (that is, 3D optimized for a fixed viewpoint or 2D with shading) visualizations are generally better solutions for information visualization purposes. A carefully designed 3D visualization can still be very effective though.

### 3.3.2 Interaction

A lack of display space is not the only problem users are confronted with in large scale information visualization. A user cannot be expected to deal with a single image consisting of thousands of information elements. A number of interaction techniques have been
devised to deal with this problem. This section will discuss some of the most important ones.

**Panning and zooming**

The most simple and often used way to deal with an amount of items that does not fit on screen is to use panning or scrolling. Panning is widely used in windowed applications, such as internet browsers and text editors. It allows the user to move a window over a much larger virtual canvas and view a small portion of that canvas. Although the interaction technique is easy to understand and well known, using it for large graphs is often problematic. It is very hard for the user to maintain a sense of context when only a small part of the entire structure is in view.

An often used solution is to provide a smaller separate display that presents the user with an overview of the entire structure and indicates the current location of the window. A disadvantage of this approach in general is that the user constantly has to switch attention between the overview and detail windows. A specific problem for graphs is that two distant nodes might have an edge connecting them, while the detail window lacks resolution to render this.

Zoomable interfaces [26] can provide overview and detail in a single screen by allowing the user to view items at different scales, but cannot show both global and local information in a single view. A third option are fisheye techniques that allow information to be displayed at different scales at once.

**Fisheye techniques**

A well-known metaphor for navigation in information spaces is the *fisheye lens*. In real life fisheye lenses magnify portions of the object near the center of the lens and compress the objects near the edge of the lens. In information visualization fisheye techniques operate under the assumption that items closer to a user’s point of interest should be visualized more prominently than items farther away. Or in other words, fisheyes trade off distance and detail. This concept was generalized by Furnas [63], resulting in a formulation of a generalized fisheye view. Generalized fisheye views define a *Degree of Interest* or *DOI* function that assigns to each information element a numerical value that indicates how interested the user is in that element, given the current task. Furnas defined the *DOI* of a point *x*, given that the user is currently interested in a point *y*, as

\[
DOI(x|y) = API(x) - D(x, y)
\]

This splits the *DOI* in a fixed *A Priori Importance* for a point *x* and a variable part based on the distance *D(x, y)* between *x* and *y*. The visualization application can use this information to display only points above a certain *DOI*. The distance function *D* is not specified further, making this definition flexible enough to apply to many different kinds of structures. Examples include rooted trees where the *API* of a node depends on its depth in the tree and the function *D* denotes the graph theoretic distance between two points, or a calendar where weekdays have higher *API* and the distance *D* is measured in time.
Application of the fisheye paradigm to graphs generally fall into one of two categories. One category, distortion oriented techniques, operate in display space and geometrically distort the sizes and positions of nodes based on their geometric distance to the focus. This can be implemented by using hyperbolic layouts or by applying a geometrical distortion function to an existing layout as a post-processing step. An example of such a distortion function is

$$h(d) = \frac{D + 1}{D + \frac{1}{d}}$$

which scales a distance \(d\) to the focus with a factor \(h(d)\) [145]. This mapping assumes that all node coordinates are normalized to \([-1, 1]\) and the factor \(D\) denotes the amount of distortion present. This mapping can then be applied in cartesian coordinates by scaling \(x\) and \(y\) coordinates separately, or polar coordinates, by scaling only the radial component. Alternatively, the mapping can be applied in only one dimension to create a ‘perspective wall’ [116] effect.

The other category are data suppression techniques that operate in data space. These hide or abstract (groups of) nodes based on the distance to the focus. Although closer to Furnas’ original idea, relatively few researchers have pursued this path [56, 137], but the idea has been gaining momentum recently [4, 66, 81]. The advantage of data suppression techniques over distortion techniques is that only a relevant subset of the nodes has to be rendered, whereas distortion oriented techniques render all nodes on screen at once. Care has to be taken, however, in the creation of meaningful abstractions of groups of nodes and in maintaining the mental map of the graph when moving the focus.

**Incremental browsing**

Incremental browsing is a technique that uses concepts from both panning and fisheye navigation. It resembles panning in the sense that the user is given a limited view of a subsection of the graph. It is similar to data suppression fisheye views because they operate in data space and the window can only be moved by selecting a new focal node.
from the currently visible nodes. After selecting a new focal node the system generates a new view. Different methods to generate a new view from a previous one are available. Some record the entire visited set of focal nodes [185], others limit the visible set to the direct neighborhood of the focal node [91, 128]. The advantage of this approach is that the new set of nodes is often relatively small, so a layout can be created quickly. Creating a new layout without destroying the mental map of the old layout is challenging, but visual cues such as animation might help. The biggest disadvantage of incremental browsing is that it never offers a complete view of the whole graph, making it impossible for the user to answer global questions. It may be more suitable for local browsing of very large graphs (such as the WWW [166]), because it provides more lookahead than a hypertext system and also records the browsing history.

3.3.3 Graph Visualization Systems

Over the past two decades a number of integrated tools have been developed to tackle the problem of visualizing large graphs. Although it is impossible to name and discuss each of them separately we give a short overview of some of the more well-known implementations in this section.

A number of basic libraries for working with graph structures are available, some very general [59, 149] and some tuned specifically to graph drawing [164, 11]. A number of commercial graph visualization libraries and plug-ins are also available [165, 184, 9]. All of these tools operate on small to medium sized graphs (up to a few thousand nodes).

Full-fledged systems that can interactively visualize very large graphs are relatively scarce. One of the earliest systems was SemNet [56] that was used for analysis of knowledge bases. It was one of the first 3D graph visualization systems and also included a rudimentary fisheye implementation based on spatial subdivision. SemNet’s scalability was limited, but its design has influenced many other research systems.

Narcissus [88] uses a three dimensional spring embedder and rich graphical techniques to present the user with a three dimensional display of a graph. It was applied to software engineering tasks and navigation of the World Wide Web. Its scalability is limited due to the rich 3D graphics and the computational cost of the spring embedder.

A system offering higher scalability is Nicheworks [181] from Bell Labs. Nicheworks can deal with graphs up to 100,000 nodes. It uses optimized algorithms to create layouts in acceptable time, for example, by choosing a smart initialization and using a force directed algorithm without repulsive forces. A small number of local repulsions is then run afterward to avoid overlapping nodes. The graphics system has been tuned and renders only part of the data during user interactions to maintain interactive rendering speeds. Nicheworks has been integrated into the products of Visual Insights [172], a commercial spin-off of Lucent-Bell Labs.

Current record-holder in the area of graph visualization in terms of sheer size is MGV [5], developed at AT&T. MGV uses a predefined hierarchy on the node set to coarsen the graph. Different hierarchical slices in the graph can then be inspected using a variety of different visualizations. MGV runs on a client server architecture with multiple servers and is capable of handling graphs of up to 250 million nodes.
Chapter 4

Valuation of problem aspects

Since finding an analytic solution to the problem “Given graph $G$ and task $T$, find an optimal visualization” is impossible, we decided to select a number of cases from different application areas and design ‘good’ visualizations for each of them. Because we want to sample the problem space in the best possible way, we picked four independent cases from the collection of all possible problems. All cases vary in terms of the size and structure of the input graph, the type and number of attributes provided with the graph and the type of questions the user wanted answered. Table 4.1 gives an overview of the valuation for each of the properties defined in Chapter 2.

There is no variation for the parameter scope because graph visualization is better suited for answering questions on global structure than providing local details. The same goes for the parameter orientation, since data oriented questions are better answered by a database application and the answers to structure oriented questions are very hard to interpret without access to the data associated with nodes and edges. Some unavoidable correlation between fuzzyness and usage pattern is also present, since users asking concrete questions usually intend to use the visualization mainly for answering those questions. Apart from that, the four cases span the problem space reasonably well, with both fuzzy and well-defined questions on dense and sparse graphs, and all graphs having different graph characteristics.

In Chapter 5 we present beamtrees. As the name beamtrees indicates we confined ourselves to trees instead of general graphs. This means that we are dealing with sparse graphs with possibly a large diameter. Each node in the hierarchy corresponds to a directory or file on disk, and with each node we had information on file size and type. A typical modern day harddisk contains about $10^5$ files and $10^4$ folders.

We were aiming for a visualization that would allow novice users to spot larger files on a harddisk immediately, while at the same time showing the directory structure on the disk. This is a fairly well-specified question (“show the $x$ largest files”), but the difficulty lies in integrating the results within an overview of the entire hierarchy structure.
Chapter 4 Valuation of problem aspects

Table 4.1 Four different visualizations categorized on the properties described in Chapter 2. Dominant aspects are shown in bold.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Beamtree</th>
<th>State Vis</th>
<th>Matrixzoom</th>
<th>Small world</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>Tree</td>
<td>General Graph</td>
<td>General Graph</td>
<td>Small World Graph</td>
</tr>
<tr>
<td>Size (Nodes)</td>
<td>$10^5$</td>
<td>$10^6$</td>
<td>$10^6$ – $10^7$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>Average degree ($f(N)$)</td>
<td>Constant</td>
<td>Constant</td>
<td>Arbitrary</td>
<td>Constant</td>
</tr>
<tr>
<td>Characteristics</td>
<td>Hierarchy</td>
<td>Regular, symmetric</td>
<td>Locally clustered</td>
<td>Includes hierarchy</td>
</tr>
<tr>
<td>Diameter</td>
<td>Large</td>
<td>Large</td>
<td>Small</td>
<td>Small</td>
</tr>
<tr>
<td>Attributes</td>
<td>Size/Type</td>
<td>State Vector and labels</td>
<td>Allowed calls</td>
<td>Arbitrary</td>
</tr>
<tr>
<td><strong>AIM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fuzziness</td>
<td>Well-defined</td>
<td>Fuzzy</td>
<td>Well-defined</td>
<td>Fuzzy</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
<td>Global</td>
<td>Global</td>
<td>Global</td>
</tr>
<tr>
<td>Orientation</td>
<td>Structure vs data</td>
<td>Structure vs data</td>
<td>Structure vs data</td>
<td>Structure vs data</td>
</tr>
<tr>
<td>Usage pattern</td>
<td>Determined</td>
<td>Unknown</td>
<td>Determined</td>
<td>Unknown</td>
</tr>
<tr>
<td><strong>CONTEXT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mental maps</td>
<td>Non-existent</td>
<td>Non-existent</td>
<td>Pre-existing</td>
<td>Non-existent</td>
</tr>
<tr>
<td>Domain</td>
<td>Disk usage</td>
<td>Finite automata</td>
<td>Software</td>
<td>Variable</td>
</tr>
<tr>
<td>User Expertise</td>
<td>Novice</td>
<td>Expert</td>
<td>Intermediate</td>
<td>Intermediate</td>
</tr>
<tr>
<td>Hardware</td>
<td>Standard HW</td>
<td>Standard HW</td>
<td>limited RAM</td>
<td>Standard HW</td>
</tr>
<tr>
<td><strong>DESIGN ORIENTATION</strong></td>
<td>User aim</td>
<td>Data</td>
<td>User aim</td>
<td>Data</td>
</tr>
</tbody>
</table>

Chapter 6 describes a visualization of state transition graphs. State transition graphs are low-level descriptions of programs, which are usually generated from higher level program descriptions and can be used to study complex program behavior. Each program state is modelled as a node and an edge represents a transition to another state as a result of an (internal or external) action. State transition graphs can be extremely large (graphs of $10^6$ nodes are no exception). However, in practice the number of possible transitions from each state is bounded by the (limited) total number of possible actions, so state transition graphs are often sparse. Because they are generated from mathematical descriptions, they often exhibit a high degree of regularity.

Because of the complexity involved in creating and analyzing these input graphs, the visualization was targeted at domain experts. Since these experts could not formulate precise questions that they wanted answered by the visualization, we decided to use these data aspects (symmetry and regularity) as a start point for design. The large scale of these graphs makes it necessary to reduce the number of nodes present on screen while keeping the computational complexity of the algorithms involved low.

In Chapter 7 we present a case study on the visualization of a large software system. Given a software call graph and a hierarchy on the components in the call graph, the architects in the project wanted to know to what extent the call patterns in the implementation conformed to the architecture they had designed earlier. The main difference with the other cases is that here we already had a suitable hierarchy at our disposal, which absolved us from computing a good clustering of the graph. The main user aim was to determine whether an edge (call) was present between two nodes and see whether or not this edge was allowed according to the architecture. We later expanded on this initial prototype by adding an architecture that makes it possible to navigate these graphs even if they were semi-external, allowing application to graphs with millions of nodes.
Finally, in Chapter 8 we describe a visualization method for the visualization of small world graphs. Small world graphs are graph structures that have a very small diameter compared to their size, but exhibit significantly more local clustering than general graphs. Many existing systems can be shown to have this small world property, from social networks to software systems. We did not confine ourselves to a specific application domain in this case however, so node attributes may vary depending on the actual domain.

In this case the aim of the visualization was fuzzy, users were initially interested in obtaining an overview of the structure of this graph and the relation between structure and node attributes. Since the only structural difference between small world graphs and random graphs is the fact that the former have some local clustering we decided to leverage this clustering for the visualization design.

A critique of the approach mentioned above might be that it varies many properties per sample, making it impossible to relate changes in the visualization to a change in a particular property. In defense of this, here we are trying to extract global truths that hold for all graph visualization designs, irrespective of application domain. Fixing all properties but one would mean we are designing variations of a visualization for an application domain, with the associated risk that our conclusions are only valid for that specific domain.

Many concepts however, keep appearing in different domains. For example clustering is a recurring subject in large graph visualization, but in these cases we see it appearing as a means (using clustering to reduce the number of visible nodes so we can show symmetry more easily in state transition graphs), as a goal (construct the visualization such that the inherent clustering in small world graphs becomes visible) and as part of the input (given a specific clustering construct a visualization of a software system). In the next four chapters we describe the above-mentioned methods in more detail. In Chapter 9 we evaluate and compare the four visualizations and present a number of recommendations that one can adhere to when confronted with new cases where graph visualization has to be applied. In Chapter 10 we present a number of broader conclusions and give suggestions for further work.
Beamtree overview

Beamtrees are useful to visualize large hierarchical structures where the size of a node in the hierarchy is also important, such as a file system for example. The above image shows a rendering of the author’s entire drive consisting of 50,000 files. Each blue cylindrical beam represents a single directory and a ring on a beam represents a file in that directory. The type of a file is represented by the ring color. When viewed from the top, the visual size of both directories and files are proportional to the size of their respective sizes on disk. When viewed from the side the depth of a beam is proportional to the depth of that file in the hierarchy. Users are allowed to rotate and zoom into the structure. Beamtrees integrate both file size and file depth in a single visualization. From the visualization above we can easily identify (1) a single large file in the root directory, (2) a chain of directories containing only a single subdirectory and (3) a large directory with a lot of small .dll files.
Chapter 5

Beamtrees

In this chapter we confine ourselves to visualizations of trees and present Beamtrees. Beamtrees are based on treemaps, but use perceptual 3D cues to clarify the structure of the tree. We applied this visualization to file systems and we also performed a user study to compare them to other tree visualization methods.

5.1 Introduction

Treemaps [98] are compact displays, which are particularly effective to visualize the size of the leaves in a tree. However, one important drawback of treemaps is that the hierarchical structure is harder to discern than in conventional tree browsers: All space is used for the display of leaf nodes, and the structure is encoded implicitly. A number of attempts have been made to overcome this problem, the most notable being the use of nesting and shading. Inspired by treemaps, we present a new algorithm that visualizes the structure more explicitly. The user is enabled to adjust the visualization to his or her preferences: Either all space can be used for the leafnodes, which results in a conventional treemap, or more space can be used for the display of the hierarchical structure, which results in a beamtree. More specifically: where nested treemaps use nesting to indicate a parent-child relationship, beamtrees use overlap (Figure 5.1).

The hierarchical structure is visualized as a stack of rectangles and shading and 3D are used to strengthen the perceived depth in the structure. The beamtree algorithm is presented in section 5.2 and section 5.3 discusses the results of a user test comparing treemaps to beamtrees. Finally, section 5.4 summarizes the results and discusses future work.

5.2 Beamtrees

In traditional treemaps every node is represented by a (possibly obscured) rectangular area (Figure 5.2b). Our intention is to use the traditional subdivision algorithm but instead of
rendering only the rectangles representing leaves, we uniformly scale all rectangles such that the ones representing non leaf nodes also become visible. The overlap resulting from this scaling then serves as a visual cue for the depth of a node in the tree.

This scaling is done in one dimension and is alternated for each level in the tree. We shall refer to this dimension as the width in the remainder of this chapter. Figures 5.2b, 5.2c and 5.2d show that the more the width is scaled down, the stronger the depth effect of the partial occlusion. An obvious disadvantage of this approach is that some leaf nodes tend to become very thin, even stronger than with standard treemaps (Figure 5.2d). To alleviate this problem and to decrease the number of separate elements in the display, we visualize leaf nodes differently. Instead of scaling their widths, we use a treemap style subdivision algorithm: Each leaf node is assigned a section of its (scaled) parent node that is proportional to its associated size. For example, node \( n \) is assigned half of the area of its parent \( f \) in Figure 5.2e. Finally, to provide for a more pleasant display we sort the children of a node, such that all leaf nodes are aggregated at the top or left side of their parent rectangle (Figure 5.2f).

Since all rectangles are scaled with the same scale factor, their areas remain proportional to the associated size of the nodes they represent. The display in Figure 5.2f still suffers from problems however. Firstly, some nodes (for example nodes \( o \) and \( s \)) have touching edges, making it more difficult to perceive them as separate items. This can be resolved by also scaling nodes in a second dimension, using constraints to ensure the structure does not break up into separate pieces. We elaborate on this in section 5.2.1. Secondly, the large number of overlapping rectangles makes the resulting display abstract and hard to interpret. This can be significantly improved by the use of shading (section 5.2.3).

### 5.2.1 Constrained scaling

Since we scaled non-leaf rectangles in only one dimension, many of them still have touching edges (Figure 5.3a). As this makes it more difficult to perceive them as different visual entities we therefore also scale the other dimension (we shall refer to this dimension as the length) of each rectangle. We obviously run into problems when applying unconstrained
5.2 Beamtrees

Figure 5.2 Scaling a treemap to a beamtree. (a) Conventional tree representation: each node has an associated size. (b) Treemap representation: area is proportional to size. (c) Scaled widths to 50%. (d) Scaled widths to 25%. (e) Scaled leaves treemap style. (f) Aggregated leaves on one side.

scaling to internal nodes, since the structure might break up into separate pieces when doing so (Figure 5.3b). Another problem is that due to scaling, some leaf nodes may overlap, which is also not desirable (Figure 5.3b). We therefore need constraints on the boundaries of internal nodes. Note that we do not have to scale the lengths of leaf nodes in this case, since for these nodes a treemap style subdivision is used.

We calculate constraints for each rectangle bottom up in the tree, starting at the deepest internal nodes. The alternative, top down scaling, is not feasible because we need to know the extents of node $N$’s scaled child nodes before scaling $N$ itself.

When scaling we discern three cases for a horizontally subdivided internal node $N$, which is represented by a rectangle with top-left and bottom-right coordinates $(x_L, y_T)$ and $(x_R, y_B)$ respectively.

1. $N$ has only leaf nodes as children: In this case we do not need to compute constraints, since leaf nodes get scaled with their parent by definition (Figure 5.3c);  

2. $N$ has only non-leaf nodes as children: Let $x_1$ denote the position of the left edge of the first child of $N$ and let $x_2$ denote the position of the right edge of the last child of $N$ (Figure 5.3d). To avoid the structure from breaking up into separate pieces when scaling, we have to ensure that $x_L \leq x_1$ and $x_R \geq x_2$. Note that the exact scaled dimensions of both child nodes are already known at this point, since we calculate these bottom up in the tree;  

3. $N$ has both leaf and non-leaf nodes as children: We can use the right edge of the last child node of $N$ (i.e., $x_3$) as a bound in the same manner as we did in the previous
Figure 5.3 Scaling lengths. (a) Problems with touching edges; (b) Problems with break-up and overlap; (c) Only leaf nodes, no constraints necessary; (d) Node $N$ needs to be constrained between $x_1$ and $x_2$; (e) Node $N$ needs to be constrained between $x_1$ and $x_3$.

Because the length of some rectangles might be scaled less than we actually intended, due to the above constraints, the area of this rectangle is no longer proportional to the size attribute of the node it represents. We solve this problem by further scaling down the width for these rectangles, such that resulting area remains proportional to the node size. For a vertically subdivided internal node the procedure is analog to the one above. The entire algorithm can now be summarized as follows:

1. A standard treemap subdivision is created and the size of all rectangles is stored.
2. The length of each internal rectangle is scaled by a given factor $l$, observing constraints (if any).
3. The width of each rectangle is scaled by a given factor $w$. If constraints had to be applied when scaling the length, decrease $w$ such that $w \times l$ remains constant.
4. Finally, after all internal rectangles are scaled, all leaves are assigned to a section of their parent rectangle that is proportional to their associated size.

5.2.2 Algorithm

To show that the above-mentioned procedure can easily be integrated in the conventional treemap algorithm, we present the entire beamtree algorithm here in detail. We use the following data types:

\[
\text{Dir} = (X,Y);
\]
\[
\text{Bound} = (\text{Min}, \text{Max});
\]
\[
\text{Rectangle} = \text{array}[\text{Dir}, \text{Bound}] \text{ of real};
\]
\[
\text{Node} = \text{class}
\]
\[
\text{sizeN} : \text{real};
\]
\[
\text{sizeL} : \text{real};
\]
\[
\text{child} : \text{array}[1..nchild] \text{ of Node};
\]
\[
\text{nleaf} : 1..nchild;
\]
\[
\text{rt} : \text{Rectangle};
\]
\[
\text{rs} : \text{Rectangle};
\]
\[
\text{function rbound} : \text{Rectangle};
\]
\end

Node attributes \text{sizeN} and \text{sizeL} store \(S(N)\) and \(S(L)\) respectively. Array \text{child} contains \(nchild\) child nodes, sorted such that the first \(nleaf\) nodes in this array are leafnodes. The dimensions of the rectangle representing the node in a regular treemap are stored in \text{rt}, while \text{rs} stores the scaled version of the same rectangle. Finally, function \text{rbound} returns the bounding rectangle for a scaled treemap rectangle and its children. To facilitate notation we use three auxiliary functions: Function \text{Alt} alternates the direction of subdivision (i.e. \(\text{Alt}(X) = Y\) and \(\text{Alt}(Y) = X\)). Functions \text{SetSize} and \text{GetSize} respectively assign and return the length or width of a rectangle:

\[
\text{procedure SetSize}(\text{var r}: \text{Rectangle}; d: \text{Dir}; s: \text{real});
\]
\[
\text{var c: real;}
\]
\[
\text{begin}
\]
\[
c := (r[d, \text{Min}] + r[d, \text{Max}])/2;
\]
\[
r[d, \text{Min}] := c - s/2;
\]
\[
r[d, \text{Max}] := c + s/2
\]
\[
\text{end;}
\]
\[
\text{function GetSize}(r: \text{Rectangle}; d: \text{Dir}): \text{real};
\]
\[
\text{begin}
\]
\[
\text{GetSize} := r[d, \text{Max}] - r[d, \text{Min}]
\]
\[
\text{end;}
\]
The main algorithm is similar to the original treemap algorithm from [98] using two extra procedures to scale down the rectangles. Note that non-leaf nodes have to be scaled before leaf nodes, because for scaling the latter we need information on the (possibly constrained) scaled dimensions of their parent node.

**procedure** Node.BeamTreeLayout(r: Rectangle; d: Dir);

**var** i : integer; f: real;

**begin**

\( rt := r \);

if \( \text{sizeN} > 0 \) then

\( f := \text{GetSize}(rt, d) / \text{sizeN} \)

else

\( f := 0 \);  

for \( i := 1 \) to \( \text{nchild} \) do

**begin**

\( r[d, \text{Max}] := r[d, \text{Min}] + f \times \text{child[i].sizeN}; \)

\( \text{child[i].BeamTreeLayout}(r, \text{Alt}(d)); \)

\( r[d, \text{Min}] := r[d, \text{Max}] \)

**end**;

if \( \text{nchild} > 0 \) then \( \text{ResizeNode}(d) \);  

if \( \text{nleaf} > 0 \) then \( \text{ResizeLeavesOfNode}(d) \);

**end**;

Scaling of internal nodes is done in method **ResizeNode**. Global parameters **LengthScale** and **WidthScale** contain the scale factors for the length and width respectively. In practice, values of 0.95 for **LengthScale** and 0.35 for **WidthScale** give good results. The scale factor is applied to the original treemap rectangle \( rt \) in the third line. If we are dealing with a case 2 or case 3 internal node (see section 5.2.1), we have to apply constraints to the scaled rectangle. Finally, the difference between the unconstrained and constrained scaled lengths is taken into account when scaling the width, to maintain size proportionality.

**procedure** Node.ResizeNode(d: Dir);

**var** L, W, x1, x2, x3: real;

**begin**

\( rs := rt; \)

\( L := \text{LengthScale} \times \text{GetSize}(rt, d); \)

\( \text{SetSize}(rs, d, L); \)

if \( \text{nleaf} = 0 \) then /* Case 2 */

**begin**

\( rs[d, \text{Min}] := \text{min}(rs[d, \text{Min}], \text{child[1].rs}[d, \text{Min}]); \)

\( rs[d, \text{Max}] := \text{max}(rs[d, \text{Max}], \text{child[nchild].rs}[d, \text{Max}]) \)

**end**

else if \( nleaf < \text{nchild} \) then /* Case 3 */

**begin**

\( x2 := \text{child[nleaf + 1].rbound}[d, \text{Min}]; \)
In a last step we resize the leaves of a node in a treemap style fashion, using the scaled size of their parent as the initial rectangle.

```plaintext
procedure Node.ResizeLeafsOfNode(d: Dir);
var i : integer; f : real; r : Rectangle;
begin
    r := rs;
    if sizeN > 0 then f := GetSize(r, d) / sizeN else f := 0;
    for i := 1 to nleaf do
    begin
        r[d, Max] := r[d, Min] + f * child[i].sizeN;
        child[i].rs := r;
        r[d, Min] := r[d, Max]
    end
end;
```

```plaintext
x3 := max(rs[d, Max], child[nchild].rs[d, Max]);
x1 := x3 − (x3 − x2) / (1 − sizeL / sizeN);
rs[d, Min] := min(rs[d, Min], x1);
rs[d, Max] := x3
end;
```

W := WidthScale * GetSize(rt, Alt(d)) * L / GetSize(rs, d);
SetSize(rs, Alt(d), W);
end;
```
5.2.3 Beamtree Visualisation

Though occlusion is a strong depth cue, it also presents some new problems, the most notable being the fact that overlapping rectangles tend to break up into visually separate pieces. A solution to this problem is the use of shading [96] to indicate the direction of subdivision. As such, nodes are no longer visualized as two-dimensional rectangles but tend to resemble three-dimensional round beams, further strengthening the perceived depth in the picture. Additional depth cues can be provided by fogging and cast shadows (Fig. 5.4).

Figure 5.5a shows a file system rendered in 2D using the above-mentioned cues. Another strong depth cue is motion parallax. We implemented a prototype using OpenGL, in which the user can rotate the beamtree structure and return to the top-down view at the press of a button. An extra advantage of a three dimensional view is that the user can move his or her viewpoint to a point parallel to a beam direction and view the entire structure in a more conventional way with one display axis indicating node depth (Fig. 5.5c). As height for each beam we choose the minimum of width and length, as this provides the most aesthetically pleasing picture. The disadvantage is that, with height varying over each beam, beams at the same level in the tree do not necessarily get laid out at the same depth. We therefore provide an option to layer the stack of beams, that is, display each beam at a depth proportional to its depth in the tree (Fig. 5.5d). Lines indicate connections between beams when beams are displayed in layered mode. To make it easier for the user to maintain context, transitions to and from layered mode are animated. Instead of explicitly modeling every leaf node in 3D, we used procedurally generated one-dimensional texture maps to represent leaves. Depending on the number of leafnodes of the node under consideration, resolution of the texture can be increased. Using this approach we were able to render more than 25,000 nodes in real-time on a PC with a GeForce2 graphics card (Fig 5.5b).
Figure 5.5 Sample beamtrees: internal nodes/directories are blue, leaves/files are colored by type if available. (a) Filesystem rendered in 2D; (b) Same filesystem in 3D; (c) Layered orthogonal view; (d) Layered isometric view. See also color plate A.1.
5.3 User test

A small user study was conducted to test if a combination between explicit hierarchical information and treemap-style size information is an improvement over existing methods. Although a recent study [17] suggests treemaps are not the most ideal visualization tool for small hierarchies, we feel there are not many alternatives for larger (say over 200 nodes) hierarchical structures. We therefore conducted a user test similar to the one in [17], comparing both the 2D and 3D beamtrees to cushion treemaps as implemented in SequoiaView [163], and nested treemaps as implemented in HCIL’s Treemap 3.2 [93]. We hypothesized that beamtrees will be more effective in globally oriented, structure related tasks since they explicitly visualize structure. Their efficiency in size related tasks would probably be slightly less than regular treemaps though, because less display space is used to visualize leaf size. A total of twelve coworkers participated in the experiment, none of which were familiar with the concept of beamtrees, although seven were familiar with general treemap concepts.

5.3.1 Setup

The experiment was set up as a $4 \times 5$ randomized design, with 5 tasks for each of the 4 visualization methods (2D Beamtrees, 3D Beamtrees, Nested Treemaps and Cushion Treemaps). Following previous user tests with tree visualization methods [17], we wanted to test the ability of each visualization to communicate both the file system structure and the file sizes, since most tasks will focus on a combination of both. We also tested the ease with which users could recall the visualization, to assess their ability to create a mental map of the structure. Since the comprehension of structure is hard to test using a single task, we subdivided this part into three subtasks, focusing on parent child relationships and node depth (both global and local). We ended up with 5 general tasks that focused on:

- File size: Users had to select the three largest leaf nodes. Any node could be selected, and a response was judged correct only if all three largest leaves were selected.
- Tree topology: Users had to indicate the level of a predetermined node. To avoid potential confusion the level of the root node was defined as 1.
- Tree topology: Users had to indicate the total number of levels in the tree.
- Tree topology: Users had to indicate the deepest common ancestor of two predetermined nodes.
- Node memory: In order to test how fast users could achieve a mental map of the tree structure, they were asked to memorize the positions of two predetermined nodes. If the user believed he or she could remember the locations, the view was closed and reshowed at a different size, to prevent users from remembering the screen location instead of node location. Users then had to indicate the positions of both nodes.
5.3 Procedure

We created 4 smaller randomized trees consisting of approximately 200 nodes and 4 randomized larger ones of approximately 1000 nodes. Node sizes were distributed using a log-normal distribution. For each visualization method participants had to perform all 5 tasks for both a smaller and a larger tree, for a total of 40 tasks per participant. Trees and methods were matched up randomly and the order of visualization methods was also randomized. The order of tasks to be performed for each tree remained the same throughout the experiment. All visualizations used straightforward slice-and-dice partitioning and were displayed at a resolution of 1280 \times 1024 pixels. Nested treemaps used a 3-pixel border.

Beamtrees used scale factors of 0.35 for width and 0.95 for length. Node labels were not displayed, so users had to rely entirely on the visualization instead of textually encoded information. Detailed size information about each node was still available by means of a tooltip window that popped up when a user moved the mouse over a node.

If users indicated they were not familiar with treemaps they were given a short explanation on the general principle of treemaps, followed by a small test to see if they had understood the concept. The tests for each visualization method were preceded by an explanation on how to interpret the visualization method, how to interact with the visualization and what tasks had to be performed. Subsequently, one timed trial run was performed to make the participant more comfortable with the task. During this explanation and trial run the participant was free to ask questions. If the participant was convinced he or she understood the visualization method and the tasks to be performed, actual testing began.

Participants were not allowed to ask questions and no feedback on response time and accuracy was provided during the timed run. Timing was self-paced and started immediately after the user was asked a question. Timing ended when the user verbally indicated he or she had found the answer, or, in the case of the memory task, thought he or she could remember the location of the nodes. At the end of all tests users rated the methods based on their subjective preference.

5.3.3 Results

Since we were interested in comparing the overall time it took to complete a specific task on a visualization we summed the task times for the large and the small tree for each user. This leaves us with a total of 60 measurements (12 users and 5 visualizations). Since we are dealing with an intersubject design, we are not interested in variance between users but only in the relative differences in the speed of the tasks. We therefore normalized the average time for each user to the global mean time for all users and then performed a within-subject ANOVA analysis. The results of this analysis can be found in appendix B. Figure 5.7 shows the average response times (in seconds) on the left along with the $\frac{1}{\sqrt{2}}$ times the 95% confidence interval. If bars overlap vertically there is no significant difference at $p < 0.05$. On the right we show the average error rate per visualization method for each of the five tasks. We observed the following results for each task:
• **Size:** Determining the three largest nodes took more time using beamtrees, which was expected since they allocate less of the screen space to leaves, making small differences in size less obvious. The error rate for this task was generally (but not significantly) higher for beamtrees, supporting Cleveland [42], who stated that interpretation speed is negatively related to interpretation accuracy. These differences were not significant at \( p = 0.05 \).

• **Depth:** Interpreting the depth of a node did not show significant difference in response times. Error rates seem to indicate that nested treemaps and 3D beamtrees perform slightly better in this respect, since both of them dually encode depth information. Nested treemaps use both an alternating subdivision scheme and nesting, while 3D beamtrees make use of overlap and a third dimension to indicate depth. Cushion treemaps use both an alternating subdivision scheme and shading, but the latter seems to be too subtle to provide accurate depth information.

• **Levels:** For global hierarchical information, such as the total number of levels, 3D beamtrees perform significantly (\( p = 0.05 \)) better than both of the treemap methods in both time and accuracy. Most users simply rotated their view to the one depicted in figure 5c and counted the number of levels. 2D beamtrees performed somewhere in between, both in response time and accuracy. Most users got frustrated using both of the treemap methods to find the maximum depth and often just took a guess, which accounts for the reasonable response times but low accuracy.

• **Common parent:** Finding the common parent did not prove too difficult for three of the methods, but 3D beamtrees stand out with a significantly higher average. This is due to the fact that some users took the wrong approach and started rotating the structure, looking at it from all sides and trying to discern if and where beams were connected. In fact, using the default top view proves much simpler, considering the response times for 2D beamtrees. We expect a definite learning factor here, so response times should improve as users become more familiar with beamtrees.

• **Memory:** The results from the memory task were comparable, though there is a definite (but not statistically significant) difference between the accuracies for 2D and 3D beamtrees. This might be due to the difference between the flat shading and aliasing effects in the OpenGL rendering and the crisper 2D representation, but we are not sure.

Based on the results of this study we expect 3D beamtrees to perform significantly better than regular treemaps when it comes to global structural information such as maximum depth or balance. Not only does the 3D view provide stronger depth cues, but when viewed from the side it provides views that are similar to a conventional layered tree layout as well. Users also had a strong preference for 3D beamtrees, as is indicated by figure 5.6. This might be due to the fact that they were better able to answer depth related questions with the 3D version or the fact that 3D visualizations are generally found more interesting than 2D versions. Most users also felt they could perform better using beamtrees if they had more experience using the visualization.
5.4 Conclusions

We have presented an alternative method for the visualization of large hierarchical datasets. The method is based on the traditional treemap algorithm. By scaling down individual nodes in a treemap we introduce occlusion as an extra depth cue, while retaining the size proportionality of treemaps. This allows us to display both hierarchical and size information in a single display. Both 2D and 3D variants were implemented and tested, with the 3D variant achieving significantly better results in determining the number of levels in the tree. Compared to other tree visualization methods that display size and structure simultaneously, like icicle plots and tree rings [17], beamtrees offer higher scalability, up to thousands of nodes.

Although 3D visualizations generally introduce new problems such as occlusion and potentially difficult interaction, we do not expect these to be too much of a problem in this case. In the default top-down view none of the leafnodes are overlapping, while interaction can be facilitated by providing meaningful preset viewpoints (i.e. top down and side-views). Adding more sophisticated interaction methods like highlighting the path to the root will also improve insight. An advantage is that, contrary to regular treemaps, beamtrees actually display almost all internal nodes. One could argue that this is also the case with nested treemaps, but for interaction to be efficient here, nesting offsets would have to be around 5-10 pixels, taking up too much valuable screen space.

Although we have only experimented with inspection, there might be some interesting research opportunities in interaction with beamtrees. Beamtrees only work with slice and dice partitioning. Long thin rectangles already prevalent in this type of partitioning, suffer even more when scaled in their width. We partially avoided this problem by applying a treemap style subdivision to all leafnodes and simply discarding any remaining beams.
thinner than 2 pixels, but this is clearly not the most elegant solution.

It is an open question whether work that has been done in the area of aspect ratio improvement is applicable to beamtrees. In summary, we think the concept of beamtrees presents a valuable improvement over treemaps, displaying depth in a more natural way without compromising the biggest advantages of conventional treemaps, namely the linear correspondence between display area and node size.
Figure 5.7 Average response times with $\frac{1}{\sqrt{2}}$ 95% confidence intervals (left) and error rate (right) for each of the five tasks. Overlapping bars indicate an insignificant difference at $p = 0.05$, see also [118]. Error rates were analyzed using a $\chi^2$ measure ($p = 0.05$).
State Transition System overview

State transition graphs are low-level models of a process (or program). Computer scientists analyze these graphs to discover potential faults in a program in the design phase, faults that can otherwise only be discovered by systematic testing. Here we show a visualization of a communication protocol for a modular hydraulic lift system. The program is represented by a three dimensional tree that displays all possible execution paths a program can take. Execution starts at the top of the tree and progresses downward through one of the branches. Blue arrows represent return loops to states that might have been visited earlier. This method is scalable enough to deal with graphs of over a million nodes and allows us to visualize transition graphs that could not be seen previously due to their size.

From the visualization above we can see that (1) there are no loopbacks to the initial state, meaning that the top part of the program represents an initialization phase that only gets executed once; (2) there are small appendices branching off from this initialization phase, from which there is no possibility to progress to other states. This signifies a potentially problematic deadlock in the program; (3) there is a lot of symmetry in the program. Not only are the three largest branches symmetrical but inside these branches we can also find smaller symmetries. We may save time when analyzing this program by only considering one of the branches.
Chapter 6

State Transition Systems

In this chapter we present a visualization method for large finite automata. Insight in the structure of these automata can be very helpful in spotting design errors in computerized systems. We exploit the symmetrical structure of our input graphs by constructing a visualization that clearly represents these symmetries. Another strong point is the use of an effective and efficient $O(N + E)$ abstraction scheme, making application to very large samples feasible.

6.1 Introduction

The last decade has seen substantial progress in the analysis of computer-based systems. A central approach in this analysis is to describe the behavior by means of a transition system [14], consisting of states and possible transitions between these states. The most commonly used techniques based on explicit state enumeration can now deal with transition systems of billions of states. So-called symbolic techniques deal with even larger systems, employing a form of abstracted state spaces. These techniques have become sufficiently effective to assist in the design of systems, and are typically and increasingly often used for advanced bug detection. These advances in technology confront us with state transition graphs of enormous size. Very little insight in the structure of transition systems reflecting the behavior of computer-controlled systems exists, whereas such insight would be helpful to improve upon the reliable construction of such systems. There are currently three main approaches for the analysis of large transition systems.

The first classical approach is to derive a much smaller abstract variant of the transition system (typically less than 100 states) and perform analysis on that. A disadvantage here is that information about the actual state space is lost, though this information could be very useful for testing purposes.

A second approach consists of asking specific questions or checking for specific conditions. Although this works well for the detection of deadlock states, properties like symmetries within or similarities between processes can be nearly impossible for a com-
puter to find. Another disadvantage is that asking specific questions requires specific knowledge about the system, which may not be available.

As a third alternative method we opt to use interactive visualization methods, as the human perceptual system is much more adept at finding patterns in and similarities between structures.

Visualizations are straightforward when dealing with a small number of states, since a simple node and link diagram can be used. Current attempts at visualization mostly use off the shelf packages like Dot or Neato [67] and the number of tools targeted specifically at finite automata is limited [97, 22]. In practice however, the finite state machines under analysis often consist of tens of thousands of nodes or more, which severely limits the usefulness of these methods specifically and node-link diagrams in general (Figure 6.1).

Here we present a new method to visualize large state transaction systems. The transition systems are generated from a high-level formal description in $\mu$CRL [70, 33]. The user is presented with an overview of the structure of the graph, and can view individual nodes and edges on demand. Symmetries and regular structures in the graph are maintained and clearly visualized as such. Various properties of the state space parameters can be visualized interactively on top of this structure.

Section 6.2 discusses the general ideas behind our method. Section 6.3 describes the method used to visualize the state transition graph in more detail, including an efficient algorithm to cluster the nodes. In section 6.4 we present a variety of methods to obtain additional insight in the state transition system. Attributes of the states are visualized using color and shape; various options for interactive queries and selection are presented. In section 6.5 we discuss a number of real world cases, with up to a million nodes. Finally, in section 6.6 conclusions are drawn.
6.2 Approach

The general approach for 2D visualizations of directed graphs is to produce a pleasant looking picture by optimizing one or two aspects of the visualization. In the case of larger graphs however, the lack of visualization space in 2D quickly becomes a problem. A second problem is that optimizing only one aspect loses its effectiveness when dealing with a large number of nodes. Several attempts to overcome these problems have been made by using 3D visualizations and by applying techniques such as hyperbolic space [124], hierarchical node clustering [175, 51] or 3D spring embedders [88].

Since finding a general solution for visualizing large directed graphs can be difficult, most existing methods are forced to use domain-specific information. In this case we focus on large state transition graphs, which are automatically generated from mathematical high-level process descriptions and generally exhibit some form of symmetry and regularity. The visualization method presented here relies on three main principles:

1. Enable the user to identify symmetrical and similar substructures: By choosing to optimize only one local aspect of the graph, such as edge crossings, many structural symmetries or similarities are ignored. The same goes for extracting a minimal weight spanning tree. The visualization in figure 6.1, for example, obscures the fact that the marked groups of nodes have identical structural properties by positioning them in different ways. A clear picture of the similarities (and to a lesser extent the symmetries) in a graph enables users to mentally break up large structures into smaller similar looking pieces thus making a complex picture easier to digest. It also facilitates analysis and comparison of similar structures.

2. Provide the user with an overview of the global structure of the entire graph. A major problem in the visualization of state transition graphs is the sheer size of the graph. No matter the quality of the layout, it is simply impossible for the human perceptual system to form a schematic idea of what a graph looks like when confronted with thousands of information elements. We therefore employ a clustering based on the structure of the graph. Structure based clustering can provide a useful high-level map of the graph, in which the user can then select an area of interest to be inspected closer.

3. Provide sufficient interaction such that user questions about various aspects of the system can be answered. Since not all information present in the original system is taken into account when constructing the visualization, we have to provide the user with other mechanisms to obtain this information. He has to be enabled to examine the system from multiple perspectives, focus on parts of the visualization he is interested in and obtain detail information on demand. Interaction plays a crucial role in this.
6.3 Visualization of the graph

Assuming we have a Finite State Machine consisting of a finite set of states and an also finite set of possible transitions (along with action labels) between these states, we can define the corresponding state transition graph by the graph $G = (V, E)$, where $x \in V$ represents a state and $axy \in E$ represents a directed edge (or arc) between the nodes $x$ and $y$. A start node $s \in V$ represents the finite state machine’s initial state. We split the visualization process for a state transition graph into four distinct steps:

1. Assign a rank (or layer) to all nodes;
2. Cluster the graph based on a structural property, resulting in a backbone tree structure for the entire directed graph;
3. Visualize this structure using a method related to cone trees;
4. Position and visualize individual nodes and edges.

6.3.1 Ranking

In a first step, nodes are assigned to a layer or rank, comparable to the standard Sugiyama-type layout [159]. That is, all nodes are assigned a non-negative discrete value that indicates their depth relative to the start node. We found two ranking methods to be the most useful, since they correspond with two types of views on processes: iterative and cyclic (Figure 6.2).
In an iterative process view, the start node \( s \) is assigned rank 0, and subsequent nodes are assigned a rank that is equal to the length of the shortest path from the start node to that specific node, similar to a breadth first search (Figure 6.2). In state transition graphs the number of ranks will generally be small [132]. This ranking has the advantage that most arcs between ranks point in the same direction, which creates a natural ‘flow’ in the picture. Since most people tend to think of processes in terms of an iterative execution of statements, users will likely be familiar with this way of visualizing processes. The biggest disadvantage however, is that the few arcs that do point in the other direction are usually longer. These backpointers may not present a significant problem when dealing with relatively simple processes, but when dealing with complicated processes they tend to span several ranks and spoil the final visualization.

In a cyclic process view, we view a process as a cyclic execution of statements, with no clear beginning and end. If the start node is assigned a rank of 0, other nodes are assigned a rank that is equal to the length of the shortest path from that node to the start node, independent of the direction of the edges (Figure 6.2). This type of ranking by definition eliminates any long backpointers in the visualization, since each node is positioned at most one rank away from any connected node. This may be advantageous if users are looking for connected clusters of nodes. The major disadvantage is that arcs between ranks do not point in one direction, which makes it harder to comprehend the graphs layout in detail. Mainly for this reason we confine ourselves to iterative ranking in the remainder of this chapter. From here on we refer to arcs that point from a low ranked node to a higher ranked node as arcs pointing down, and vice versa. Furthermore, note that in both rankings horizontal arcs connecting nodes in equal ranks can also occur.

### 6.3.2 Clustering process

In the second step nodes are clustered to reduce the visual complexity of a graph, based on a local structural property that we define in this section. Instead of clustering by focusing on a single global property, such as ‘minimal edges between resulting clusters’ or a single node property we propose a clustering based on an equivalence relation between nodes. We aim at the creation of a tree structure based on the original graph, which we can use as a backbone to display the complete structure of the entire graph. To this end, we first modify the original simple directed graph and next simplify this modified graph by clustering the nodes. Given a digraph \( G \), consisting of a set of nodes \( V \) and a set of arcs \( E \), and given a ranking \( R : V \rightarrow \mathbb{N} \) that maps nodes to a non-negative discrete rank, we define a new set of arcs \( E' \) by removing any arcs spanning more than one rank from \( E \). Additionally, to facilitate definitions we reverse the direction of the remaining arcs that point upward. More formally:

\[
E' = \{ a_{xy} \mid x, y \in V \land a_{xy} \in E \land 0 \leq R(y) - R(x) \leq 1 \} \cap \\
\{ a_{xy} \mid x, y \in V \land a_{xy} \in E \land R(x) - R(y) = 1 \}
\]

No arcs in the digraph \( G' = (V, E') \) are pointing upward since for all \( a_{xy} \in E' \) \( 0 \leq R(y) - R(x) \leq 1 \) holds. Note that this does not necessarily mean that \( G' \) is acyclic since a
cycle consisting of nodes all having equal rank is still possible. Let \( D(x) \) be the set of all nodes that can be reached from \( x \) via zero or more arcs in \( E' \). We now define two nodes \( x \) and \( y \) to be equivalent iff a row \( (x = z_1, z_2, \ldots, z_N = y) \) of nodes with equal rank exists, such that

\[
\forall 0 \leq i < N : D(z_i) \cap D(z_{i+1}) \neq \emptyset
\]

In other words we consider two nodes to be equivalent if they share one or more common descendant nodes in the modified graph \( G' \). It can be shown [77] that the relation outlined above is an equivalence relation, so by definition its equivalence classes are non-empty and disjoint, which makes them very suitable to use as clusters. Since all nodes in a cluster have the same rank, we can extend the concept of rank to clusters. The rank of a cluster containing node \( x \) is then equal to the rank of \( x \).

We can now define a relationship between clusters that can be used to construct the backbone structure: A cluster \( C_1 \) is defined to be an ancestor of a cluster \( C_2 \) iff \( \text{Rank}(C_1) = \text{Rank}(C_2) - 1 \) and there exists an arc in \( E' \) connecting a node in \( C_1 \) with a node in \( C_2 \). A cluster \( C_2 \) is defined to be a descendant of \( C_1 \) iff \( C_1 \) is an ancestor of \( C_2 \). Since each node is in exactly one cluster, each cluster has at most one ancestor and cyclic ancestor relations are not possible, we can state that the resulting cluster structure is a tree (Figure 6.4).
6.3 Visualization of the graph

6.3.3 Clustering algorithm

In a usable application the implementation of this clustering process should be linear. Instead of using the naive approach of storing $D(x)$ for each node $x$ and checking for a non-empty intersection, we devised a recursive algorithm that clusters all nodes in linear time. In this paragraph we outline this algorithm for computing the backbone tree of a digraph $G = (V, E)$ where each node $n$ has been given a rank $R(n)$. For this we consider the graph $G'$, as defined before. Consider the nodes $v$ and $x$ with $a_{vx}$ an arc in $G'$. According to the definition of $G'$ there are two cases for $x$, either $R(x) = R(v)$ or $R(x) = R(v) + 1$.

In the first case node $x$ is equivalent with node $v$ since $D(x)$ and $D(v)$ have a non-empty intersection containing at least $x$. So $x$ and $v$ are in the same cluster. The second case is more complicated. If the ranks of $v$ and $x$ differ, they are not in the same cluster. However, arcs crossing ranks can induce that $v$ has to be merged with other nodes without a direct connection to $v$. We therefore first compute the cluster of $x$. For all nodes $y$ in the cluster of $x$ we can state that $D(x) \cap D(y)$ is not empty. Hence, for all these nodes $y$ we have to add all nodes $w$ with an arc $a_{wy} \in E'$ and a rank equal to $R(v)$ to the cluster of $v$, since $D(v) \cap D(w)$ is not empty (Figure 6.5).

These two cases form the heart of the recursive procedure $\text{ClusterTree}(v : \text{Node}, c : \text{ClusterNode})$ given below. The precondition of this procedure is that all nodes in $c.\text{Nodes}$ are in the same cluster as node $v$ and $v.\text{Cluster} = \text{nil}$, that is, $v$ has not yet been assigned to a cluster. A post condition for the procedure is that the subtree of the backbone tree with its root at the cluster of $v$ has been fully computed.
Figure 6.5  Cluster algorithm case analysis.

```plaintext
type ClusterNode = record
  Nodes : set of Node;
  Anc : ClusterNode;
  Desc : set of ClusterNode;
end;

procedure ClusterTree(v : Node, c : ClusterNode)
begin
  c.Nodes := c.Nodes ∪ {v};
  v.Cluster := c;
  forall a_vx in E' with x.Cluster=nil do
    if R(x)=R(v) then ClusterTree(x, c);
    if R(x)=R(v)+1 then
      ClusterTree(x, new ClusterNode);
    x.Cluster.Anc := c;
    c.Desc := c.Desc ∪ {x.Cluster};
  forall y in x.Cluster.Nodes do
    forall a_wy in E' with w.Cluster=nil do
      if R(w)=R(v) then ClusterTree(w, c);
  end;
ClusterTree(StartNode, new ClusterNode);
```
The cluster algorithm has a time complexity linear in both the number of edges and the number of vertices of $G'$ since each edge is examined at most once and each vertex is at most once a parameter for the procedure.

### 6.3.4 Visualizing the backbone tree

Before making a choice for a layout, we first state our requirements for a good layout:

- Symmetry is important and therefore a visualization that produces a more symmetrical picture is to be favored. Clusters and nodes with the same structural properties should be treated in the same way.

- There has to be a clear visual relationship between the backbone structure and the actual graph. It is easier for the user to maintain context when inspecting a small detail section if this detail looks approximately the same in close up view as it did in the global overview.

- The visual size of the clusters has to be related to the number of nodes in a cluster to prevent cluttering. Clusters with a larger number of nodes have to be visualized by larger visual elements.

Although classical 2D layouts are very predictable, familiar and easy to use, the lack of visualization space quickly becomes a problem when dealing with larger graphs, especially when considering that we want to visualize larger clusters as larger nodes in the tree. A possible technique to deal with this problem is to move from a 2D to a 3D layout, which gives us an extra dimension to increase the cluster size and allows us to easier display symmetries. We aim for a visualization that depicts clusters as circles in a horizontal plane. A plane is reserved for each rank, with the topmost plane containing clusters with rank 0. The backbone tree is laid out in a manner resembling cone trees [143], with ancestor clusters positioned at the apex of an imaginary cone and their descendant clusters placed at the base of the cone. In other words, our clusters have the same status as tree nodes in a cone tree. To emphasize the hierarchy in the cluster structure, truncated cones are drawn between related clusters. Figure 6.6 gives an impression. The overall process adheres to the basic concepts of cone trees but with a few alterations:

A. Clusters (the nodes in the cone tree) are visualized as circles of different sizes;

B. Symmetry is improved by also allowing clusters to be positioned in the center of the cone’s base;

C. The final resulting structure is given more ‘body’ and some extra visual cues are added.

**Ad A.** Normal cone trees consist of a collection of similar looking nodes. The tree nodes in our modified cone tree however, are the clusters we defined in the previous section. Since each cluster contains a different number of nodes we represent them by
different sized circles. Nodes will be placed on the circle boundary, so we choose to keep the circle’s circumference proportional to the number of nodes in the cluster, which results in the same amount of visualization space for each node.

**Ad B.** We present a heuristic for creating symmetrical layouts and discern the following cases for the positioning of the \( C \) descendant clusters of a cluster \( A \). If \( C = 1 \) the descendant cluster is positioned directly below \( A \). If \( C > 1 \) we space the clusters evenly over the base of a cone with its apex at the center of \( A \). The base diameter of this cone can be computed by using a recursive method similar to the one used in [39]. However, since positioning all \( C \) descendant clusters over the base may not always yield a symmetrical solution we make the following three exceptions:

- If there is a unique largest cluster among the descendant clusters, we position this cluster directly below \( A \) in the center of the cone’s base (Figure 6.7a).

- If there is one unique smallest cluster among the descendant clusters, we center this cluster when there are no largest clusters centered (Figure 6.7b) or when there is a largest cluster centered and the smallest cluster does not have any descendants. This prevents clusters from potentially overlapping each other.

- If after centering clusters based on the above exceptions, only one non-centered cluster remains, we choose not to center the largest cluster. This produces a more balanced layout (Figure 6.7b).
6.3 Visualization of the graph

Ad C. Since nodes are positioned on the circle boundaries, most edges between nodes in a cluster and nodes in a descendant cluster will typically run within a section of space bounded by a truncated cone. A simple but effective way to reduce the visual complexity of the graph then, is to visualize these two clusters as a truncated cone. The cone’s top radius is set equal to the radius of the ancestor cluster and the cone’s bottom radius is equal to the radius of the descendant cluster. If we are dealing with multiple descendant clusters, the cone’s bottom radius is equal to the radius of the base of the (imaginary) cone the clusters are positioned on. Although this method provides a good overview of the graph’s global structural properties it suffers from some problems inherent to 3D visualizations, the most notable being the problem of objects occluding each other. To overcome this problem and at the same time improve use of available visualization space we rotate non-centered clusters (and their descendants) slightly outward. Finally, transparency is added which further reduces this effect and provides extra visual clues when looking at still pictures. Some examples are shown in sections 6.4 and 6.5.

6.3.5 Positioning individual nodes

The previous two paragraphs presented a method to reduce visual detail by clustering nodes, providing a better global overview. The next step is to assign an optimal position to the individual nodes in the graph, given the fact that nodes are positioned on the circle edge. An optimal positioning of nodes satisfies the following requirements:

1. Short edges between nodes. A visualization is more effective if a node is kept close to its neighbors.
2. Maximum possible distance between nodes in the same cluster. Nodes are to be kept as far apart as possible to reduce cluttering and may not coincide.
3. Where possible emphasize symmetry in the structure by positioning nodes with the same properties in the same way.
Clearly the first two requirements contradict, since positioning two nodes in the same cluster that have the same parent node further apart leads to a greater total edge length. Another problem is the computational complexity. Although positions can be calculated by minimizing an error function or using a force directed approach, the number of nodes we are dealing with is generally too large to provide visualization at an interactive level. Another disadvantage is that two runs of the same optimization algorithm on virtually identical graphs may produce radically different layouts, which makes it impossible to identify similar substructures within a graph. We therefore select a rule-based approach, in which the position of a node is governed by local structural node properties. We use a two-step heuristic to position the nodes. First, we assign initial positions, based on the positions of nodes in ancestor clusters, similar to [170]. That is, nodes are positioned at the barycenter of their parents position. To enforce some regularity on the resulting layout, each cluster is subdivided into a number of slots after which nodes are rounded to the nearest slot. In a second step we adjust these positions to increase the inter-node distance, by dividing nodes sharing a slot symmetrically over a section of the cluster. The size of this section is governed by the occupancy of neighboring slots. A more detailed description of the layout method can be found in [77].

6.3.6 Visualization of edges

Given the positions of the nodes, the edges between them can be visualized. The standard way to show edges is simply to draw a straight line between two nodes. Edge direction is then usually depicted with a small arrowhead, or with transparency or edge thickness. We found that such subtle cues are not effective here, because of the large amount of edges. Also, the use of color is not the most intuitive, pre-attentive cue. How can we show direction more effectively? Inspired by figure 6.2a, we used the shape of an edge to indicate whether we are dealing with a downward or backward edge: Straight lines indicate downward edges, while curved lines denote upward edges (Figure 6.8). This cue is very effective in graphs using iterative ranking. It provides a more natural cue, emphasizes cycles in the graph and also prevents backward edges from being obscured, because they are now shown outside the cluster structure. The effect is substantially less effective in graphs using cyclic ranking though, in this case we will have to fall back on using color.

6.4 Interactive Visualization

In the previous section we described how the state transition graph can be visualized. However, this visualization is not the final answer yet. Firstly, the amount of detail can be overwhelming, hence the user has to be enabled to focus and zoom in on various aspects. This can be easily dealt with. The user can select a certain cluster, upon which only this cluster and its descendants are shown, simplifying its inspection. Secondly, only the structure of the state transition graph is displayed, and not the information associated with the states and transitions.
Typically, the relevant aspects of a system are modelled as a set of state variables. Each state is described by a particular valuation of these state variables; whereas a transition, occurring upon an event, leads to a change in the values of the state variables. Hence, the values of state parameters have to be shown to obtain more insight, especially in the relation of the structure of the graph with the semantics of the system studied. The same holds for the transitions in the graph. Each transition is labelled with an action label, that encodes the internal or external event that triggers this transition.

One could aim for a single image, showing all information simultaneously, but we think this is not possible. In a large system there can be tens to hundreds of state variables and their visualization for hundreds of thousands of states is a difficult multivariate visualization problem in itself, even if we would not aim at relating them to the structure of the graph. Instead, we opt for interaction here. The user has to be enabled to ask questions on the system studied and has to be provided with visual answers. Some examples of typical questions are:

- What is the value of a certain state variable?
- Which states are often visited?
- Given a start point, which states can be reached in a given number of steps?
- In which areas of the graph do transitions with action label 'A' occur?
- What is typical for a certain branch of the graph?

In this section we present various solutions. In section 6.4.1 we show how color can be used to visualize properties of clusters, in section 6.4.2 we show how local structure can be emphasized and in section 6.4.3 we present a correlation based method to obtain information on sets of nodes.
6.4.1 Cluster properties

Each node has associated attribute values. The display of these values per node is possible, for instance via color. However, this will often give too much detail and most of the time only a fraction of all nodes will be visible. A convenient option is to visualize properties per cluster of nodes. We have realized this as follows. First, for each cluster a certain property is calculated; secondly the value of this property is mapped to a value between 0 and 1; and finally, this value is used as an index in a predefined color table, which gives the color of the ring corresponding to a cluster. We implemented this using one-dimensional texture mapping, such that we could use color interpolation in HSV space. As a result, the tree structure is smoothly colored according to the clusters’ properties. Several properties of clusters can be visualized. A simple property is the distance of a cluster from the root node. Visualization of this attribute enhances the notion of flow through the graph (Figure 6.9a). It also helps in understanding the 3D structure of the tree: Overlapping transparent branches are now easier to distinguish. Another important question of users is where nodes with some specific (combination of) attribute values are located. The corresponding property per cluster is the fraction of all nodes within the cluster that satisfy the request of the user (Figure 6.9b). In our system the user can select the state variable from a list and adjust its query value with a slider, thereby enabling him to brush quickly over a large range of values. The same functionality is available for action labels, where the property per cluster is the fraction of in-edges for that cluster having a specific action label.

Stochastic information can be shown as well. Suppose that we start in the start node and make a random walk through the graph, what is the probability that our walk will end...
6.4 Interactive Visualization

**procedure** Walk (n : Node; p : real);
begin
if $p < \epsilon$ then
  $n$.Prob := $n$.Prob + $p$
else
  $q := \frac{p}{N}$;
  $n$.Prob := $n$.Prob + $q$;
  forall $m$ in next($n$) do
    Walk($m$, ($p - q$) / |next($n$)| )
end;
forall $n$ in $V$ do $n$.Prob := 0; Walk(StartNode, 1.0);

**Figure 6.10** Probability algorithm

at a certain node? The probability can be estimated using the algorithm in figure 6.10. This version is compact, a breadth first version however gives a better efficiency.

We assume that the length of the walk is exponentially distributed with average length $N$. Equivalently, at each step of our walk we have a chance of $1/N$ of ending the walk. At each step we accumulate the probability $q$ of ending the walk in the current state. Given that the probability of reaching a state equals $p$, $q$ is equal to $p/N$. Next, we make a step to all connected states, and repeat this recursively. The probability of reaching a connected state is equal to $(p - q)/M$, with $M$ the number of connected states. Distribution of probabilities ends when the probability $p$ of reaching a state falls below a certain threshold $\epsilon$. Normalizing the resulting probability distribution to [0..1] yields again a lookup color value, which can be used to color the cluster structure.

Figure 6.9c shows a sample probability distribution. Areas located immediately after backpointers light up because they are relatively highly travelled. This process also spends relatively much time in the small ’sink’ in the middle left of the picture. Several variations can be imagined. For instance, more realistic results can be obtained if for each type of transition the real-world probability is known. In that way less visited (i.e., exception handling) routines would show up less brightly colored. In a similar manner many more options for attribute visualization are possible, such as displaying the average fan out per cluster, which highlights the more complex sections of a graph; or the average change in state values, highlighting sections where the values of state variables change abruptly.

### 6.4.2 Node Details

Although the overview provides important structural information, in some cases it might be desirable to view the actual connections between nodes in detail. Execution might end up in a particular branch of the structure because of choices made much earlier in
Figure 6.11 Tracing node paths. An initial state at the top of the visualization was selected and the visualization shows all possible traces from that state. A small number of states (in red) were selected based on their state parameter. See also color plate A.4.
execution. When dealing with a visualization displaying a large number of nodes and edges at once, such causalities are impossible to trace. We solved this by allowing the user to select a node, after which the system also selects nodes that can be reached in maximally $N$ steps. This selection of the neighborhood of a node allows the user to immediately see the effects of choosing a particular path on the choices to be made later on. In the same way it is possible to select a node and have the system also select nodes from which the selected node can be reached in maximally $N$ steps. Since we wish to be able to use color to display additional node properties, we increase the size of a node to indicate it is selected. This has the pleasant side effect of visually emphasizing the node structure. To enhance this effect, selected edges (edges with both of their nodes selected) are drawn thicker, while the transparency of the backbone structure is increased. Figure 6.11 shows a part of a sample structure in which we selected one of the uppermost states. It can clearly be seen that execution continues into either the left or center branch. Subsequent analysis learns that depending on which initial state is selected, execution progresses into either one of the outside branches, while the center branch is accessible from all states.

### 6.4.3 Correlating attributes and locations

An observer will usually also want to know what the typical characteristics of particular clusters are. If he can identify a single common value of a state variable for the particular branches in the system, he can pin higher-level semantic concepts on different areas of the graph by using his knowledge on the state variables. We have integrated this into our application by allowing the user to select a substructure and let the system determine the correlation between a set of nodes with a common property and nodes in the selected region. The correlation between the two properties $x$ and $y$ over $N$ samples can be given by the Pearson correlation coefficient $r$:

$$r = \frac{N \sum xy - \sum x \sum y}{\sqrt{(N \sum x^2 - (\sum x)^2) \cdot (N \sum y^2 - (\sum y)^2)}}.$$  

Suppose we wish to know the correlation between the two boolean properties $x$ and $y$ defined as “a node has value $v$ for state parameter $p_i$” and “a node is an element of a selected set of nodes $S$” respectively. If we substitute a numeric value of 1 for true and 0 for false, we obtain a binary value pair for each of the $N$ nodes. The correlation coefficient can then be computed by substituting:

$$\sum x^2 = \sum x = |\{n \in V : n.p_i = v\}|$$

$$\sum y^2 = \sum y = |S|$$

$$\sum xy = |\{n \in V : n.p_i = v\} \cap S|$$

Computing $r$ for all possible combinations of $p_i$ and $v$ yields a list of correlation factors between $-1$ and $1$. A value close to 1 indicates that a property $p_i = v$ is typical for the selected region. A value of $-1$ indicates that none of the nodes in $S$ have the property $p_i = v$, while all other nodes do.
6.5 Cases

This section presents a number of real world cases in which we applied this visualization method to analyze the state space. All analyses were performed by experts on the visualized transition system, with assistance from us. Note that in all cases, experts initially did not have any concrete questions they wanted answered, so all analyses were exploratory in nature. In the following sections we give a description of the most important findings.

6.5.1 Modular Jack System

In this first case we analyzed a modular hydraulic jack system. If needed, extra hydraulic jacks can be added to an existing setup of similar jacks to increase maximum lift capacity. The whole setup of jacks can be operated from the controls of any individual jack, so obviously some kind of synchronization will be necessary. The corresponding synchronization protocol was analyzed by the CWI in Amsterdam [72]. Figure 6.13 shows the behavior of the protocol visualized using iterative ranking, with the number of jacks increasing from two in the first picture up to seven in the last picture. Although there are major differences in complexity between the individual protocols, the images are remarkably similar. Sometimes the number of branches in the backbone differs, but the overall structure of a branch remains the same. Figure 6.12 shows enlargements from the 2 and 3 jack systems. Notice how the global structure of the backbone tree remains constant while the detailed node structure varies.

Figure 6.12 Closeups: (a) section of the 2-jack protocol, (b) section of the 3-jack protocol and (c) close up of a deadlock trace in the 3-jack protocol. See also color plate A.3.
Figure 6.13 Behavior of modular jack system consisting of (a) 512 nodes, (b) 2,860 nodes, (c) 15,609 nodes, (d) 70,926 nodes, (e) 284,077 nodes and (f) 1,025,844 nodes. See also color plate A.5.
Figure 6.13 also shows the scalability of our visualization method: from about 500 nodes in the first picture up to over one million nodes in the last picture. Important features of all behaviors are the relatively thin strings of clusters that branch off at the top of the visualization. Figure 6.13a clearly shows that it is impossible to return to the start node after commencing execution, since there are no backpointers returning to the start node. This means that these clusters must belong to an initialization phase. Another striking feature is the perfect symmetry between each leg in the protocol. We used the correlation functionality described in the previous section to determine that the (internal) name of a jack is what separates states in one leg from states in the other. This also explains the symmetry since the behavior of the protocol should not be dependent on the jacks internal name.

The visualization for a system of three jacks (figure 6.13b) differs from the rest of the systems, in the sense that there are small sections splitting of from the initialization part. Figure 6.12c shows a close-up of this section. Upon further inspection, this section did not show backpointers emanating from the bottom most nodes, meaning that there are no more possible transitions when the system is in one of these states. These deadlocks were due to a problem in the physical relays in the jack system and only surfaced in the three jack configuration. Note that this deadlock can also easily be spotted by conventional analysis of the state transition graph (actually, in this case it has been, see [72]), but nevertheless a picture in which you can actually point out the deadlock is a great asset in communication.

6.5.2 IEEE 1394 - Fire Wire

A second case we applied this visualization method to was the Fire Wire protocol [94]. Fire Wire is a high-speed serial protocol, which is currently widely used in digital video processing. We confined ourselves to analyzing the link layer of the protocol, which provides an interface between the high-level transaction layer and the physical layer.

A formal description [115] of the protocol was used to simulate the behavior of two Fire Wire links connected via a serial bus. One feature we focused on are the two largely similar bulky areas in the center of the visualization (fig 6.14). As a start point for analysis we selected the bottom part and the system correlated this part with one state value that was unique for that part.

Looking at the original specification we determined that this variable represented a Boolean array of length two that kept track of which links have requested control over the bus during a so called fairness interval and that had a value of $[true, true]$ in the selected region (see also figure 6.15). During each fairness interval each link may request control over the bus at most once. A fairness interval ends when the bus has been idle for a specific amount of time, after which the system returns to its initial state.

The top part of the visualization describes the system behavior when either one of the two links gains access to the bus for communication. In the bottom part, the other link has requested use of the bus. Similarities are expected between the top bulk and bottom bulks, since the behavior of the system will not be radically different when the second node requests use of the bus. However, behavior should not be influenced by which node
Figure 6.14 Behavior of the link layer of the FireWire protocol, simulated using two links (25,898 nodes). The two large bulks in the center of the visualization indicate two (similar) communication phases. The asymmetries between the branches of the top bulk are unexpected, because they indicate that the behavior of the system differs, depending on which link requests use of the bus first. The dark line indicates a sample execution trace of the system, information like this is useful to ensure that tests cover as much of the state space as possible. Colors indicate cluster depth. See also color plate A.6.
requests the bus first, so the asymmetries between the branches of a single bulk are rather unexpected. Unfortunately, we were unable to verify whether these asymmetries originate from the original protocol design or from the formal protocol model.

6.5.3 The VLTS Benchmark set

The Very Large Transition System (VLTS) Benchmark set [28] is a recent set of currently forty real-world labeled transition systems, meant to serve as a test set for scientific tools and algorithms dealing with transition systems. Sizes of the transition systems in the set vary from a few hundred nodes to 33 million nodes. We applied our visualization algorithm to all transition systems in the set, up to the point where we ran out of physical memory. Figure 6.16 shows a subset of this collection. Visual representation of these large graphs allows for a number of hypotheses that can be made almost immediately. Firstly, it seems like \texttt{vasy\_25\_25} and \texttt{vasy\_40\_60} are both very regular graphs, although the fact that \texttt{vasy\_25\_25} is a path could also be deduced from both the number of nodes and edges and the fact that it is connected. Secondly, \texttt{cwi\_371\_641} bears a strong visual resemblance to the state space of the FireWire protocol in the previous section and is in fact generated from the same formal description [115]. Such strong global similarities between a graph of 25k and a graph of 371k nodes could not have been found with any conventional tool, or even by comparing these graphs computationally node by node. The same goes for the visualizations of \texttt{vasy\_83\_325} and \texttt{vasy\_166\_651}, that have very similar representations and also turned out to come from the same industrial application. Apart
Figure 6.16 A number of visualizations of transition systems taken from the VLTS benchmark set. See also color plate A.7.
from these observations, small observations such as the fact that vasy_8_38 is symmetrical and has a large number of deadlocks, considering the relatively small number of backpointers or the fact that cwi_214_648 has a strong branching structure can prove very valuable when trying to understand the global structure of a state space.

6.6 Conclusions

We have presented a new visualization technique to gain more understanding of large state spaces. Existing techniques can only provide legible diagrams for state spaces of at most a few thousand states. As far as we know there are no other techniques that are able to generate meaningful diagrams of larger state spaces. This makes comparison of this technique to other techniques in this domain impossible. In terms of characteristics, this technique offers a number of distinct advantages over traditional (static) graph drawing techniques:

• **High scalability:** by not displaying each individual state and transition we can effectively visualize a number of nodes that is at least two orders of magnitude larger than the best conventional techniques. On a medium desktop PC with a high-end graphics card we were able to display representations of graphs consisting of over a million nodes. At the same time, the user is still able to interactively zoom in on parts of interest and view individual transitions if needed.

• **Predictability:** In contrast to some other popular graph layout approaches (for example force directed methods or simulated annealing) this method is highly predictable. As a consequence, similar input graphs also lead to similar looking visualizations. This allows us to compare graphs that are generated from similar formal descriptions but have widely differing sizes.

• **Speed:** The performance of the method for large graphs is currently limited by available memory and the speed of the graphics card. The algorithm that generates the global layouts has a time complexity that is linear in both the number of states and the number of edges. See table 6.1 for details. The visualization itself is instantaneous.

• **Interactivity:** Since it is impossible to display all information related to a state transition graph in a single picture, interaction is critical. In our application the user is enabled to zoom into subsections of interest, color parts of the system based on state vector values or transition label values, query state variables, perform probability analysis and much more.

This does not mean however, that this technique is the ultimate answer to every question one might have for a specific state transition graph. As with any visualization technique it is simply meant as a support tool, to be used complementary to other techniques. Nevertheless, a tool that is capable of producing a meaningful image of very large state spaces greatly enhances general understanding of these state spaces.
### Table 6.1
Timing and memory results on a number of real world graphs, measured on a Pentium IV 2.4 GHz

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Edges</th>
<th>Time (sec)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>191</td>
<td>244</td>
<td>0.016</td>
<td>0.2</td>
</tr>
<tr>
<td>429</td>
<td>592</td>
<td>0.016</td>
<td>0.4</td>
</tr>
<tr>
<td>2,860</td>
<td>4,485</td>
<td>0.047</td>
<td>2</td>
</tr>
<tr>
<td>15,409</td>
<td>27,152</td>
<td>0.203</td>
<td>10</td>
</tr>
<tr>
<td>18,746</td>
<td>73,043</td>
<td>0.29</td>
<td>16</td>
</tr>
<tr>
<td>70,926</td>
<td>145,915</td>
<td>1.187</td>
<td>51</td>
</tr>
<tr>
<td>157,604</td>
<td>297,000</td>
<td>11.31</td>
<td>111</td>
</tr>
<tr>
<td>164,865</td>
<td>1,619,204</td>
<td>26.34</td>
<td>236</td>
</tr>
<tr>
<td>166,464</td>
<td>651,168</td>
<td>13.18</td>
<td>125</td>
</tr>
<tr>
<td>214,202</td>
<td>684,419</td>
<td>3.19</td>
<td>167</td>
</tr>
<tr>
<td>386,496</td>
<td>1,171,872</td>
<td>16.04</td>
<td>310</td>
</tr>
<tr>
<td>1,025,844</td>
<td>2,932,909</td>
<td>93.68</td>
<td>750</td>
</tr>
</tbody>
</table>

Figure 6.17 Situations in which the visualization does not perform well: (a) adding a new state to two previously unconnected states and (b) graphs with a small average path length.
Unfortunately the method also has its weak points (see also figure 6.17). Firstly, it does not deal well with large graphs that are highly connected, that is, graphs in which the average shortest path length between states is small. In this case nodes tend to clutter together in a very small number of clusters, which gives no information on the internal structure of the graph. Secondly, the method is not stable in the sense that in worst case scenarios, adding a single state to the graph might radically change the global layout if this state connects two previously unconnected sections. Thirdly, the layout algorithm used to position individual nodes in a cluster does not always produce an optimal layout. Finally, we found that researchers that we confronted with the visualization of a state space they had generated generally had a hard time interpreting the pictures, because in most cases they had no reference as to what the state space they were analyzing should look like. The option to mark sections of the visualization based on a specific state parameter or action label proved a great help in relating the original specification to the shape of the visualization.

Future work on this method should focus on integrating state reduction techniques such as abstraction and bisimulation [14] into the visualization. One can think of the possibility of interactively collapsing similar looking subsections or a visualization of a large state space morphing into a smaller version. Another option is to recursively apply this method to more complex sections of the graph, which may split large clusters into a number of smaller ones. This may also partly remedy the stability problem because it can split collapsed sections back into their separate components. Finally, the method currently has a rather large memory footprint and this is an aspect that deserves further scrutiny.

Summarizing, we have not seen any technique that is even remotely similar, and we sincerely believe that this visualization technique is a great step forward in understanding the global structure of state-transition diagrams. Actually, by looking at and interacting with the visualization tool we became aware of many properties of state spaces that, although sometimes obvious and sometimes more obscure, we had not realized until we saw them.
6.6 Conclusions
Matrixzoom overview
Matrices are an alternative way of visualizing graphs. Here we use a matrix representation of a graph to visualize the dependencies between components in a large software system. Each column and each row represents a separate component of the system. A component can consist of other (sub)components. A cell at position \((x, y)\) is colored if subcomponent \(x\) makes use of subcomponent \(y\). Green cells signify calls that were expected by the software architect, red cells indicate calls that were not expected.

The user is allowed to zoom into each of the cells in the matrix, to obtain a finer grained view of the call relations at a lower level of abstraction, until eventually the code level is reached. The system also supports plain geometrical zooming, expansion of individual columns and dynamic labeling of components.

The system architects can make a number of interesting observations from this image, for example: (1) Red cells in an otherwise empty area of the matrix usually indicate programming errors that cause unwanted dependencies in the system; (2) Long vertical column of colored cells signify interface components, components that handle most incoming calls from other parts of the architecture; (3) The bar diagram below the matrix indicates the size of each (sub)component, in this case we can immediately spot a component that is disproportionately large.
Chapter 7

Matrixzoom

In this chapter we depart from the traditional way of visualizing graphs and use adjacency matrices to represent a graph. We show that it is much easier to scale graph visualizations that use adjacency matrices, because we do not need to compute a 'good' node link layout. As an application area we chose a large software system in which the developer wanted to check the conformance of code with the architecture. Strong points of this visualization are its ability to deal with arbitrarily dense graphs and its scalability.

7.1 Introduction

Visualization can play an important role during the complete lifecycle of software systems, from the global design of a new system to the reverse engineering of legacy systems. Diagrams are a standard aid during the design of a system. In the maintenance of large legacy systems, for which no queryable design information exists or for which the design information is outdated, visualization can provide more insight into the structure of the system by aiding in the detection of patterns or features.

We focus on the use of visualization for the stage in between, where a high-level architectural design is transformed into an implementation. In large systems, the high-level architecture of a system is designed by a small team of system architects, and implemented by a substantially larger team of programmers. It is impossible and often not even desired for the architects to manage, or even specify, every little implementation detail, due to the size of the system. This often leads to subsystems being implemented with an interface that is too wide, subsystems growing too big or calls being made between subsystems that are not supposed to communicate, for example. Determining these potential deficiencies as early in the development process as possible can save much time later on.

A second problem when dealing with very large software systems is that even the top architects have difficulties to maintain a birds eye view of the entire architecture. Although they often have general knowledge of most subsystems, knowing exactly how all pieces fit together can be a daunting task. In this case software visualization can also
help, serving as a useful memory externalization and communication tool.

We use a very large software development project that was performed at Philips Medical Systems as a running example. The project dealt with the redesign of a medical imaging platform. To indicate the scale of the project: It is one of the largest software engineering efforts ever undertaken by Philips, consisting of well over 25,000 classes (including instantiated template classes), implemented in over 3 million lines of code, with a comparable amount of testing code. Approximately 300 programmers and 12 architects have been working on the project since 1997. Architects working on the system were interested in the following aspects:

- To what degree does the current implementation conform to the architecture specification? Are there any calls made that are not allowed?
- Which subsystems are affected if a subsystem is changed? Or, in other words, which subsystems call or are called by the changed subsystem?
- What are the changes in the system over time?

The next sections describe a visualization tool that is able to deal with the scale of the project and can provide answers to the questions stated above. Section 7.2 describes the problem and the approach we took, section 7.3 covers related work and section 7.4 deals with some visualization issues. We discuss an evaluation of the prototype in section 7.5. In section 7.6 we present improvements that were made in a follow-up project that allow application of this method to much larger samples. Finally, we conclude in section 7.7.

7.2 Approach

We model the system under consideration as a directed graph $G(V,E)$. The set of nodes $V$ represents the components in the system. Components form a rooted tree $T$, with component $s$ representing the entire system as root. An edge $e_{xy} \in E$ represents a call from component $x$ to a component $y$.

Every component $x$ has an integer abstraction level $d(x)$ equal to the depth of the component in the hierarchy. The maximum depth of the tree is equal to $D$. The system under consideration here has five different abstraction levels named System, Layer, Unit, Module and Class. Every component $x$ is contained in exactly one other parent component $P(x)$, unless $d(x) = 0$. The set of leaf nodes of a particular node $x$ is denoted by $\text{Leaves}(x)$.

The predicate $C(x,y)$ denotes the existence of a specific edge and is true if and only if a call $c_{xy}$ exists from component $x$ to component $y$. Note that the call graph is directed, so $C(x,y)$ does not imply $C(y,x)$.

Given a strict hierarchy on the components and a set of relations between the lowest level components we construct a data structure that has the following properties:

- **Strictly hierarchical.** Nodes form a tree structure in which $d(P(x)) = d(x) - 1$ for every node $x$ with $d(x) > 0$. 

7.3 Matrix visualizations

- **Layered calls.** Edges can only run between two nodes at the same depth in the hierarchy. Or equivalently: $C(x, y) \Rightarrow d(x) = d(y)$.

- **Call aggregation.** We aggregate edges upward through the hierarchy. That is, for every edge $e_{xy}$, we also store the edge $e_{P(x)P(y)}$. If this edge already exists we increase its multiplicity. Equivalently: $C(x, y) \Rightarrow C(P(x), P(y))$ and $\text{mult}(e_{xy}) = \#(e_{ij} \in E : i \in \text{leaves}(x) \land j \in \text{leaves}(y))$.

Figure 7.1 gives an example of a dataset that meets these properties, although the actual dataset has more levels in the hierarchy and a higher density.

The actual call data and composition hierarchy are extracted from the C++ code by a system developed at Philips Research Laboratories [134]. It consists of a number of PERL scripts and takes about 7 hours to extract data. Calls that are not interesting from a software engineering perspective, such as callback interfaces, Microsoft Foundation Classes calls or standard C++ function calls (such as `strcpy`), are filtered from the data.

Providing a meaningful, multi-level, stable and interactive layout is by no means an easy task. Traditionally, system architects are fond of node-oriented visualizations, such as the node-link diagram. In these visualizations, the node (or component in this case) and its characteristics are the prime objects of interest. Unfortunately, node-link diagrams usually cannot provide meaningful and readable layouts for more than a hundred nodes at a time if the graph is dense.

Node link diagrams are also not always as stable as one would like. Addition or removal of a single node can lead to a radically different layout. Although methods to alleviate this problem exist, a stable layout cannot be guaranteed. Because of these issues, node link diagrams are not very suitable for the purpose discussed here.

7.3 Matrix visualizations

We opt for a link-oriented visualization and the use of call matrices. Call and, more general, adjacency matrices have been used before in graph visualization as an alternative to the traditional node link diagrams in [8, 5, 23, 156, 186] amongst others. More specifically, inter-class call matrices have been proposed [131] to visualize calls between different classes of a relatively small object oriented system. Colors represented the frequency of calls between different classes. The user could also bring up a detailed view in a separate display, showing calls between different methods in a class. We expanded on this idea by using multiple levels in the hierarchy, allowing us to apply it to much larger samples. The different views have also been integrated by providing a smooth zoom transition between them, similar to the SHriMP [157] interface. This gives the impression of a single coherent information space.
7.4 Visualization

In this section we define the concept of a hierarchical call matrix on a hierarchy \( T \). Although we are dealing with a case in which calls only exists between components at the same level, we give a more general definition of a call matrix below. This allows us to reuse it in section 7.6, where we no longer require this.

Define two nodes in \( T \) to be incomparable in \( T \) if neither is an ancestor of the other. An antichain in \( T \) is an (ordered) set of nodes that are pairwise incomparable. One can think of an antichain as a (partial) horizontal cut through a hierarchy. The set of gray nodes in figure 7.1 are an example of an antichain.

A call matrix \( M(A, B) \) is then a \(|A| \times |B| \) matrix describing the connectivity between two antichains \( A \) and \( B \). If we denote the \( i^{th} \) element in an antichain \( C \) as \( C_i \) then cell \( M(A, B)_{ij} \) is equal to the \( 1 \times 1 \) matrix \( M(A_i, B_j) \) and contains the multiplicity of edge \( e_{A_iB_j} \).

We also define a function \( \text{zoom}(x): \mathbb{N} \rightarrow \mathbb{N} \) that computes a new antichain in the subtree rooted at node \( x \). For example, a valid choice for \( \text{zoom}(x) \) is \( \text{children}(x) \) or \( \text{leaves}(x) \). We use \( M_Z(x, y) \) as a shorthand notation for \( M(\text{zoom}(x), \text{zoom}(y)) \).

We can render a call matrix \( M(A, B) \) by drawing an \(|A| \times |B| \) grid and coloring a cell \((i, j)\) if the number of edges between \( A_i \) and \( B_j \) is greater than zero. From a visualization viewpoint, using a call matrix has the following advantages over node link diagrams:
• **Uniform visual representation**: The only visual element used in the visualization is a matrix-cell. This makes zooming in matrices easy to implement. Zooming into a node \( x \) in a node link diagram corresponds to displaying the matrix \( M_Z(x, x) \), zooming into edges from node \( x \) to node \( y \) corresponds to displaying the matrix \( M_Z(x, y) \).

• **Recursive structure**: We can nest the matrix \( M_Z(A_i, B_j) \) inside the cell \( M(A, B)_{ij} \). This means that visual representation of all edges between descendant nodes of \( A \) and \( B \) is by definition contained in the visual representation of the aggregated edge between \( A \) and \( B \). This makes it very easy to construct multiscale visualizations.

• **Stability and predictability**: Contrary to other popular graph visualization methods (such as force directed methods), addition of a call is guaranteed to never radically change the layout, since every call has its own prereserved section of visualization space.

The construction of a matrix visualization is not difficult as such, but a careful design is required for an optimal result. In this section we will elaborate on some of the subtleties in the construction of a hierarchical call matrix.

### 7.4.1 Matrix subdivision

In the above definitions we have not explicitly made a choice for the function \( zoom(x) \). One valid choice might be \( zoom(x) = leaves(x) \). Unfortunately, this means that if we want to give an overview of the system by rendering \( M_Z(s, s) \) we have to render a matrix of about 25,000 \( \times \) 25,000 items. Since this is well above the limit of modern day raster displays, we have to limit the number of rows and columns displayed. In this case the branching factor of \( T \) is in the order of 10, so choosing \( zoom(x) = grandchildren(x) \) leads to matrix with dimensions in the order of \( 10^2 \times 10^2 \), which can easily be shown on a standard display.

Each cell \( M(A, B)_{ij} \) in the matrix represents the calls between components \( A_i \) and \( B_j \). It seems natural to let the dimensions of cell \( M(A, B)_{ij} \) depend on the sizes of components \( A_i \) and \( B_j \), hence we first define the size of a component. The size of a component in a software architecture can for example be measured by the number of source lines or methods it contains.

A component’s size on screen can then be set to the sum of the sizes of its subcomponents. This leads to a visualization as displayed in figure 7.2a, which falls short in several respects. Large subcomponents take up an area of visualization space that is the square of their size suppressing small but possibly important components. The wide variety in row and column sizes (ranging from a few pixels to almost half of the visualization space) also makes the resulting matrix messy and hard to ‘read’.

Since we are displaying only a limited number of components on one axis at a time, a much better option is to use the number of currently visible subcomponents as a size measure instead. Figure 7.2b shows a visualization using this size measure, resulting in a much more regular, grid-like subdivision of the visualization space.
One of the disadvantages of using this view-dependent measure is that the size of a component depends on the currently visible matrix. This means that the size of a matrix cell changes when moving the view from one matrix to the next, since the number of visible subcomponents changes non-uniformly. This has as a consequence that matrix cells may look different depending on the abstraction level they are viewed on (figure 7.3).

A second disadvantage is that the displayed size of a component is no longer proportional to the size of that component relative to the system. This can clearly be seen in figure 7.2, where component E (actual, more descriptive component names are confidential information) takes up well over half of the system in figure 7.2a, and roughly one third of the system in figure 7.2b. Both these problems can be solved by applying appropriate visualization techniques, as we shall see in the following sections.

7.4.2 Multilevel zooming

A common problem in multilevel visualizations is that users are provided with different levels of visual abstraction, without immediately comprehending how these different levels interrelate. This problem is aggravated by the fact that the visual representation of a single matrix cell depends on the current level of data abstraction. If nothing were done to remedy this, users would quickly lose context when navigating the structure. This leads to information fragmentation instead of a consistent mental map. The use of a smooth, continuous semantic zoom from one abstraction level to another can be of great help in this case [156, 155, 25].
7.4 Visualization

(a) (b)

Figure 7.3 Identical matrix cells at different levels of visual abstraction. (a) shows the part of $M_z(P(x), P(y))$ representing $x$ and $y$ and (b) shows $M_z(x, y)$. See also color plate A.8.

We a user selects a matrix to zoom into, we therefore show a smooth transition from that matrix to its submatrix. The best results were obtained with a combination of linear interpolation and crossfade. Interpolation can be done in a straightforward fashion by animating every gridline in the high level representation to its position in a lower level representation. Figure 7.3 shows the positions of a single gridline (red) at two different levels of abstraction. The final effect is that grid cells expand or contract, depending on the number of subcomponents they contain. At the same time we perform a crossfade (that is, the transparency of one representation is increased, while the other representations transparency is decreased) to avoid visual elements suddenly appearing out of nowhere. Figure 7.4 shows a series of frames from a zoom operation, a full animation can be viewed at [75]. We can use the same procedure for panning operations.

Zoom/pan trajectories are computed using a novel method [178]. A smooth and efficient path of the virtual camera is derived analytically, such that the perceived velocity of the moving image is constant. Apart from semantic zooming, the visualization system also supports traditional geometric zooming into sections of a matrix cell, which is convenient since many parts of the matrix are very sparsely populated. Since it is possible that some matrix cells occupy less than a pixel on screen (depending on the geometric zoom of the user), any cell containing a call is rendered transparently at at least a preset minimum size. This avoids user having to hunt for cells containing calls, at the expense of possible overlap between calls. The use of transparency partly remedies this, though.
7.4.3 Multidimensional Information

Besides their suitability for multilevel visualizations, another advantage of matrix-type visualizations is that they offer space to display additional information. In a standard node-link view, edges are usually displayed as thin lines of varying length and orientation. This makes picking, coloring and labelling more difficult. In contrast, a matrix visualization has the advantage of a uniform edge representation and four matrix edges on which we can display node attributes. Two of the four edges are used to display node names, and the two edges remaining can be used to visualize node attributes such as subsystem size, number of changes or a number of system metrics such as (inverse) coupling, complexity etc. Figure 7.5a shows an example in which we used hierarchical histograms on the right and bottom matrix edges to visualize the size distribution within the largest layer of our system. One unit takes up over 95% of this subsystem (see also Fig 7.2a) with the major bulk of this unit being formed by only two modules.

Edge attributes can be visualized by means of color or transparency, other cues such as shading or texture could also be considered. We visualized a number of attributes in figure 7.5. Figure 7.5a uses color to indicate whether an implemented call was allowed according to the architects specifications. This information is of prime interest to the architect. Figure 7.5b uses a color scale to indicate the local neighborhood of a call. Calls that have a shorter path-distance to the call under consideration are indicated in red. Finally, in figure 7.5c transparency is used to indicate the multiplicity of a call. Other attributes that are more interesting from a software engineering perspective, such as the creation time of a call in the development process or the person responsible for that specific call, could also be shown, but unfortunately these were unavailable in the system under consideration.

7.5 Results

Although the use of a matrix representation for the display of graphs might seem awkward at first, one can easily identify general system characteristics with some experience. For example, library-type high level components that receive a lot of incoming calls, but only make a few outgoing calls themselves can be identified by scanning for vertical columns...
of calls. The same goes for controller components, which contain many outgoing calls but receive fairly little incoming calls. This type of behavior can be identified at different levels of abstraction: Layer $E$ (rightmost major column in figure 7.5b) is a typical library layer, and within this layer there are two units acting as an interface (large orange vertical columns in figure 7.5b). If we zoom in on calls to this specific layer we can identify a number of unit subcomponents acting as interfaces within these units. Figure 5c shows such a zoomed section.

One of the first practical results of the system was the observation that the (list oriented) information the architects previously used as a system summary was incomplete. A large number of low-level inter-class calls were not reflected in calls at higher abstraction levels due to errors in the data extraction process. Since in this visualization every call has its own place, which means that the lack of a call somewhere can also be spotted, a number of very sparsely populated submatrices immediately drew the attention. In terms of the initial aims of the visualization, we found the following:

**Spotting unwanted calls** When using the system, architects performed a quick scan of the system at the top abstraction level, and zoomed in on calls that they could not immediately understand (generally lone calls in an otherwise empty submatrix, such as the ones in the bottom left corner of figure 7.5b) or calls that were marked as ‘not allowed’. After one or two semantic zooms they could identify the function of a call by the component naming conventions they used and decided whether a call was wanted or unwanted. Calls that are initially considered as ‘not allowed’ by the system, can interactively be set to ‘allowed’. This leads to a steady reduction in the amount of unwanted connections between subsystems, as these are either removed from the code (in case of an implementation fault) or removed from the set of unwanted connections (in case the architects did not foresee this call).
Determining component dependencies  Spoting component interdependencies is easy if we take only direct dependencies into account. Architects can simply check a single row or column in the matrix to find all components calling or called by the current component. These components deserve special attention when changes are made to the component under consideration. Detecting indirect dependencies in a matrix visualization is much harder however, since calls that are structurally close, such as calls $C(a, b)$ and $C(b, c)$ for example, can visually be located far apart. This can be partially countered by indicating the neighborhood of a call upon selection or by displaying the $n$-step closure of relation $C(a, b)$. Both are not very satisfying solutions however. In fact, invalidation of the ‘semantic closeness equals visual closeness’ principle is probably the main argument against matrix type representations. However, also traditional graph layout methods do not always succeed in maintaining this principle, given the large number of long edges often present in layouts of complex graphs.

Stability  Matrix representations present a stable, regular and clean layout of complex interrelations. The addition of a small number of new calls or new components does not significantly change the resulting visualization. Architects can make snapshots of the system at regular intervals and easily track the evolution of their system. Stability also aids in the planning of a project: dependencies that are to be implemented in the future could be indicated in a different color. The same goes for sections of the system that have already been tested. The main advantage of a stable layout however, is the fact that architects are presented with a consistent visual image of their design. Mentally fragmented information is aggregated in a single image, aiding in the creation of a mental map of the software system as a whole.

System shortcomings  Initially users were enthusiastic about the system. They liked the ability to inspect an overview of the system as a whole and zoom down to the individual methods. However, they also noted a number of practical shortcomings in the system:

- It is not possible to modify the hierarchy on the fly, even though system architects would like this possibility. A good practical example is an architect that is trying to decide how moving a module to another component will affect the system.
- There is no direct link from the visualization to the code. This means that users that have located an illegal method call, first have to scroll to a (possible large) class file to change it.
- Many of the computer systems in use at Philips Medical Systems did not have hardware graphics support. This led to slow display in general and very choppy animations in particular.
Finally, we compare a matrix representation with traditional node link diagrams generated with GraphViz [67] in figure 7.6. Figure 7.6b shows a force directed layout of the subcomponents of the five top level layers. The layout was slightly modified to reflect the cluster structure. The large number of long edges between subsystems obscure a lot of interesting information that can be extracted using the matrix type visualization in figure 7.6a. For example the fact that layer E is called much more than it calls other layers is completely obscured in figure 7.6b. The same goes for the fact that D mainly calls E while calls to other layers are very rare. The large number of long edges also makes it hard to determine where an edge starts or ends. Although visualization techniques can be applied to partially solve these problems, one might also wonder to what extent node-link diagrams are part of the problem instead of the solution to visualizing large graphs. Extracting local structure from a matrix visualization requires much more effort however. Figure 7.6c shows a node link representation of the subsystem shown in figure 7.3. In general, for graphs that are small (say less than 100 nodes) and sparse, we think node link diagrams are superior to matrix representations.

7.6 Memory issues

In the previous sections we have outlined a visualization that allows us to interactively explore a large graph structure that has a given hierarchy $T$. One important restriction that we imposed on the data structure is that an edge can only run between two nodes with equal depth in $T$. In other words, we can only view relations between components that are at the same level in the hierarchy. This keeps the total number of edges stored in memory limited, as for each edge $e_{xy}$ in memory we also have to store an edge $e_{P(x)P(y)}$. We denote the number of edges induced by $Leaves(s)$ by $L$, that is, $L$ is the number of edges running between the leafnodes in $T$. If we assume that the average branching factor
of \( T \) is \( p \) and the number of edges is linear in the number of nodes (i.e., the graph is sparse), the total number of edges in memory is approximately constant if \( p \) is large:

\[
L + L \sum_{d=0}^{D} \frac{1}{p^d} \approx L \frac{p}{p - 1}.
\]

However, in many cases it can be extremely useful to also allow the user to view the relationships between components whose abstraction levels differ. For example, a user might be interested in all classes called by a specific unit. If we want the graph to remain fully internal, this means that for each edge \( e_{xy} \) stored in memory we also have to store two edges \( e_{P(x), y} \) and \( e_{x, P(y)} \) in memory. Figure 7.7b shows that for each horizontal edge \( e_{xy} \) at depth \( d > 0 \), we have to store \( 2(d - 1) \) additional non-horizontal edges connecting all the ancestors of the head to the tail and vice versa. Under the same assumptions we can compute the total number of edges from:

\[
(2D - 1)L + (2D - 3)\sum_{d=0}^{D-2} \frac{1}{p^d} + L \sum_{d=0}^{D} \frac{2(D - d) - 1}{p^d} \approx 2LD \frac{p}{p - 1}.
\]

In this case the number of edges in memory is linear in the depth, which might not seem too bad. However, for very large graphs and deep hierarchies, storing \( 2DL \) edges can be very costly. For example, a 3 million edge graph with a hierarchy depth of 5, and a (very minimal) memory footprint of 128 bytes per edge already consumes 2GB of memory for the edges alone. In even worse cases, it could be that even the \( L \) edges between leaf nodes in the hierarchy do not fit in memory, in which case we have to deal with a semi-external graph.

### 7.6.1 Architecture

To solve both these problems a scheme has been devised in which we do not store all aggregated edges in memory explicitly, but instead compute them on the fly from a memory resident subset. For that we borrow the concept of a **cover** in the component hierarchy tree \( T \) from [1]:

A **maximal antichain or cover** of \( T \) is an antichain in \( T \) to which no new nodes can be added. The singleton set containing the tree root and the set of all of the leaves are simple examples of a cover of \( T \). A cover \( C \) of \( T \) together with the set of aggregated edges induced by the nodes in this cover are called a **C-view**.

We also assume that we have random access to the degree of each leaf node in \( T \), since even in the semi-external case this can be pre-computed by doing one pass over \( E \). As an upper bound for the degree of a non-leaf node we sum the degrees of its children. Note that we cannot compute an exact value in these cases, since that would involve explicit aggregation of all edges.

If \( E \) does not fit in RAM we need to compute a view whose at most \( M \) number of edges will fit in memory. For this we use a greedy procedure (See figure 7.8) that, given
Figure 7.7 Storing edges in memory: (a) when allowing only matrices \([X,Y]\) in which \(A(X) = A(Y)\); (b) when allowing all possible matrices.

a bound \(B\) on the number of nodes, starts from the rootnode and iteratively expands (i.e., replaces a node with its children) the node with the highest degree. Expansion is halted when the current cover cannot be expanded without exceeding \(B\). Finally, the edges in the \(C_M\) view are computed and subsequently stored in RAM. The rationale behind expanding the largest degree nodes is that we want to avoid expanding sparse areas of the graph. Note that we have to use the very conservative \(B = \sqrt{M}\) as an input bound on the number of nodes in the cover \(C_M\), since we don’t know the exact number of edges in the \(C_M\)-View beforehand.

Instead of having the user commence navigation at the rootnode and require him to expand a large number of nodes manually during navigation, we can use the same procedure to compute a screen-cover \(C_S = Cover(S,T,\text{Root})\), where \(S\) is equal to the maximum number of nodes we can show on screen at once. For matrix oriented visualizations this is typically around 200 nodes. Figure 7.9 shows a schematic representation of both antichains, with \(C_S\) above \(C_M\) since typically \(S << M\).

We can implement interactive navigation by initially showing the user a \(C_S\)-view. The system fetches the multiplicity for an arbitrary edge \(e_{xy}\) from disk or from memory depending on the position of \(x\) and \(y\):

- If \(x\) and \(y\) are both on or above (that is, are ancestors of a node in) \(C_M\) we first determine the descendants of \(x\) and \(y\) on \(C_M\), get the edge set induced by them from RAM and aggregate these edges upward to \(x\) and \(y\).

- If either \(x\) or \(y\) is below \(C_M\) we have to fetch the edge set induced by their leaf descendants from disk and aggregate it upwards. To do this quickly we use a disk index that is based on a kd-tree as is detailed in [7].
function Cover(Size $B$, Node $N$

input: An upper bound $B$, start node $N$.

output: A greedy cover $C$ of the subtree rooted at $N$ with
no more than $B$ elements such that max(degree($C$)) is minimum.

$\{
C := \{N\};$
Finished = false;
while not Finished do 
  Finished := true;
  for each $p$ in $C$ (highest degrees first) do
    if (|children($p$)| + |$C$| - 1 $\leq B$) then 
      $C := C \cup$ children($p$) - {$p$}; // expand $p$
      Finished := false 
  }

Figure 7.8 Pseudo code for the greedy cover computation. This assumes a
start node $N$ and a bound $B$ on the maximal number of nodes in the
cover.

7.6.2 Visualization modifications

The mechanism described above was implemented on top of the prototype visualization
as presented in the previous sections. We made a number of changes to the navigation
and display capabilities (see also figure 7.10).

- **Hierarchy**: Instead of defining $zoom(x) = grandchildren(x)$ we set $zoom(x) =
children(x)$. The hierarchies we were dealing with had an average branching in the
order of $10^2$ or sometimes more. Intermediate levels were introduced for nodes that
had more than 300 children, according to the procedure specified in [3].

- **Expansion and hiding**: Clicking the $i^{th}$ row header or $j^{th}$ column header replaces
the node in that particular row or column with its children. More formally, it pro-
duces the matrices $M(A \cup children(A) - \{A_i\}, B)$ and $M(A, B \cup children(B) - \{B_j\})$, respectively. The user can also choose to hide these rows or columns resulting in
matrices $M(A - \{A_i\}, B)$ and $M(A, B - \{B_j\})$.

- **Zooming**: Clicking a subcell $(i, j)$ in a matrix $M(A, B)$ produced a view of ma-
trix $M_{i,j}(A_i, B_j)$. To avoid the user having to manually expand multiple possibly
sparse matrices we also included the ability to have the system select an optimal
expanded view by using the Cover procedure. In this case the resulting view is
$M(Cover(S, A_i), Cover(S, B_j))$.

- **Context**: Since adding the ability to view all possible matrices $M(x, y)$ makes it
easy for a user to lose context, we added iconic representations of the hierarchy tree
7.6 Memory issues

indicating the position of all nodes in $x$ and $y$ in $T$ to the top and left sides of the matrix.

- **Aspect**: The aspect ratio of the rendered matrix was matched to the aspect ratio of the screen by scaling it. This allows better use of screen space, and prevents interaction problems in matrices with very unfavorable aspect ratios.

- **Information**: Since in our application the property of interest was graph density we used the color scale to indicate the density of the subgraph induced by the leaf nodes of $x$ and $y$, instead of the conformance of calls between $x$ and $y$ to the architecture.

- **Representation**: The user is allowed to switch to a node link view of the currently visible matrix if it is small and sparse enough.

7.6.3 Application

We applied this massive graph prototype to a number of large semi-external data sets. The first one was taken from the SEER [142] cancer database. SEER stands for the Surveillance, Epidemiology and End Results program of the US National Cancer Institute. It is an authoritative source of data on cancer incidence and survival. Each record consists of 72 items that provide specific cancer diagnosis and treatment information, patient demographics and geographical and temporal data. SEER data is widely utilized to identify
Figure 7.10 A screenshot of the improved interface, showing racial distribution for colon cancer cases in Connecticut counties. The overview window is in the top left, hierarchy trees are on the top and left of the matrix, node labels are on the bottom and right. See also color plate A.12.
geographic and population differences in cancer patterns, to investigate environmental factors that influence cancer incidence and survival, and to study cancer treatment outcomes. Standard epidemiological techniques include regression methods and comparison of a variety of rates and ratios.

We can view this dataset as a bipartite graph from the set of patients to the set of attribute values. Of particular interest in this representation are large bi-cliques, dense clusters containing a group of patients and a group of properties. These bi-cliques can signify a correlation between a particular group of patients and a set of properties, which might be useful information to epidemiologists (See figure 7.11).

On the set of patients we defined a geographical hierarchy with 20 subtrees corresponding to the geographical 20 SEER registries (visible on the left hand side of the canvas, see figure 7.10). Following epidemiologists advise, we extracted 17 attributes and built a hierarchy for them (visible at the top of figure 7.10). This hierarchy consists of separate subtrees for cancer specific information such as primary cancer site and tumor size; treatment type (surgery, amount of radiation); temporal information (diagnosis date, age, etc) and patient demographics. In this case we were interested in geographical differences between cancer occurrences, if we had been interested in age differences we would have defined an age hierarchy on the patient set.

Here we used subsets of the entire SEER dataset, detailing colon, prostate and skin cancer, containing 437,738, 358,783, 142,406 nodes and 5,182,643, 5,526,803, 1,866,531 edges respectively. The prototype was running on a machine with 750MB of total RAM. Among the interesting findings we mention the following:

Figure 7.11 A small biclique in a group of 9 patients. This may lead to the conclusion that the skin cancer incidence rate for people between 30 and 40 in San Francisco is higher.
The skin cancer age distribution in San Francisco county in the San Francisco-Oakland registry shows a peak (darker red area) at around 35-36 years of age. This peak is not present in other counties in the same registry. Note that the cause of this might be anything from more sun(bathing), higher skin cancer awareness in San Francisco county or simply a younger overall population, so it’s very hard to draw immediate conclusions from this observation. Nevertheless, it is interesting enough to investigate further. (See figure 7.12a);

Skin cancer race distribution in Iowa is severely tilted to whites. This might again be a consequence of the racial distribution in Iowa (likely) or it might be that whites are more susceptible to skin cancer (also likely). To answer these questions the data must be compared to the population distribution. (See figure 7.12b).

We want to point out that our knowledge of epidemiology is very cursory and the intention is to show that with some guidance from domain experts one can enhance the interface to ‘automatically’ find patterns like the ones described above. A pundit could say that all of this could have been done by just using a data base query system. The point is well taken if the user knows precisely what he is looking for. However, we work under the premise that very often this is not the case. What is offered here is a tool that can help domain experts quickly formulate a set of hypotheses for further analysis. It is a visual and intuitive exploratory data analysis tool that may help the user get a quick glimpse over the entire data space in a hierarchical manner.

Recent work by Abello [6] reports on the use of an memory external algorithm that produces a hierarchy for a phone traffic graph with over 260 million vertices (US telephone numbers) and over 4.3 billion edges (phone calls for a period of over 20 days). The resulting hierarchy is based on the recursion pattern of an fully external divide-and-conquer procedure that computes a minimal spanning forest for a semi-external graph.
We wondered if we could use our interface to navigate a restricted aggregated-view of this 260M node spanning tree (since we had no access to the corresponding lower level data due to privacy constraints) containing 200K nodes. Contrary to the SEER data set both axes of the matrix display the same hierarchy here. The top level matrix represents an aggregated view of the entire spanning tree, where each node in this aggregation is a contraction of a part of the minimal spanning tree by itself (See figure 7.12c). In other words, the depicted hierarchy trees together with the matrix show an anatomical description of a giant semi-external tree which the original author had not seen before, even though he had previously worked with the corresponding data set.

7.7 Conclusions

Since displaying large graphs as visual networks often brings more problems than it solves, we have advocated the use of matrix oriented graph representations. Adjacency matrices have been used to display dynamic properties of object oriented programs on a smaller scale [131] and we found that they exhibit a number of properties that makes them attractive to use in the context of general graph visualization. The fact that adjacency matrices use a uniform representation for edges between two clusters of nodes and edges within a single node cluster avoids problems with zooming into single edges. Their recursive structure makes it simple to employ smooth multiscale zooming. We improved on existing methods by providing a smooth semantic zoom between abstraction levels, which makes it easier for users to maintain context. Their visual stability provides a consistent birds-eye view of the entire graph. Matrix representations can also serve as a useful externalization of possibly fragmented knowledge of the user. Finally, the fact that the graphical representation of a matrix can be computed in linear time, makes them also very scalable. We have shown this by applying a matrix oriented visualization to the navigation of graphs with millions of edges.

An important topic we have not touched on is the ordering of rows and column within a matrix cell. Ideally, one would like to see the pattern of relations between subcomponents reflected in the row and column ordering. As an example, given subcomponents x, y and z with C(x,y) and C(y,z), the row column ordering (x, y, z) is clearly preferable over (z, x, y). Finding an optimal ordering is NP-complete [111], and finding a ‘good’ row and column ordering for an arbitrary subgraph is a special case of finding a good clustering. A number of polynomial time heuristics exist for computing good orderings [40, 117] but we managed to avoid the problem here by using the data attributes to obtain a suitable ordering.

In cases where we are dealing with large graphs containing possibly dense structures (making the use of node link diagrams difficult) matrix based visualizations can offer a valuable alternative to node link based visualizations. When supplied with a good hierarchy on the node set, the implementations above can be used to visualize arbitrary semi-external graphs, which in practice means up to $10^7$ nodes. If a good hierarchy is not available we have to construct one ourselves by clustering the graph, which can be quite a challenge on semi-external graphs.
Small world graphs overview

Small world graphs are graphs in which the path length between any two nodes in the graph is relatively small, making visualization of these graphs more difficult. Here, we used a sample knowledge base consisting of 500 painters and sculptors as an example. Each artist is cross referenced to a number of other artists, based on diverse criteria such as subject style, era or common friends.

Each sphere in the picture represents either a cluster of artists or and individual artist. Color represents artist birthyear with red colors being more recent. Groups are formed based on the internal clustering structure of the graph, so a sphere usually indicates a coherent group of artists, and in many cases we can also pin an artistic movement on a sphere. The user can browse both the global and detail structure by means of a moveable lens that expands clusters near the lens and collapses clusters farther away.

Even with limited knowledge on art history we can make a number of observations from this rendering: (1) There is an artist with cross references to both ancient and modern artists (Wood); (2) We can identify 2 artists that are positioned close to modern artists but have very different birthyears (Bosch and Arcimboldo); (3) We can pick out clearly defined clusters of artists such as the one in the bottom right, corresponding to the constructivism movement.
Chapter 8

Small World Graphs

In this chapter we construct a visualization for a special class of graphs, called small world graphs which are difficult to visualize because of their highly connected nature. We show that we can exploit the inherent clustering structure of these graphs by constructing a layout that reflects these clusters. We present a novel focus and context scheme that lets the user browse one area of interest, while the surrounding nodes are represented at a higher abstraction level. Strong points of this visualization are its ability to generate layouts in which the node density represents cluster structure and an intuitive focus and context method.

8.1 Introduction

First identified by Milgram [121] in social networks, small world graphs are graphs which have a small average path length (average shortest path between nodes) compared to their number of nodes, but have a high degree of clustering compared to a random graph of the same size. The degree of clustering can be expressed by the clustering index [177, 15]. The small world property has been identified in many real world graphs such as social networks, biological neural networks, software systems, power grids, cross referenced knowledge bases and the internet. Effective visualization of these graphs can provide valuable insight in their structure.

As a running example throughout this chapter we use a graph of 500 cross-referenced painters and sculptors, taken from The Art Book [133]. References between artists were made based on time periods and artistic movements, but also on the style and feeling of their work. The resulting connected graph consists of 500 nodes and 2486 edges. The graph has an average path length of 4 and a clustering index of 0.18. A random graph of similar size has a clustering index of 0.0093. We therefore consider it to have small world characteristics. Note that small world characteristics does not imply that the graph is dense, a sparse graph can also be small world.

Here we choose for a node-link representation instead of a matrix layout. The only
class of node link layout algorithms that are general enough to deal with small world graphs are force directed algorithms, which model a graph as a system of repelling particles and springs. The situation in which the energy in such a system is minimal constitutes an optimal drawing. Force directed algorithms can provide satisfactory results for many types of graphs. However, they often produce disappointing results when it comes to dense and/or highly connected graphs in general and small world graphs in particular. Figure 8.1 shows a layout for our artists graph, generated with GEM [61], a typical example of a force directed graph drawing package. Based on the semantics of the input data and the relatively high clustering index, we would like the visualization to clearly display any strongly connected subgraphs that might correspond to artistic movements. Yet we observe that the nodes are distributed rather uniformly over the image, and that the large number of overlapping edges makes it hard to draw conclusions on the global structure.

In this chapter we present a new method to visualize small world graphs, such that insight in the structure can be obtained more easily. Section 8.2 presents our approach. In section 8.3 we describe how the layout can be improved by employing a force model based on recent work by Noack [127]. In section 8.4 we show how we can reduce the number of screen items and still provide a detailed impression of the area we are interested in. Finally, in section 8.5 we draw conclusions and discuss future work.

8.2 Approach

One way to show the structure in graphs is to determine graph clusters (loosely defined as highly connected subgraphs) a priori and present these to the user. However, as we explained in section 3.2 a hard problem with the use of any automatic clustering method
is the fuzzyness of the problem itself. Although the concept of a graph cluster is easy to
understand intuitively, there is no full-proof formal definition of the concept of a natural
graph cluster for every task the user wants to perform. A second problem is that hav-
ing a decent clustering does not absolve us from constructing a consistent visualization
of this clustered graph afterwards. We decided to take an alternative approach by first
constructing a layout that reflects the cluster structure and derive the clustering from this
layout.

Force Directed Algorithms (see section 3.1.4) are a popular layout method, but applying
them to highly connected graphs, results in cluttered pictures as illustrated by figure
8.1. A recent promising direction is the introduction of a force model that maps the con-
nectivity between two sets of nodes to geometric distance between those sets [127]. This
results in highly connected groups of nodes being displayed as areas with a higher node
density.

Recently, Auber at al.[15] visualized small world graphs by computing an edge strength
metric for each edge in the graph. The graph is then partitioned into subgraphs by remov-
ing all edges that have an edge strength below a given threshold. Although initial results
are promising, this method was not very effective on our sample graph. We believe this
is due to the rather small clustering index of this graph (0.18), whereas Auber et al. state
that their approach works best on graphs with higher clustering indices.

Other methods to obtain insight into large cluttered structures are Information Galax-
ies, Information Landscapes [182] and the related Graph Splatting method [114]. Galax-
ies and Information Landscapes (see also section 3.3.1) are mostly used to visualize large
document collections. Documents are represented as multidimensional points, clustered
and subsequently projected onto a two dimensional display by multidimensional scaling.
Galaxies plot this two dimensional space as a series of points, but tend to suffer from
information overload. Information Landscapes improve on this by constructing a smooth
three dimensional height surface, where the height indicates the relative concentration
of themes in an area. Graph Splatting [114] is a related technique that creates a height
field by convolving node positions obtained from a force directed layout with Gaussian
splatls. These methods however, offer a continuous representation of something that is
inherently discrete, and although nodes can be plotted on the resulting height field, the
actual relationships between nodes are often lost.

8.3 Force model

Force directed algorithms model a graph \( G = (V, E) \) as a physical system and then try
to find positions \( p_i \) for all nodes \( i \) such that the total energy in the system is minimal.
The force on a vertex depends on the attractive force exerted by its incident edges and
the repulsive force exerted by other nodes in the graph. Both forces are functions of the
distance between nodes. We can write the total force exerted on a vertex \( i \) as

\[
F_i = \sum_{i,j: e_{ij} \in E} f(p_{ij})|\bar{p}_{ij}| - \sum_{i,j: i \neq j} g(p_{ij})|\bar{p}_{ij}|
\]
with \( p_{ij} = |p_i - p_j| \) and \( \bar{p}_{ij} = (p_i - p_j)/p_{ij} \). The functions \( f(x) \) and \( g(x) \), which determine the attractive and repulsive forces respectively, typically follow physical analogies. For \( f(x) \) this is usually Hooke’s spring law:
\[
f(x) = A \cdot (x - x_0)
\]
where \( x_0 \) signifies the zero energy length of the spring. For the repulsive forces an inverse square law is used, inspired by electrostatic fields:
\[
g(x) = \frac{B}{x^2}
\]
Here, \( A \) and \( B \) are constants representing the strength of the attraction and repulsion respectively. The total potential energy of the system can then be expressed as
\[
P = \frac{A}{2} \sum_{e_{ij} \in E} (p_{ij} - x_0)^2 - B \sum_{i \neq j} \frac{1}{p_{ij}}
\]
In the optimal node configuration the potential energy of the system is minimal. Finding this global minimum directly is often not feasible, due to the high dimensionality of the problem, hence an iterative approach has to be used. A commonly used method is optimizing by steepest descent. Starting from a random configuration, the nodes are moved in the direction of the forces exerted on them, such that in the end the total force on each node is zero, or, in other words, a minimum energy state is reached. Often this will be only a local minimum, but usually the result is visually pleasant enough. Instead of using the standard \( O(N^2) \) per iteration approach we implemented a multibody method based on space subdivision [20]. This reduces the computation time per iteration to \( O(N \log(N)) \).

Conventional force models minimize the total variance in edge lengths, attempting to keep all edge lengths close to \( x_0 \). For dense graphs with a small diameter (such as small world graphs) this results in a uniform node distribution, because all nodes are kept close to their neighbors. Instead, force models that position tightly coupled groups of nodes closely together and loosely coupled groups of nodes far apart may provide much better results. Noack derived a class of force models for which this property holds, the so-called \( r \)-PolyLog energy models [127]. In these models the total potential energy is defined as
\[
P = \sum_{e_{ij} \in E} (p_{ij} - x_0)^r - \sum_{i \neq j} \ln(p_{ij}).
\]
Note that 3-PolyLog has an energy function that is equal to the one used by [62]. The most interesting in this respect however is the 1-PolyLog model, that has associated force functions
\[
f(x) = 1
\]
and
\[
g(x) = \frac{1}{x}.
\]
Noack proves for a simple 2 cluster case that by using the 1-PolyLog (which he dubs LinLog) model we obtain an embedding in which the distance between two clusters of
nodes is inversely proportional to their coupling. The coupling between two clusters $C_1$ and $C_2$ is a measure of their connectivity and is defined as $E(C_1, C_2)/|C_1||C_2|$, where $E(C_1, C_2)$ indicates the total number of edges connecting nodes in $C_1$ and $C_2$. Using a constant force attraction might seem strange at first, but it allows intra-cluster edges to grow while inter-cluster edges shrink. In [127] some results on artificially generated graphs with high clustering indices are provided.

Experimental results on real-world graphs using the LinLog energy model show that it is more susceptible to getting stuck in a local minimum than conventional methods when started from a random initial node configuration. Although this is hard to prove in general, we can reason that the force exerted on a node connected by a single long edge to the rest of the graph is much less in the LinLog model than in the 2-PolyLog model. This makes it easier for the combined repulsive forces from other nodes to prevent this node from reaching its optimal location. A better solution is not to use a random layout as a starting point but first create an acceptable layout with another force model. Generalizing this we used a $r$-PolyLog force model at step $m$ out of a total of $M$ steps ($0 \leq m < M - 1$) of the optimization process, where

\[
\begin{align*}
    r &= r_{\text{start}} & \text{if } 0 \leq m < t_1M, \\
    r &= \alpha_m r_{\text{start}} + (1 - \alpha_m) \cdot 1 & \text{if } t_1M \leq m < t_2M, \\
    r &= 1 & \text{if } t_2 \leq m < M,
\end{align*}
\]

with $0 \leq t_1 < t_2 < 1$, $r_{\text{start}} \geq 2$ and $\alpha_m = \frac{t_2M - m}{t_2M - t_1M}$.

Figure 8.2 Layouts with random initialization: (a) LinLog force model ($P = -381705, M = 1500$) and (b) Our force model ($P = -544602, M = 1500, t_0 = 0.5, t_1 = 0.6, r_{\text{start}} = 2$). See also color plate A.13.
This resulted in layouts with a substantial smaller energy than the ones we obtained starting from a random initial positioning in the same number of iterations, as is shown in figure 8.2.

8.4 Visualization

Given a layout $L_G$ of a graph $G$, we can state that highly connected subgraphs are now visible as areas with a higher node density. However, the image shown in figure 8.2(b) is still noisy and hard to read. Apart from that, displaying all nodes and edges simultaneously is simply not an option when dealing with larger graphs. We have to provide a way to reduce the number of visible elements, while maintaining the global structure of the graph. In other words, we have to provide a visual abstraction of $L_G$. At the same time the user will also be interested in details, so we have to find an acceptable way to integrate detail with context. The visualization of edges also remains a problem, since the large number of long edges in the picture make it hard to trace individual connections. We discuss each of the problems mentioned above in the next sections.

8.4.1 Visual Abstraction

A simple but effective way to provide visual abstraction without losing a sense of the global structure is to render the nodes as overlapping spheres with constant size in screen space. The resulting configuration of partly overlapping spheres visually abstracts a cluster of nodes, showing it as a blob like structure. At the same time, the internal structure of a cluster is still visible as a shaded Voronoi diagram, giving an impression on how the cluster falls apart when we zoom in. By keeping the screen size of nodes constant while zooming in on the structure, we obtain progressively more details. In fact, this method can be seen as a fast alternative to the approach proposed in [154], with the added ability to continuously display the graph at any level of abstraction by zooming in or out. Figure 8.3 shows our sample graph at three zoom levels, nodes are colored based on birthdate. For example, we can see that 16th century painter Giuseppe Arcimboldo is classified as (a very early) surrealist, which is not so strange considering his paintings.

Another often used way to provide abstraction is to compute a clustering on $L_G$. In the layout the distance between clusters is inversely proportional to their coupling. This means that we can use the geometric distances between nodes as a useful clustering metric, since the most tightly coupled clusters should be geometrically closest. We perform a bottom up agglomerative clustering on $G$ using a distance measure $d(i,j)$ between two clusters $i$ and $j$. Define a cluster $k$ as a tuple $(c_k, r_k, d_k, f_k)$, where $c_k$ represents the cluster’s center, $r_k$ its radius, $d_k$ is a measure of the internal connectivity and $f_k$ its parent cluster. Initially we assign all nodes $i$ to separate clusters $(p_i, r_{base}, 0, \text{nil})$. Next, we iteratively merge clusters by selecting the two clusters $i$ and $j$ with minimal $d(i,j)$ and merging them in a new cluster $k$. For all edges $e_{ij}$ incident to $i$ or $j$ we create a new edge $e_{k}$. If an edge $e_{k}$ already exists we increase its weight by $w(e_{ij})$. 


8.4 Visualization

The new cluster has to provide some visual feedback on the amount of leaf nodes it contains and their average position. Let $LN(k)$ be the set of leaf nodes in cluster $k$. A straightforward choice for $c_k$ is the average position of all leaf nodes $l \in LN(k)$. We let the radius $r_k$ depend on $|LN(k)|$. Linear (radius is proportional to size), square root (area is proportional to size) or cube root (area is proportional to volume) functions are possible candidates for $r_k(|LN(k)|)$. We found that the square root function provides better results in general, since a linear function sometimes leads to excessively large clusters, while a cube root function makes it too hard to discern size differences.

This leaves us with choosing a suitable distance function $d(i, j)$ to determine the distance between two clusters. Often used metrics in clustering literature define the intercluster distance as a function of the distances between their (leaf)members [82]. Well known variants are minimum link, maximum link and average link distance (See also section 3.2.1). Minimum link defines the distance $d(i, j)$ between clusters $i$ and $j$ as the distance between the two closest members of both clusters, maximum link uses the distance between the two furthest members, while average link uses the average distance between all members. Clustering using the minimum link measure has the tendency to form long chains, while maximum link clustering is very sensitive to outliers. Average link clustering generally provides a good optimum between both and is our metric of choice here. For the distances between cluster members (i.e., the leaf nodes) we can choose either the geometric distance or the edge length, with the edge length between unconnected nodes set to infinity. The latter method does not allow clusters that are not interconnected but positioned geometrically close by a suboptimal layout to be merged. Although aesthetically less pleasing, we use the edge lengths for clustering because it creates clusters that reflect the structure of the graph better.

Applying the cluster algorithm outlined above results in a binary hierarchy or dendrogram, where the (continuous) level $d_k$ in the hierarchy for a cluster $k$ having children $i$ and $j$ is equal to the maximum of $d(i, j)$, $d_i$ and $d_j$. Another way to view $d_k$ is as the maximum distance between any two leaf nodes in the cluster. A simple way to create abstractions for a clustered graph is to define a global degree of abstraction $DOA \in [0, 1]$.
118 Chapter 8 Small World Graphs

and display only those clusters \( k \) for which \( d_k \leq d_{\text{root}} \cdot \text{DOA} < d_f \) (figure 8.4). To create a continuous transition when \( \text{DOA} \) is changed we can interpolate between the size and position of cluster \( k \) and the size and position of its parent cluster using a local parameter \( \lambda_k \) where

\[
\lambda_k = \frac{(\text{DOA} \cdot d_{\text{root}} - d_k)}{(d_f - d_k)}
\]

\( \lambda_k \) is a parameter that indicates the amount of interpolation between the position and size of a cluster \( k \) and the position and size of its parent \( f_k \) based on the degree of abstraction at that point. We can state that \( 0 \leq \lambda_k \leq 1 \) for all visible clusters, because \( d_k \leq d_{\text{root}} \cdot \text{DOA} < d_f \). Based on user preference we can either use linear, exponential or slow-in slow-out interpolation (figure 8.5). We found a combination of a linear interpolation for the positions and an exponential interpolation for the cluster size to give the best results. Interpolating cluster sizes exponentially prevents small isolated clusters from growing to their parents size too quickly. Using \( \lambda \) we can compute sizes and positions for all clusters visible in the current ‘slice’ of the hierarchy and we can show the graph at an arbitrary level of detail by varying the DOA. A similar idea was coined in [137], but their ‘horizons’ are defined on a hierarchy with discrete levels and the hierarchy is generated based on a simple recursive space subdivision technique that does not always generate very accurate clusters. The procedure outlined above is in effect a continuous version of this idea. Figure 8.8a shows our artists graph at a higher level of abstraction, representing a reduction of 97% in the number of visible nodes.

8.4.2 Clustering comparison

To judge whether the visual clustering provided in the layout is a good one, we compared the results of the layout against two other clustering algorithms and a base line clustering computed by experts.
8.4 Visualization

Baseline clustering

We added expert data on a painters’ artistic movement using the same source [133] and used it to color the nodes (Figure 8.2b). Unfortunately, we were unable to provide movements for all artists in the database and unclassified artists are colored in grey. For all modern movements (right half) the image shows a clear coherence between the movement and the clustering in the layout. For most classical painters (left half) this coherence is much less clear however. Although we are able to find some small structural clusters based on movement, we suspect that most pre-17th century artists are likely to be clustered based on another property, such as geographical location.

MCL clustering

We also used the public implementation of the MCL algorithm (see also 3.2.2) to compute a clustering. MCL has one main free parameter (inflation) that governs the number of resulting clusters. Higher values result in higher cluster granularity. We used an inflation value of 3 resulting in 59 clusters. Higher inflation values resulted in too many (94) clusters, lower values resulted in too few clusters (26).

Figure 8.6 shows the computed layout in which the 15 largest clusters computed by MCL have been indicated in color. Generally speaking, MCL clusters are a good match with the baseline clusters, and correspond well with the layout. To further investigate whether the partitions computed by MCL are accurately reflected in the layout we plotted the lengths of inter-cluster edges against the lengths of intra-cluster edges. Note that the

---

Figure 8.5 Interpolation of sizes and positions based on current DOA. Linear interpolation on the left, exponential interpolation on the right.
The cutsize of the partition MCL found is 1172 out of a maximum possible of 2486, meaning that almost half of the total edges in the dataset are inter cluster edges. This is testament to the difficulty in properly clustering this dataset.

We expected to find short edge lengths for edges inside clusters, while edges connecting clusters should have longer edges. The plot in figure 8.6b shows that nodes in the same MCL cluster generally have short edges between them. This means that MCL clusters will be rendered as compact groups. On the other hand, nodes that are not in the same cluster can also have short edges between them (although less frequently). This means that MCL clusters are not always clearly separated in an image.

To further investigate the correspondence between the MCL clustering and our clustering at different levels of abstraction we computed four different MCL clusterings with different inflation values. We then abstracted the layout (using the average edge length measure) to show the same number of clusters and computed the Rand index [139], a measure of how well two partitions correspond. The Rand value is computed by dividing the total number of pairs that are correctly clustered (that is, the sum of the number of pairs that are in the same cluster in both partitions and the number of pairs that are in different clusters in both partitions) by the total number of possible pairs. A Rand index of 1 signifies complete correspondence, a Rand index of 0 indicates no correspondence. A problem with the Rand index is that the correspondence between two random clusterings will never be 0. We therefore also computed the more conservative adjusted Rand index [92] that takes this into account.

Table 8.1 shows the Rand and adjusted Rand indices for different inflation values. The
Table 8.1 Cluster correspondence between MCL and the layout.

<table>
<thead>
<tr>
<th>MCL Inflation</th>
<th>Clusters</th>
<th>Rand index</th>
<th>Adjusted Rand Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>26</td>
<td>0.92</td>
<td>0.34</td>
</tr>
<tr>
<td>3</td>
<td>59</td>
<td>0.97</td>
<td>0.48</td>
</tr>
<tr>
<td>4</td>
<td>94</td>
<td>0.98</td>
<td>0.39</td>
</tr>
<tr>
<td>5</td>
<td>111</td>
<td>0.98</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Rand values are very high for all partitions. The adjusted Rand values average around 0.40, indicating a reasonable correspondence between both partitions. Given that the layout optimization procedure ends up in a local minimum in most cases this is not a bad result.

**Newman’s method**

Intrigued by the method used by Newman (see 3.2.2 and [125]) we decided to implement it and apply it to our test dataset. Results using this method were disappointing however. Newman’s algorithm tries to make a greedy decision on clustering based on local connection characteristics of the nodes. From the resulting dendrogram we can see that the method is very prone to chaining in this case. It only finds a handful of very large clusters, along with some smaller rest-clusters. The optimal partition according to this algorithm consisted of 8 clusters.

Based on the above results we can say that the layout computed by the modified spring embedder algorithm is a reasonable match with a clustering computed by a dedicated algorithm on multiple levels of scale.

### 8.4.3 Detail and Context

Providing both detailed information as well as a global context in one image is one of the fundamental challenges in Information Visualization. Although we are able to provide views of a graph at different levels of abstraction, the user will probably be more interested in obtaining detail information for some specific section of the graph. One solution is to embed this detail in the global graph structure in a useful way, allowing the user to maintain both a context for the local area he is interested in, as well as a consistent global mental map of the entire structure. For this we employ a semantic fisheye-view. In section 3.3.2 we explained that fisheye techniques can be classified as either distortion oriented techniques or as data-suppression techniques.

Distortion oriented techniques provide a local geometric zoom that is easily understandable, but still require all nodes to be drawn. Data suppression techniques avoid this and only display a fraction of the nodes, but they require extra attention to keep the relationship between nodes and their abstractions understandable. Standard ways to do this
are by visual containment (i.e., always displaying a node visually contained in its abstraction) or animation (by animating from an abstracted layout to a detailed layout). Since both techniques have their individual merits, we propose to integrate them in a single scheme (fig 8.8c).

That is, we use a geometric fisheye by varying the local zoom factor with the distance from a focus and a semantic fisheye by varying the DOA with the distance from the focus. We therefore divide the visualization area into three concentric sections, based on the distance $s$ of a point $p$ to a focal point $f$ (figure 8.9): A focal area $A$ where $s \leq R_f$, an area $B$ with $R_f < s \leq R_{DOA}$ in which we increasingly abstract nodes and an area $C$ in which we keep the degree of abstraction constant. To also allow zooming into tightly packed area, we keep the radius of these areas constant in screen space. In other words, we use an increasingly smaller focal area when geometrically zooming into an image.

For points within the focal area $A$ we use a variable geometric zoom. We prefer a function $Z$ that magnifies sections close to $f$ but does not distort when $s \geq R_f$. A suitable function is the Fisheye Distortion function from [145], i.e.,

$$Z(s) = \begin{cases} 
\frac{(s+1)^r}{s^{r/R_f + 1}} & \text{if } s < R_f \\
s & \text{if } s \geq R_f
\end{cases}$$

In this case $z$ is a factor that determines the amount of zoom near the focus. Instead of also scaling the node sizes, like a common fisheye lens does, we keep the size of nodes in screen space constant, analogous to the abstraction method mentioned at the beginning of

**Figure 8.7** Comparison with Newman’s fast [125] method: (a) Resulting unbalanced clustering; (b) Dendrogram of agglomeration order. Note the chaining effects. See also color plate A.16.
the previous paragraph. This gives the visual effect that a tightly packed cluster of nodes gets ‘pulled apart’ near the center of the fisheye lens.

For points outside the focal area we use a semantic fisheye, that is, points \( p \) for which \( R_f < s \leq R_{DOA} \) are increasingly abstracted with increasing distance from the focus (area \( B \)). In other words, instead of a single global abstraction level \( DOA \), we define a function \( DOA(s) \in [0, 1] \) that specifies a degree of abstraction as a function of the distance \( s \) of a point \( p \) to \( f \).

We then render the graph by only rendering the highest clusters in the hierarchy \( k \) for which

\[
d_k \leq d_{root} \cdot DOA(|c_k - f|) \wedge d_{root} \cdot DOA(|c_{f_k} - f|) < d_{f_k}
\]

We can find these clusters quickly by inspecting the hierarchy in a top-down fashion. We use a DOA function that does not abstract nodes in area \( A \), is increasing in area \( B \) and has a constant value in area \( C \):

\[
DOA(s) = 0 \quad \text{if} \quad s < R_f
\]

\[
DOA(s) = a(s) \quad \text{if} \quad R_f < s \leq R_{DOA}
\]

\[
DOA(s) = a(R_{DOA}) \quad \text{if} \quad R_{DOA} < s
\]

The choice of an increasing function \( a \) is arbitrary, but linear and cubic functions are straightforward choices here. Figure 8.8 shows samples of both. By using a constant DOA for nodes in area \( C \), we avoid changing the interpolated sizes and positions for these nodes when the focus changes. Since movement is such a strong visual cue, viewer attention might get distracted from the focal area to the context, which is undesirable. Figure 8.9 shows these concepts schematically.

Summarizing the above, a node \( i \) at distance \( s = |p_i - f| \) from the focus point is displayed by:
Figure 8.9 View of visualization area indicating the three different methods of visual abstraction: area A uses a fisheye lens to distort node positions, area B incrementally abstracts nodes and area C displays nodes with a constant DOA to avoid unnecessary motion in the periphery.
Figure 8.10  Series of frames showing the effect of a changing focus on the lay-out. Surrealist artists are indicated in red, artists belonging to the Pop-Art movement in green. See also color plate A.18.
1. **abstracting** the node based on a function $DOA(s)$.

2. **projecting** the abstracted node at position $f + Z(s) \cdot \frac{p - f}{s}$.

By visually indicating the edge of the fisheye distortion, we make these two different zooming actions (one geometric and the other semantic) more understandable for the user (Fig 8.8c). The overall effect obtained when moving the focus resembles the Magic Lens [31]. Fig 8.10 shows a number of intermediate frames when moving the focus. The fact that only $O(\log(N))$ nodes are visible on average (depending on the choice of the $DOA$ function) allows us to maintain interactive performance. We also provide a consistent, albeit abstracted, picture of the context. In full geometric distortion techniques the positions of items in the context depends on the current position of the focus. This prevents the creation of a consistent mental map.

We have applied our method to various other graphs. As an example, figure 8.11a shows the largest connected part of the data set used in [114]. The nodes represent 824 papers in the proceedings of the IEEE Visualization conferences from 1990 to 2002, with edges denoting citations. We could identify several main research areas by looking at the conference session the papers appeared in.

An obvious extension to the method outlined above is the use of multiple foci. We can simply change the current function $d(p, f)$ that returns the distance of a point $p$ to the focus, with a function $d_M(p, F)$ that returns the distance to the nearest focus in a collection of foci $F$. Figure 8.11b shows a similarity graph on 10,000 pop artists with two foci. Layout took 4.5 minutes on a Pentium IV 2.2GHz, clustering took another 3 minutes.

### 8.4.4 Edges

So far we have only considered the visualization of nodes, this section discusses the visualization of edges. Traditionally, edges are visualized in node link diagrams as straight lines with a fixed width. As figure 8.2b shows, many long intersecting edges lead to a noisy image. We make the following observations:

- Line crossings can only be resolved by using the Gestalt principle of visual continuity, i.e., we assume that the direction of the lines after a crossing is the same;

- The use of screen space of edges is proportional to their length;

- The longer edges account for only a small part of the total number of edges.

Based on the first observation, we can improve visual continuity by displaying edges as shaded tubes. The shading augments the visual continuity, and also depicts edges and nodes similarly. We plot short edges closer to the viewer than long edges, resolving most intersections and de-emphasizing longer edges. Note that for larger graphs (such as the one figure 8.11b) we still have to fall back on simple unshaded lines to avoid the large rendering overhead.
Figure 8.11 Larger samples: (a) Citations between papers published in 12 years of IEEE Visualization conference proceedings, volume visualization in blue, flow visualization in red, terrain visualization/surface simplification in green and information visualization in yellow; (b) Similarity relations between 10,000 pop artists. The visualization uses two foci, one on country (right) and one on reggae (top). Dataset courtesy of Overture/Yahoo Research. See also color plate A.19.
As a consequence of the second observation, long edges use up a large part of the available pixels, although they are no more important than short edges. To resolve this imbalance we aim to keep the amount of space for each edge constant. However, simply keeping the product of width and length of edges constant leads to edges that are too thin, and also, we are using 3D tubes here. Therefore, we keep the volume of an edge constant.

Based on the third observation, we can state that a small portion of the edges takes up a large portion of the display space. In our case, we found that the 5% longest edges account for 29% of the total edge-length and hence use up almost one third of all pixels allocated to edges. We can decrease the total number of pixels used for the display of edges in the graph substantially by leaving out only a few percent of the longest edges. Since simply eliding these edges would cause a loss of information, we instead choose to draw a small fraction (say, 5%) of the longest edges transparently.

8.5 Conclusions

We have presented a new method that enables us to get insight in the structure of small world graphs. Small world graphs are notoriously hard to visualize because of their highly connected nature. Our method contributes:

- A modification to a force model from [127] in order to obtain layouts that better reflect the natural cluster structure of the graph.
- A method for continuous visual abstraction that combines both explicit clustering and visual clustering. Both clustering methods provide smooth interpolation between various levels of detail.
- The combined use of the previous techniques in an interactive generalized fisheye scheme, using both semantic and geometric distortions.

The resulting visualization shows an abstracted view of the entire graph, in which a close up view is integrated. When provided with a balanced clustering, on average only $O(\log(N))$ clusters are visible and the system is fast enough to perform at interactive speeds. If the computed clustering is very unbalanced (i.e. a majority of the nodes clumps together in one tight group), we need to apply a geometrical zoom to this group first, to avoid expanding them all at once with the focus.

A possible improvement concerns the amount of uncertainty in the visualization. The further away from the focus a cluster is, the more inaccurate its representation. The current node representation (i.e. a sphere) gives the impression of a data item with a clearly defined position and size. An alternative might be to use splats for node representation and vary the shape of the splat with increasing distance to $f$. That is, we use a sphere for nodes close to $f$ and a Gaussian splat for nodes in the periphery. Also, automatic labelling of clusters is an important issue. Currently the meaning of a cluster can only be determined by inspecting leafnodes inside that cluster manually. The usefulness of these visualizations would certainly be increased if we could characterize each cluster automatically.
Chapter 9

Reflection

In the previous four chapters we have presented four very different visualization methods for four very different classes of graphs. In this chapter we first compare them to discover differences and commonalities. Based on these commonalities we extract four key problems that we ran into in each of the visualizations, and which we think crucial in any large graph visualization. Finally, based on these insights we present a number of practical guidelines that one can adhere to when constructing a visualization for a new class of graphs.

9.1 Comparison

We compare both the flexibility and scalability of the visualizations from the previous four chapters. Ideally, we hope we can identify one visualization as superior over the others and use it as a basis for further research.

9.1.1 Flexibility

In this section we assess the flexibility of the methods, that is, to what extent visualization methods can deal with graph types that are outside their original scope. We therefore applied each of the methods to a dataset from the domain of another method. Figure 9.1 gives a visual overview of the results.

Beamtrees are clearly the least flexible of the four, since we can only apply them to trees. In visualizing trees they outperform the other visualizations in both speed and readability however. The graph we used for testing was a 20K node graph representing a C:\windows subdirectory. The FSM visualization suffers from the fact that the clustering algorithm puts each node in a single cluster and creates a conetree-like representation. Another problem here is that the degree of some nodes in the hierarchy is very high, leading to layout problems. The matrix-based visualization required us to create a clustering hierarchy for the tree. We therefore grouped all leafnodes (i.e., files) with their respec-
Figure 9.1 Comparing the visualizations from the previous four chapters: each row shows four different visualizations of a single dataset. First row shows a 20K node tree, second row shows a 3K node finite automaton, third row shows a 30K node layered software structure and the bottom row shows a 500 node small world graph. See also color plate A.20.
9.1 Comparison

tive parents reducing the hierarchy to 936 nodes (i.e., directories). The image shows the directories with the highest number of files as the biggest cells. The important fact that the input graph was a tree however, is not clearly visible. The Graphview visualization performs better in this respect, but needed over two hours (!) of computation time and some manual adjustment to compute a decent layout of the tree. It was very susceptible to local minima because internal nodes with a high number of child leaves did not move to an optimal position. As a result branches of the tree overlap and the structure is not always clear. With some effort we can still extract the two directories with the highest number of leafnodes (windows and windows\system).

The finite automaton graphs lead to nice clean, symmetrical pictures by the FSM visualization. The image clearly shows that the graph consists of three separate highly symmetrical parts and that there is a branching in the first section of each part. For the matrixview we created a hierarchy by recursively applying MCL to the original input graph. The matrix image does show a repeating structure, but with subtle differences between each repetition. It is doubtful that more complex symmetries can be detected by using a matrix representation. The Graphview visualization managed to untangle the 3000 node input graph correctly, again at the expense of a long computation time. Graph characteristics like branching and symmetries can still be seen, though far less clear than with the FSM visualization.

For the matrix graph we used the low level inter-class calls from the Philips dataset [76] containing 30K nodes. To avoid problems with disconnected components we selected the largest 8K component. The problem with the finite state visualization is that there is no single node that we can appoint as startnode. This makes the images very hard to interpret since there is no intrinsic node ordering in the software system. Such ordering is present in finite automate systems, because distance from the initial state can be thought of as approximately equivalent with execution time. The Graphview visualization is unable to provide a good layout for the largest component in this graph because it is too dense. We used a metric called peeling [3] to indicate highly connected classes. The red group corresponds to the EventNotification infrastructure, which is also visible as a strong vertical band in the matrix visualization.

The 500 node artists graph rendered by the FSM visualization suffers from the same problem as the call-graph in the previous example. We cannot identify a specific initial state, making it hard to assign any relevance to this specific picture of the graph. We could randomly choose another startnode and end up with a completely different picture. For the matrix visualization we used MCL again to create a two level hierarchy on the graph. The color of cells indicate the aggregated weight of the calls they represent. Although most high weight edges are near the diagonal of the matrix, the relatively large number of off-diagonal cells indicates that this dataset is hard to cluster.

Judging from the pictures in figure 9.1 the graphview visualization seems the most flexible, providing a reasonable layout of sparse graphs at the expense of a longer computation time. If the graph is large and dense the graphview visualization performs sub-par, and a matrix representation might be more appropriate. Another observation that we can make based on this small sample is that a visualization that is tailored to the characteristics of the input graph outperforms a general visualization.
9.1.2 Scalability

With scalability we mean the ability of a visualization to deal with graphs of increasing sizes. Although commonly the size of the graph is equal to its number of nodes $N$, we also take the density of the graph into account in our discussion. Scalability can be divided into computational scalability and representational scalability. The former makes a statement on the complexity of the algorithms involved, while the latter refers to the efficacy of the visualization method itself. Table 9.1 gives an overview of the practical limits of each of the visualizations.

In the Beamtrees visualization the number of edges is always equal to $N - 1$. The computational complexity of the visualization is linear, making it suitable to relatively large trees. In practice we can show representations of trees between 50K and 100K nodes with relative ease. Beyond this point rendering and interaction become problematic. Trees with a high internal branching factor reduce the effectiveness of the visualization since we have to render a single beam for each internal node.

In the Finite State Machine visualization the average degree of a node is constant [132], so the number of edges will be linearly related to the number of nodes. The FSM visualization relies on the clustering algorithm to reduce the potentially large graphs to smaller manageable trees. This algorithm can be implemented in $O(|N| + |E|)$, allowing us to generate high level overviews of a million node graphs in a number of seconds (see table 6.1). At this point memory concerns become an issue, as the application uses about 1K/Node. The generation of a detail overview can take more time however, because the algorithm to layout the individual nodes has worst case $O(N^2)$ complexity. Transition graphs with a small diameter severely reduce the performance of the clustering algorithm and provide less desirable results.

In the Matrixzoom visualization we paid special attention to memory issues and created a visualization that can deal with graphs in which the edge set does not fit RAM. We did not confine ourselves to a specific application (although the initial use case was a software architecture) so the density of our input graphs can vary. However, since matrices are also effective for very dense graphs, matrixzoom can deal with most graphs as input. The only requirements are a node set that fits RAM along with a suitable hierarchy on this set. In practice this means that it can handle graphs with a few million nodes and several millions of edges.

The Graphview visualization offers a familiar and flexible representation but suffers in terms of scalability. The entire geometry of the visualization is based on a single layout of all nodes in the graph. This layout step takes time $O(N^2 \log(N))$ and takes up the majority of computation time. Apart from that, the visualization of graphs with higher density is problematic, as is witnessed by the Graphview visualization of the software system in figure 9.1. Graphview’s representational limits are sparse graphs of at most 10K nodes. Beyond that both computation time and graph representation become a problem.

It is interesting to see that although the GraphView visualization is the most flexible one, it also scales the worst. In general, we think there will always be a trade-off between flexibility and scalability. Visualizations of very large graphs are often tuned specifically to the underlying data and task, just to be able to cope with the size of the dataset. It is
9.2 Common design aspects

One aim of this research was to identify core aspects that one faces in general when designing a graph visualization. In developing visual representations for graphs a number of high level problems kept surfacing. Inspired by [120], we propose the following four problems as fundamental in the creation of a visualization of any large graph:

- **Representation**: How to design a suitable representation that maps a relation between two items to the screen?

- **Abstraction**: How can we define a suitable way to abstract the graph? Creating a good abstraction avoids displaying all relations in a massive graph at once.

- **Navigation**: How can we integrate the abstracted view(s) resulting from the previous step and the detailed connectivity information in the original input graph in such a manner that the user understands their interconnection?

- **Mapping**: How can we map the abstract visual representation of a graph back to the real world entity the user is familiar with? Without such a mapping it is impossible for the user to derive any useful insights from a visualization.

In the next sections we discuss the above mentioned problems in more detail and we evaluate how we approached them in our prototypes. Table 9.2 gives a comprehensive overview and the schema in figure 9.2 shows interrelations between these problems from a developer perspective.

<table>
<thead>
<tr>
<th>Visualization</th>
<th>Rep.Scal</th>
<th>Comp.Scal</th>
<th>Max Density</th>
<th>Main practical problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beamtrees</td>
<td>$10^3$</td>
<td>$10^6$</td>
<td>Linear</td>
<td>Trees with high internal branching</td>
</tr>
<tr>
<td>FSM</td>
<td>$10^7$</td>
<td>$10^6$</td>
<td>Linear</td>
<td>Small diameter graphs, memory issues</td>
</tr>
<tr>
<td>Matrixzoom</td>
<td>$10^6$</td>
<td>$10^6$</td>
<td>No Limit</td>
<td>Obtaining suitable hierarchies</td>
</tr>
<tr>
<td>Graphview</td>
<td>$10^5$</td>
<td>$10^4$</td>
<td>Log</td>
<td>Dense graphs</td>
</tr>
</tbody>
</table>

Table 9.1 Estimates of theoretical bounds on both computational and representational scalability. Actual numbers may be lower depending on characteristics of the graph.

precisely this customization that decreases the flexibility of the visualizations. The Finite State Machine visualization is an excellent example of this trade-off between the use of the input graph’s properties for the visualization and the ability to apply the visualization to other areas. The Finite State Machine visualization assumes that the input graph is sparse and has a large diameter, and that the user is interested in seeing internal symmetries. If this is not the case, the usefulness of the visualization will suffer.
Figure 9.2 Relations between core data mapping problems of representation, abstraction, navigation and mapping viewed from a developer perspective. Grey blocks indicate the expanded information visualization pipeline as defined by [38], the traditional information visualization pipeline sees visualization and presentation as a single block.
9.2 Common design aspects

9.2.1 Representation

The first core problem one faces when visualizing large graphs is choosing a suitable graphical representation for a given data set. Currently a number of different graphical ‘languages’ are available. The list below is exhaustive to the best of our knowledge:

- **Node-link Diagrams**: Probably the most well-known and recognisable representation, node-link diagrams represent a node in a graph with an arbitrary glyph and a connection between two nodes with a connector. The choice of glyphs and connectors is arbitrary, but basic shapes such as circles or squares and (straight) lines are often used. Node link diagrams can provide pleasing pictures for sparse graphs with large diameter, but have trouble with dense graphs. Computing a ‘nice’ node-link layout remains computationally expensive.

- **Matrices**: A less well known representation, matrix representations view the edge set of the graph as a set of tuples \((x, y)\) and render it as a scatterplot. The only free variables in this representation are the order of the rows and columns. We can either compute an approximation to an optimal ordering (since computing an optimal ordering is NP-complete [111]) or provide the user with a random or user specified ordering. Matrices can deal with graphs of arbitrary density and can easily be abstracted as we have shown in Chapter 7.

- **Containment**: A relation between nodes \(x\) and \(y\) can also be shown by containing the representation of \(y\) within \(x\). Since containment cannot deal with cycles in the containment relation it is only really useful to show the dependency relations in trees or DAGs. Treemaps are the main proponent of the containment approach.

- **Overlap**: Overlap can show a unidirectional relationship between two items, by drawing one node glyph over the other. This representation can be applied to general graphs, in effect merging the separate node and edge representations. We applied it to trees, resulting in visualizations as presented in Chapter 5.

We identified a few general principles that affect the overall readability of a particular graph representation. One of these is *visual closeness* that states that the distance between the visual representations of nodes \(x\) and \(y\) should be somewhat proportional to the graph theoretical distance of \(x\) and \(y\).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Beamsizes</th>
<th>Static Transition Graphs</th>
<th>Matrixzoom</th>
<th>Small world graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation</td>
<td>Overlap</td>
<td>Abstracted Node-Link</td>
<td>Matrix</td>
<td>Abstracted Node-Link</td>
</tr>
<tr>
<td>Abstraction</td>
<td>None</td>
<td>Structural</td>
<td>Data, Visual</td>
<td>Structural, Visual</td>
</tr>
<tr>
<td>Navigation</td>
<td>Geometrical</td>
<td>Two-step zoom</td>
<td>Multi-level zoom</td>
<td>DCC, DCC</td>
</tr>
</tbody>
</table>

Table 9.2 Different solutions for the core graph visualization problems. SL, DL, SCC and DCC signify static and dynamic labelling and static and dynamic color coding respectively.
Figure 9.3 Four different representations of a graph: (a) Node link representation; (b) matrix representation; (c) containment representation; (d) overlap representation.

Not following this principle leads to graph visualizations in which the global structure is harder to extract, because there is no spatial ordering in the data. The matrix-based visualization does not adhere to this principle, since the two endpoints of a single edge are positioned on different axes of the matrix. User provided hierarchies (as in Chapter 7) or optimized orderings help in keeping clusters of nodes together but extracting global connectivity information from matrix based information remains difficult.

A downside to this closeness principle is that in dense graphs most nodes will be close to each other. This, combined with the large number of edges in dense graphs, results in often unreadable node-link diagrams of dense graphs. A possible solution then, is to infer relations by closeness only and not explicitly visualize all edges. In other words, if two nodes are visually close they will probably (but not certainly) share an edge and vice versa.

A potential ordering in the node set can also be an important factor in deciding on the desired representation. Ordering can be done on any ordinal node attribute. This includes attributes that are part of the data (for example ‘year of birth’ in the artists data set from Chapter 8) or node attributes that can be computed from the data. In Chapter 6 we used a computed measure that represents the distance from the initial state as an ordering, since it approximately corresponds to execution time.

### 9.2.2 Abstraction

Another problem in graph visualization concerns the scale of graphs. Even if we could compute a layout for a 10K node graph and embed it efficiently, a user will have a hard time extracting global knowledge from this visualization. If we can provide the user with a suitable coarser abstraction of the graph instead, we can use this smaller graph to provide a high level overview of the entire structure. Possible abstraction methods include (but are not limited to):
9.2 Common design aspects

- Data abstraction: Data-abstraction uses domain attributes associated with nodes and edges to group them into clusters. The advantage is that a semantic meaning can usually be assigned to a cluster. For example, in software call graphs the name of the method called usually gives an indication on the architectural component the method resides in (Chapter 7). However, we have to carefully choose our domain attributes to cluster. If the attribute is unrelated to the structure, for example the first letter of the name of an artist in Chapter 8, we end up with a random clustering that does not tell us very much.

- Structural abstraction: This assumes we have an algorithm that operates only on the input graph structure and returns a graph of smaller proportions. The challenge is to design an algorithm such that the abstracted graph is still a faithful representation of the structure of the original graph. Traditional graph clustering algorithms (see section 3.2) often provide reasonable results, but suffer from high computational complexity. In Chapter 6 we detailed an algorithm that clusters nodes based on confluencing paths and operates in linear time. Finding a general method that can create a suitable structural abstraction for any type of graph in reasonable (preferably linear) time is still an open problem.

- Visual abstraction: Instead of performing the abstraction on the input data, we can perform the abstraction step on a visualization of the input data. That is, we group nodes that visually form a coherent cluster into a single visual entity. Visual abstraction reduces the amount of data that is visible on screen, thereby reducing the rendering load on the machine and cognitive load on the user. It does still require us however to compute a good initial visualization of the complete underlying data set. This can be achieved by creating a layout that matches the clustering structure (Chapter 8) or using a given hierarchy to re-order a matrix (Chapter 7).

A match between the data abstraction and the structural abstraction means there is a correlation between the structure of the data set and some of the attributes in the data set. Such relations are usually hard to detect with conventional analysis. Examples include the correspondence between artist birthdate and graph structure in figure 8.3. A match between the visual abstraction and the structural abstraction indicates that the visualization is a good rendition of the underlying structure of the graph. In an ideal case there should be a correspondence between all three abstraction methods.

9.2.3 Navigation

The navigation problems result from the abstraction problem. In most cases a user will be interested in answering questions with a varying scope (see section 2.2). This means that the visualization needs to have a facility that allows the user to mentally integrate the different levels of detail provided by the different abstractions. One often used way is to allow a user to ‘click open’ a high level abstraction and view the details at the level below. However, if the number of steps between the highest and the lowest abstraction level is high (in practice: bigger than two) the user will tend to lose his or her orientation.
in the data set. This loss of context makes it hard to mentally integrate detail information within the global overview of the entire graph. We have implemented different navigation solutions, ranging from limiting the number of steps to two in Chapter 6, providing small higher level maps of the graph in a corner of the display (Chapter 7) and providing a detail view that is integrated within the overview in Chapter 8. One technique that works well is the use of animation (Chapters 7 and 8) to provide a smooth transition between two different views. User tests [25] have shown that users integrate pieces of information more easily when provided with a short animation showing their relationship. Another technique that appears to work well is to provide simultaneous access to items through multiple different views. For example, through the main visualization and a traditional one-dimensional tree browser on the side.

9.2.4 Mapping

With mapping we mean the ability of the visualization to transfer attribute information to the user. A visualization creates a (sometimes abstract) image from a real world data set, but it should also provide a mechanism to help the user translate these images back to the original problem domain.

One such mechanism is assigning descriptive textual labels to nodes and edges, also referred to as labeling. Without labeling, the nodes and edges of a graph remain abstract entities. Labeling not only involves the entities of the input graph itself, but also any abstractions that the visualization creates. Without descriptive labels for these abstractions they remain meaningless until the user inspects their contents, which can be quite time consuming. Providing a concise, meaningful label for a group of labeled entities is still a largely unsolved problem. Another problem in labeling is that static labels often take up too much screen space, resulting in overlapping, unreadable labels. A simple solution that we found particularly effective is the use of dynamic mouse-over labeling, in which a label for an item is only shown when the mouse is over that item.

Another useful mechanism is to assign colors to specific property values or entire color scales to a property. Color is a pre-attentive cue that quickly allows a user to assess whether there are any correlations between a property and the structure or if certain properties are specific to a cluster.

Typically the visualization should offer multiple means to map attributes to visual cues and let the user decide which on is most relevant for him or her at that time. Also, integration with other analysis tools can be invaluable for a user to obtain a holistic view of the data set. The abilities to export and import data to different programs, perform what-if analyses and link back directly to the data set in the application domain contribute greatly to the usability and acceptance of any visualization. In practice these mechanisms remain under used in academic prototypes because of the implementation difficulties involved.
9.3 Developing new visualizations

Which solution to use for the various aspects mentioned above in a new practical case depends on a number of factors. Because of the large variety in input graph characteristics and user aims it is impossible to give a general recipe for constructing a successful graph visualization. Also, a visualization that is customized for a specific task will generally outperform a multi-functional visualization.

Instead we present the insights that we have gained over the course of this work in the form of a number of general recommendations and guidelines. We divide them into groups based on the problem space subdivision we presented in Chapter 2 (data, aim and context) and indicate which of the general problems mentioned above they might be applicable to.

9.3.1 Data

Even though the mathematical definition of a graph is very simple, the resulting structures exhibit a large variety in characteristics. A visualization designer should examine and use these characteristics as a basis for his design. Specifically, we recommend the following pointers:

- Explore smaller versions of the data set manually. Try to get a feeling for the unique properties of this graph, i.e., properties that distinguish this graph from a random one. Example properties include ‘tree(like)’, ‘high diameter’, ‘clustered’ or ‘symmetrical’. Construct the visualization such that these properties are emphasized or exploited (representation, abstraction).

- It is a waste of computational resources to visualize unconnected graphs. Since there are no edges between unconnected parts, no information can be inferred from their relative placement. Always perform a connected component analysis as a first step and present the user with a list of components. If necessary, compute visualizations for each component separately and show them in a grid-like fashion (representation).

- Keep trying to push for bigger data sets throughout the design process. If a visualization works for a few hundred nodes it is very tempting to keep using relatively small data sets as input.

- If possible, try to extract a single dimension from the data set that can be used for spatial ordering in the visualization. For example, there may be an inherent time dimension present in the data set (such as in workflow graphs for example) (representation, abstraction).

- If the graph is dense, consider using a matrix representation. If the graph is dense and weighted, another option is to prune the original input graph based on edge weights and visualize the pruned graph (representation).
9.3.2 Aim

The other major design guideline should be the aim of the user. In practice this is often problematic because users do not have enough general knowledge on the graph data set to pose specific questions. Finding out what the aim of the user actually is can be quite challenging.

- Users often have no concrete idea what their specific aim is. Talk to users to find out what information they find important, and what information they need to solve a particular problem. Sit down with users and try to get a feel for the sometimes fuzzy problems they are dealing with. Try to put yourself in the shoes of the user by taking on their problems. If possible use an iterative prototyping approach to determine what the interests of the users are (representation, abstraction, mapping).
- Talk to users to find out what information they do not find important. Use this information to reduce the complexity of the graph (abstraction).
- Just because a user does not express interest in a specific property does not mean the user is not interested in seeing that property. Sometimes users are not even aware of properties until they are shown (representation).
- If a user indicates interest in a specific property of the dataset, there usually also exists interest in the absence of that property. For example, if a user is interested in relations between software components, the absence of a relation where the user expects it is also significant (representation).
- A visualization will never be able to fulfill all needs for all tasks. If possible choose a specific task and tune the visualization for that. A visualization that does one thing well is still better than a visualization that does a lot of things badly.

9.3.3 Context

Contextual factors have more influence on the implementation and deployment of a visualization than on its design. Nevertheless, any successful visualization will have to take these into account.

- A large scale successful visualization needs a direct coupling with analysis tools that are already established in that particular application area. Visualization is not a magic tool in itself, but should rather act as a unification between different views on a complex problem (mapping).
- Users have trouble mentally keeping track of a hierarchy if it is more than two or three levels deep. Offer alternative representations of the hierarchy in the same view or integrate the detail visualization in a global overview (navigation).
- Access to a visualization should be quick and effortless. Any user will be much more inclined to use a visualization that is operational in a few seconds than one that takes 5 minutes to load the data.
9.3 Developing new visualizations

- If the user already has a predefined mental image of the data, use this as a basis for your visualization (**representation**).

- Data formats will remain a problem, a successful commercial visualization will need to support all accepted data formats for that application area.

- Choose your data structures carefully. Try to optimize for memory from the beginning, but also keep them flexible enough to adapt to changes.

- Nobody will use a visualization when its response time is slow. Interactive exploration can already be time consuming even when the visualization is real time. A response time of over a few seconds for basic interaction will kill any interest in a visualization quickly (**mapping**).

The points mentioned above can serve as a good starting point for developing a successful graph visualization, but are by no means set in stone. Graph visualizations, like any visualization, also require inspiration, creativity and lots of failed attempts to get them right.
Chapter 10

General conclusions

In the past chapters we have sampled the space of possible answers to the original research question, namely: How can we use computer supported visualization to help users gain insight in the structure and content of large graphs, with thousands or millions of nodes? We developed four different solutions for four different classes of graphs and then compared them side by side in the hope to discover differences and common aspects.

One of the findings was that graph visualization beyond a certain scale usually exploits the characteristics of a given graph in order to be effective. It is precisely this customization that makes it more difficult to apply a successful prototype in other application areas. Comparing our prototypes, we found four fundamental sub-problems appearing in all of them:

- Representation: How do we choose a suitable visual representation for a graph?
- Abstraction: How can we abstract a graph such that the abstraction conveys the essential properties of the original?
- Navigation: How do we help the user in mentally integrating detail and overview information?
- Mapping: How do we relate the visualization and the information that it represents?

We have presented a number of possible solutions to all of these problems, along with a number of useful guidelines that can help in developing a graph visualization for a new problem. In this chapter we present concluding remarks on both graph visualization and evaluation along with suggestions for further research.

10.1 On graph visualization

We think that from a data perspective, visualizing large graphs is presently one of the two hardest problems in information visualization (the other one being visualizing large multidimensional datasets). Visualizing large graphs is (and will remain) tough because:
• There are many types of graphs and there is no single representation known that performs optimally for all of them. This fact alone rules out a completely generic solution.

• The connectivity of a graph can only be fully understood by examining all of its edges. One could make a statement on the approximate composition of a set by examining 99% of that set. It is much harder to make a statement on the approximate connectivity of a graph by examining 99% of its edges and nodes, because the missing 1% might specify a connection between two previously unconnected nodes.

• One of the reasons why abstraction of a graph dataset is hard is because it is very difficult to formally specify the notion of a natural graph cluster.

• There is a dichotomy between the abstract graph and its mental representation. On one hand non-planar graphs are abstract entities with no direct representatives in our tangible (spatial) world, on the other hand they appeal strongly to our sense of Euclidean spatial ordering (as is witnessed by the terms ‘diameter’, ‘distance’ and ‘neighbors’ in graphs). As a result, all but the most trivial graphs have many possible mental representations, all of which are, to some extent, inaccurate.

Although we have not restricted us to a specific type of graph, there are many types of complex graph data that we have not looked at, such as time dependent graphs (which add another dimension to an already complex problem) or visualizing differences between two graphs. In general, these types of problems are combinations of a multidimensional data set and a graph data set. Instead of a single scalar value, nodes and/or edges contain a vector of values, representing the data set(s) in which they are valid.

It should be clear from the preceding chapters that developing a graph visualization that performs well for any task on an arbitrarily sized graph from any application area is a virtually impossible task. However, developing a specific graph visualization is doable. In the previous chapter we have identified four core problems that will surface in any large graph visualization and provided possible solutions to each of them.

A variety of graph visualizations can then be obtained by mixing the different solutions to each of these subproblems. For example, we could design a visualization that clusters large trees and then visualizes them as node link diagrams while providing smooth animation between levels. Or apply a focus and context method to matrix visualizations. Even applying the same set of solutions to the same problem might lead to different looking visualizations, depending on the design choices for the details. Evaluating all these possible solutions is a problem on its own.

### 10.2 On evaluation

Out of the four techniques we presented in this thesis, three were evaluated. One of these evaluations was a formal user experiment (Chapter 5), the other two visualizations were
informally evaluated in practice by experts (Chapters 6 and 7). Evaluation in information visualization still remains a somewhat under-lit subject, although this has been improving in the last couple of years. We believe this is due to a combination of the following reasons, some conceptual and some practical:

- **Unknown what to evaluate:** The main purpose of a visualization is to help the user ‘gain insight’ in the dataset. However, the amount of insight gained by a user is difficult to measure and subjective because it depends on the knowledge already present and the innate ability of a user to digest and process visual information. Most formal evaluations therefore evaluate (simple) local tasks, such as finding the immediate neighbor of a node or locating a node with a specific property. How these simple tasks influence the creation of a global mental model is unknown. Apart from that, it is often easy to select a simple task set that lets the visualization under evaluation outperform others on that specific taskset.

- **Flexibility:** Most visualization tools are custom tuned for a specific type of dataset. Especially graph visualization tools tend to use underlying characteristics of the dataset in their visualizations for good reasons, as we have seen in the previous chapter. This makes comparing different graph visualization tools very much like comparing apples and oranges.

- **No public access to visualizations:** As most researchers do not make their prototypes publicly available after publication, it is hard to test new approaches against existing approaches.

- **Different researcher interests:** Strictly speaking, it is unfair to have the evaluation of a prototype done by the same people that built it. Also, most researchers interested in designing visualizations are often not very interested in evaluation and vice versa. Because of limited access to prototypes (see above) there is only a small number of visualization tools that make it into the hands of human-factor researchers.

- **Lack of domain experts:** Large datasets often spring from complex application areas. It is difficult to find a group of experts that are qualified to evaluate visualizations of these datasets and are willing to spend a significant amount of their (expensive) time.

We think the road towards finding an optimal solution for a specific type of graph should be an evolutionary one. This does not imply that we should let all solutions loose in the real (wild) world and see which ones survive, because in the real world visualizations get killed for not having an export function to Excel (specifically, version 2003) or because they are implemented in Linux where it should have been Windows and vice versa.

Instead, as a first step, we propose to create a benchmark set of real world graph datasets. This set should include datasets with different sizes and from different application areas, ideally with a set of associated tasks and problems. This allows relatively easier comparison of the capabilities of each visualization, because we can compare different visualizations of the same dataset side by side. If possible, we can also include a
list of different insights found by domain experts and score the visualization on how many of these it can correctly display. This allows us to separate weak visualizations from the strong ones and use the ideas in the strong ones to create better ones. In this respect the Infovis contest organized annually during the IEEE InfoVis symposium offers a good initial platform.

Of course, the practical implications are that any dataset used in a published visualization method should be publicly released along with the prototype. Many of these datasets and tools represent company IP property, which might make publishing these difficult. Yet we sincerely believe that if the information visualization field wants to evolve, it needs a better way to assess the quality of a visualization. Having a standardized suite of test datasets can be a first step in the good direction.

10.3 Future work

We have presented a broad space of sample solutions, yet there are a number of approaches that we have not tried yet. One less travelled road that might reduce the complexity of the graph is by randomly sampling its node or edge sets. Much like the properties of a large set of items can be measured by looking at a small random sample, maybe it is possible to make statements on the connectivity of a graph by looking at the connectivity of a random sample.

Another approach abstracts the whole graph by only visualizing the node set. Edges can then be inferred from the distance between two nodes: if they are close they will most likely be connected, if they are far they will most likely be unconnected. Detailed information can then be displayed by a mouse over function, or by the user selecting a set of origin and destination nodes. This largely avoids the edge overlap issue that node-link diagrams have with dense datasets. The layout method presented in Chapter 8 might offer a good starting point.

Further research should also focus on finding ways to efficiently create abstractions for large graphs. Although general clustering algorithms can offer decent abstractions for medium sized graphs in acceptable time, their $O(N^2)$ or higher running time often prohibits applications to really large samples. What is needed are (preferably linear time) algorithms that can quickly break down a large graph into structurally relevant parts, the connected component algorithm being a trivial example.

Finally, another interesting observation that we made is that visualization is much more easily accepted by decision makers than domain experts. The former group needs a compact high level overview of a complex problem, and preferably one that is easy to communicate. In this respect it is strange that the visualization community almost exclusively targets experts, who are mainly interested in the interrelations of details and thorough analysis. As a result most currently developed visualizations are interactive, complex and customizable programs that require a lot of interaction time. On the other end of the scale there is definitely also a need for visualizations that produce a single, clear, even static image of something complex.

In general, we believe graph visualization is indispensable when it comes to analyzing
graphs. It is simply impossible for humans to reason about the structural information in large graphs without drawing a picture, whereas computers lack the creativity and pattern recognition skills to come up with useful insights about graphs. The integration of the best of both of these worlds in both visualization and data mining may bring significant benefits in all areas of science in the years to come.
Chapter 10 General conclusions
Bibliography


[122] MILLER G. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. The Psychological Review, 63 81–97, 1956.


[146] SGI. 3D File System Navigator for IRIX 4.0.1+.


Plate A.1 Sample beamtrees: internal nodes/directories are blue, leaves/files are colored by type if available. (a) Filesystem rendered in 2D; (b) Same filesystem in 3D; (c) Layered orthogonal view; (d) Layered isometric view.
Plate A.2 Visualizing node attributes, the intensity of the color shows (a) depth of a cluster; (b) number of nodes in a cluster that have a specific property value and (c) the number of times a cluster was visited during a random walk.

Plate A.3 Closeups: (a) section of the 2-jack protocol, (b) section of the 3-jack protocol and (c) a deadlock trace in the 3-jack protocol.
Plate A.4 Tracing node paths. An initial state at the top of the visualization was selected and the visualization shows all possible traces from that state. A small number of states (in red) were selected based on their state parameter.
Plate A.5 Behavior of modular jack system consisting of (a) 512 nodes, (b) 2,860 nodes, (c) 15,609 nodes, (d) 70,926 nodes, (e) 284,077 nodes and (f) 1,025,844 nodes.
Plate A.6 Behavior of the link layer of the FireWire protocol, simulated using two links (25,898 nodes). The two large bulks in the center of the visualization indicate two (similar) communication phases. The asymmetries between the branches of the top bulk are unexpected, because they indicate that the behavior of the system differs, depending on which link requests use of the bus first. The dark line indicates a sample execution trace of the system, information like this is useful to ensure that tests cover as much of the state space as possible. Colors indicate cluster depth.
Plate A.7 A number of visualizations of transition systems taken from the VLTS benchmark set.
Plate A.8 Identical matrix cells at different levels of visual abstraction. (a) shows the part of $M_z(P(x), P(y))$ representing $x$ and $y$ and (b) shows $M_z(x, y)$.

Plate A.9 Series of frames showing the transition from the cell in figure 7.2b to the cell in figure 7.3
Plate A.10 Displaying additional data: (a) call permissions; (b) call neighborhood (red calls are closer to call under pointer) and (c) call multiplicity.

Plate A.11 (a) Distribution of skin cancer versus age in the San Francisco - Oakland register, showing a strange peak in San Francisco county (4th row from the top); (b) Distribution of skin cancer versus race in Iowa’s counties and (c) high level view of a spanning tree of a 260 million node graph, detailing phone traffic. The user opened a window that shows a node-link representation of the matrix.
Plate A.12  A screenshot of the improved interface, showing racial distribution for colon cancer cases in Connecticut counties. The overview window is in the top left, hierarchy trees are on the top and left of the matrix, node labels are on the bottom and right.
Plate A.13  Layouts with random initialization: (a) LinLog force model ($P = -381705, M = 1500$) and (b) Our force model ($P = -544602, M = 1500, t_0 = 0.5, t_1 = 0.6, r_{start} = 2$).

Plate A.14  Using spheres to emphasize clusters. The node screen size is kept constant when zooming in such that clusters fall apart in closeups. Colors indicate artist birthdate.
Plate A.15 Comparison with MCL: (a) 25 largest MCL clusters indicated in color. Note the correspondence with 8.2b; (b) Edge length distributions plotted for internal and external cluster edges.

Plate A.16 Comparison with Newman’s fast [125] method: (a) Resulting unbalanced clustering; (b) Dendrogram of agglomeration order. Note the chaining effects.
Plate A.17 Using explicit clustering to abstract the graph: (a) Straight cut, (b) Semantic Fisheye using a cubic DOA function and (c) Semantic Fisheye with a linear DOA function integrated with a geometrical fisheye.

Plate A.18 Series of frames showing the effect of a changing focus on the layout. Surrealist artists are indicated in red, artists belonging to the Pop-Art movement in green.
Plate A.19  Larger samples: (a) Citations between papers published in 12 years of IEEE Visualization conference proceedings, volume visualization in blue, flow visualization in red, terrain visualization/surface simplification in green and information visualization in yellow; (b) Similarity relations between 10,000 pop artists. The visualization uses two foci, one on country (right) and one on reggae (top). Dataset courtesy of Overture/Yahoo Research.
Plate A.20  Comparing the visualizations from the previous four chapters: each row shows four different visualizations of a single dataset. First row shows a 20K node tree, second row shows a 3K node finite automaton, third row shows a 30K node layered software structure and the bottom row shows a 500 node small world graph.
Appendix B

Statistics for beamtree user experiment

The total time for each task is defined as the sum of the answer times for both the small and the large randomized tree. All time measurements were subsequently normalized to the grand mean for that task to remove inter subject variance. We then performed a within-subject ANOVA for each task.

B.1 Estimating the size of a node

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested</td>
<td>12</td>
<td>51.58</td>
<td>14.84</td>
<td>22.93</td>
<td>73.70</td>
</tr>
<tr>
<td>Cushion</td>
<td>12</td>
<td>56.81</td>
<td>20.62</td>
<td>18.84</td>
<td>87.69</td>
</tr>
<tr>
<td>2DBT</td>
<td>12</td>
<td>57.17</td>
<td>18.98</td>
<td>21.53</td>
<td>76.99</td>
</tr>
<tr>
<td>3DBT</td>
<td>12</td>
<td>61.88</td>
<td>16.71</td>
<td>28.33</td>
<td>88.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subjects</td>
<td>8195</td>
<td>11</td>
<td>745</td>
<td>4.17</td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td>638</td>
<td>3</td>
<td>213</td>
<td>1.18</td>
<td>0.33</td>
</tr>
<tr>
<td>Error</td>
<td>5943</td>
<td>33</td>
<td>180</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>14776</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.1 ANOVA results for the size task. No significant effects at \( p < 0.05 \).
B.2 Estimating the depth of a node

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested</td>
<td>12</td>
<td>39.78</td>
<td>17.44</td>
<td>12.78</td>
<td>67.84</td>
</tr>
<tr>
<td>Cushion</td>
<td>12</td>
<td>31.97</td>
<td>14.09</td>
<td>12.47</td>
<td>58.99</td>
</tr>
<tr>
<td>2DBT</td>
<td>12</td>
<td>35.96</td>
<td>21.52</td>
<td>14.90</td>
<td>80.91</td>
</tr>
<tr>
<td>3DBT</td>
<td>12</td>
<td>32.84</td>
<td>13.12</td>
<td>15.84</td>
<td>66.10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subjects</td>
<td>5780</td>
<td>11</td>
<td>525</td>
<td>2.51</td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td>450</td>
<td>3</td>
<td>150</td>
<td>0.74</td>
<td>0.53</td>
</tr>
<tr>
<td>Error</td>
<td>6737</td>
<td>33</td>
<td>204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>12967</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.2 ANOVA results for the depth task. No significant effects at $p < 0.05$.

B.3 Estimating the number of levels in the tree

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested</td>
<td>12</td>
<td>76.44</td>
<td>23.26</td>
<td>43.97</td>
<td>109.78</td>
</tr>
<tr>
<td>Cushion</td>
<td>12</td>
<td>82.61</td>
<td>35.95</td>
<td>35.33</td>
<td>152.79</td>
</tr>
<tr>
<td>2DBT</td>
<td>12</td>
<td>66.20</td>
<td>27.44</td>
<td>29.97</td>
<td>115.32</td>
</tr>
<tr>
<td>3DBT</td>
<td>12</td>
<td>37.50</td>
<td>16.54</td>
<td>22.46</td>
<td>82.79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subjects</td>
<td>12033</td>
<td>11</td>
<td>1094</td>
<td>1.86</td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td>14364</td>
<td>3</td>
<td>4788</td>
<td>8.13</td>
<td>0.00</td>
</tr>
<tr>
<td>Error</td>
<td>19429</td>
<td>33</td>
<td>589</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>45826</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.3 ANOVA descriptives for the levels task. Significant effects at $p < 0.05$. 

B.4 Finding the common ancestor of two nodes in the tree

<table>
<thead>
<tr>
<th></th>
<th>Nested</th>
<th>Cushion</th>
<th>2DBT</th>
<th>3DBT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested</td>
<td>6.17</td>
<td>10.23</td>
<td>38.94</td>
<td>45.115 (*)</td>
</tr>
<tr>
<td>Cushion</td>
<td></td>
<td>16.41</td>
<td>45.115 (*)</td>
<td>28.71 (*)</td>
</tr>
<tr>
<td>2DBT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3DBT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.4 Post Hoc Tukey HSD for the levels task showing absolute differences between means. (*) indicates a difference in means that is significant at the .05 level.

B.4 Finding the common ancestor of two nodes in the tree

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested</td>
<td>12</td>
<td>23.17</td>
<td>12.98</td>
<td>10.23</td>
<td>49.68</td>
</tr>
<tr>
<td>Cushion</td>
<td>12</td>
<td>22.04</td>
<td>11.53</td>
<td>2.73</td>
<td>42.86</td>
</tr>
<tr>
<td>2DBT</td>
<td>12</td>
<td>25.84</td>
<td>14.20</td>
<td>8.95</td>
<td>57.93</td>
</tr>
<tr>
<td>3DBT</td>
<td>12</td>
<td>49.68</td>
<td>37.33</td>
<td>12.15</td>
<td>149.75</td>
</tr>
</tbody>
</table>

Table B.5 ANOVA results for the common task. Significant effects at $p < 0.05$.

<table>
<thead>
<tr>
<th></th>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subjects</td>
<td>8105</td>
<td>11</td>
<td>737</td>
<td>1.90</td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td>6174</td>
<td>3</td>
<td>2058</td>
<td>5.32</td>
<td>0.00</td>
</tr>
<tr>
<td>Error</td>
<td>12760</td>
<td>33</td>
<td>387</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>27039</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.6 Post Hoc Tukey HSD for the common task showing absolute differences between means. (*) indicates a difference in means that is significant at the .05 level.
### B.5 Recalling the position of a previously indicated node

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested</td>
<td>12</td>
<td>27.70</td>
<td>13.20</td>
<td>10.33</td>
<td>51.22</td>
</tr>
<tr>
<td>Cushion</td>
<td>12</td>
<td>31.54</td>
<td>14.37</td>
<td>13.24</td>
<td>65.21</td>
</tr>
<tr>
<td>2DBT</td>
<td>12</td>
<td>36.38</td>
<td>15.74</td>
<td>20.83</td>
<td>76.29</td>
</tr>
<tr>
<td>3DBT</td>
<td>12</td>
<td>34.79</td>
<td>13.76</td>
<td>16.07</td>
<td>70.54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subjects</td>
<td>4692</td>
<td>11</td>
<td>427</td>
<td>3.27</td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td>530</td>
<td>3</td>
<td>177</td>
<td>1.35</td>
<td>0.28</td>
</tr>
<tr>
<td>Error</td>
<td>4305</td>
<td>33</td>
<td>130</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>9527</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.7 ANOVA results for the memory task. No significant effects at $p < 0.05$. 
Samenvatting

In de laatste twee decennia heeft de computer de mens in staat gesteld enorme hoeveelheden informatie te vergaren, te bewaren en te bewerken. Als een direct gevolg van deze virtuele vloedgolf aan informatie is het voor de moderne mens steeds moeilijker om te bepalen welke informatie voor hem relevant is. Dit probleem wordt nog eens verergerd door het feit dat het moeilijk is om het begrip ‘relevante informatie’ vooraf te specificeren. Er is daarom een duidelijke behoefte aan verkennende gereedschappen die het mogelijk maken dergelijke informatie te achterhalen, voordat exact bekend is wat relevant is en wat niet.

Visualisatie (meer specifiek, computer ondersteunde visualisatie) is een onderzoeksgebied dat zich bezig houdt met het gebruiken van computer grafiek om grote hoeveelheden data inzichtelijk te maken. Denk bijvoorbeeld aan gegevens uit windtunnels, computersimulaties of atmosferische metingen.

Dit proefschrift gaat over het inzichtelijk maken van de informatie in abstracte data, zoals bijvoorbeeld de inhoud van databases, grote organisatiestructuren of communicatietoren. Abstracte data heeft geen voor de hand liggende ruimtelijke representatie, wat het zichtbaar maken bemoeilijkt. We beperken ons hier tot abstracte gegevens in de vorm van netwerken (ook wel grafen genoemd). Deze gegevens bestaan uit een verzameling nodes (bijvoorbeeld personen, stukjes software of internetpagina’s) waartussen relaties (bijvoorbeeld ‘is een vriend van’, ‘gebruikt’ of ‘verwijst naar’) zijn gedefinieerd.

Een veelgebruikte manier om een graaf visueel te maken is het node-link diagram, een verzameling symbolen verbonden door lijnen. Het automatisch genereren van dergelijke diagrammen is echter niet eenvoudig, zelfs als de graaf relatief klein is. Grotere grafen (met meer dan een paar duizend nodes) vormen een nog groter probleem en resulterende diagrammen zijn vaak erg moeilijk leesbaar. De onderzoeksvraag die dit proefschrift probeert te beantwoorden is dan ook: hoe kunnen we visualisatie gebruiken om inzicht te krijgen in de structuur en inhoud van grote grafen, met duizenden of zelfs miljoenen nodes?

In Hoofdstuk 2 karakteriseren we het probleem aan de hand van drie hoofdaspecten: data (wat zijn de karakteristieken van de gegevens?), doel (wat wil de gebruiker met deze gegevens?) en context (wat zijn de randvoorwaarden?).

Hoofdstuk 3 beschrijft bestaande technieken die nuttig kunnen zijn bij het ontwerpen van een visualisatie en geeft een overzicht van de huidige toestand van het onderzoeksgebied. In Hoofdstuk 4 selecteren we vier verschillende versies van het visualisatie pro-
bleem op basis van de karakteristieken in Hoofdstuk 2 en construeren daar vervolgens visualisaties voor.

Hoofdstuk 5 beschrijft Beamtrees, een visualisatie methode voor boomstructuren die is gebaseerd op het gebruik van overlapping om relaties weer te geven. Als toepassingsgebied is de bestandsstructuur van een harde schijf gekozen. Een kwantitatieve gebruikerstest is uitgevoerd om onderzoeken of er een betere informatie overdracht plaatsvond.

In Hoofdstuk 6 wordt een techniek beschreven die is ontwikkeld voor het visualiseren van eindige automaten. Een eindige automaat beschrijft het gedrag van een computerproces op een elementair niveau. Bestudering van deze automaten kan leiden tot het ontdekken van fouten in software zonder deze expliciet te testen. Eindige automaten kunnen in grootte variëren van tientallen mogelijke toestanden tot honderden miljoenen toestanden. De ontwikkelde visualisatie kan de globale structuur en symmetrieën weergeven in automaten van een paar miljoen toestanden.

Hoofdstuk 7 is een case-study in het visualiseren van software systeem dat ontwikkeld werd bij Philips Medical Systems. In veel systemen wordt door een groep softwarearchitecten een globaal ontwerp gemaakt, dat daarna door een veel grotere groep programmeurs wordt geïmplementeerd. Tijdens dit implementatieproces wordt vaak (bewust of onbewust) afgeweken van het oorspronkelijke ontwerp, wat invloed kan hebben op de uitkomst. De ontwikkelde visualisatie is gebaseerd op een matrix representatie van een graaf en geeft architecten een overzicht van het ontwikkelde systeem, zowel op architectuur niveau als op code-niveau. Dit prototype is later uitgebreid met mogelijkheden om ook grafen aan te kunnen die zo groot zijn dat ze niet in het geheugen van een computer passen.

Hoofdstuk 8 beschrijft een visualisatie voor grafen waarin elke node vanuit een willekeurige node bereikbaar is in een klein aantal stappen. Deze zogenaamde ‘small world’ graphs komen vaak voor in de praktijk en zijn moeilijk te visualiseren vanwege hun sterk verbonden karakter. De methode gebruikt een nieuw plaatsingsalgoritme om interne structuren zichtbaar te maken en een nieuwe interactiemethode die het mogelijk maakt om tegelijkertijd gedetailleerde informatie en de globale context te zien.

In Hoofdstuk 9 vergelijken we de vier ontwikkelde prototypes. Een eerste conclusie is dat als we een visualisatie schaalbaar willen maken, we vaak karakteristieke eigenschappen van de te visualiseren data gebruiken om deze te vereenvoudigen. Hierdoor is het vaak moeilijk om een visualisatie te hergebruiken voor een ander applicatiegebied.

Aan de hand van de bovenstaande prototypen destilleren we vier kernproblemen die optreden tijdens de ontwikkeling van een visualisatie van een grote graaf. Deze problemen zijn:

- **Representatie:** Hoe kiezen we een geschikte visuele representatie voor een graaf?
- **Abstractie:** Gegeven een grote graaf hoe creëren we een kleinere graaf die de hoofdeigenschappen van de grotere behoudt?
- **Navigatie:** Hoe stellen we een gebruiker in staat een geabstraheerde graaf te verkennen zonder daarbij het overzicht te verliezen?
Mapping: Hoe koppelen we de daadwerkelijke informatie en de abstracte visualisatie van die structuur?

Vervolgens presenteren we een aantal mogelijke oplossingen voor deze problemen en geven een aantal grondregels die van nut zijn bij het ontwikkelen van een graafvisualisatie de novo. In Hoofdstuk 10 tenslotte, presenteren we algemene conclusies en suggesties voor verder onderzoek.
Het proefschrift dat u heeft doorgelezen of simpelweg hebt opengeslagen op deze pagina (na het lezen van de stellingen natuurlijk) is niet alleen het werk van ondergetekende. Een lange lijst van personen heeft in meer of mindere mate invloed gehad op de gedachtenkrankels en uitwerkingen daarvan die u in voorgaande pagina’s terugvindt.

Op professioneel gebied ben ik erg veel dank verschuldigd aan mijn eerste promotor, prof.dr.ir. Jarke (J.) van Wijk. Jack 1, een van de redenen dat ik dit project heb aangenomen was onze uitermate prettige samenwerking. Een proefschrift schrijven wordt een stuk gemakkelijker als je een promotor hebt die in je gelooft en die je vertrouwt, eentje waar je altijd binnen kunt lopen met willekeurige vragen en die je alle vrijheid geeft om jezelf op wetenschappelijk gebied te ontwikkelen. “Promoveren is zwoegen in de donkere kelders van de wetenschap” grapte je wel eens, maar dat heb ik toch zeker niet zo ervaren de afgelopen vier jaren, waarvoor mijn dank.

Eveneens dank aan mijn tweede promotor prof.dr.ir. Jan-Friso Groote, wiens aantrekkelijke enthousiasme een drijvende kracht was achter Hoofdstuk 6 van dit proefschrift.

I would especially like to thank dr. James Abello for taking a seat in my doctorate committee. James, thanks for giving me the opportunity to work in a different environment in a different country and for your advice on common practices in said country. I’ve always enjoyed our cooperation and the interesting discussions (both over the phone and over a beer) that evolved from it and I hope we can continue this way of work in the future. Thanks also go to the other members of my doctorate committee, prof.dr.ir. Erik Jansen, prof.dr.ir. Jean-Bernard Martens, prof.dr.ir. Robert van Liere and prof.dr. Guy Melançon for reserving some of their valuable time to read and/or comment on my thesis.

Ook aan mijn collega’s in de Visualisatie groep en de OAS groep bewaar ik goede herinneringen, al bestaan die herinneringen op de een of andere manier voornamelijk uit totaal niet werk-gerelateerde dingen als vlotten bouwen, pizzas eten, pottenbakken en films maken. Hetgeen waarschijnlijk een indicatie is dat we dat soort dingen vaker zouden moeten doen.

Eveneens ben ik dank aan verschuldigd aan André Postma, Ben Pronk en Ivo Canjels bij Philips Medical Systems voor hun tijd, waardevolle suggesties en andere input die uiteindelijk hebben geleid tot een gedeelte van het werk in Hoofdstuk 7. For the other part of that chapter I have to thank David Millman for his help in the preprocessing of the

---

1[Ja’k]
data and Rutgers DIMACS for giving me the opportunity to work there.

Ook de samenwerking met mijn mede-bestuursleden van AiOOE (inmiddels PromoVE) was erg prettig en bood een interessant kijkje in de bestuurkant van deze universiteit. Bij deze wil ik Edgar, Erik, Mustafa, Dirk-Jan en Koen bedanken voor hun inzet.

Tenslotte, mijn dank aan mijn (ex-)kamergenoten Slava, Hannes, Dennie, Danny en Lucian (in chronologische volgorde) en collega lunchers Aad en Muck voor discussies over computergerelateerde dingen, discussies over de dingen des levens, discussies over hoe Nederland wel of niet bestuurd zou moeten worden en vermakelijke discussies over onzinnige trivialiteiten. Veel succes in jullie verdere carrières, ik houd contact.

Op privé vlak ben ik veel dank verschuldigd aan mijn ouders. Pa en ma, bedankt voor jullie steun en liefde. Jullie hebben me altijd het gevoel gegeven dat de keuzes die ik maak de juiste zijn en hadden vast nooit gedacht dat de aankoop van een Apple IIe in de vroege jaren ’80 hiertoe zou leiden. Eveneens dank aan mijn broer, schoonouders en al mijn vrienden, wie invloed op dit proefschrift relatief gering is, maar zonder wie mijn leven een stuk saaier zou zijn.

En tot slot (en ja schat, dit is echt de erplek) mijn dank aan Maaike, mijn lief, mijn sociaal geweten, mijn spiegel en de trap onder mijn achterste die ik zo af en toe nodig heb. Bedankt voor al je onvoorwaardelijke liefde, steun en geduld. “Sometimes you just have to take the leap and build your wings on the way down” is dan misschien wel op mij van toepassing, maar dat het een duo-sprong zou worden had je waarschijnlijk niet gedacht...

Frank van Ham

september 2005
Curriculum Vitae

Frank van Ham was born on January 21st 1976 in Veldhoven, Noord-Brabant, The Netherlands. After completing his secondary education at the Anton van Duinkerken College, he studied computer science at the department of Mathematics and Computer Science at the Technische Universiteit Eindhoven. He received his Master’s degree (with honors) in 2000. His Master’s thesis was titled “Visualization of State Transition Graphs”.

He was briefly employed as a scientific programmer, working on the SequoiaView prototype for large disk visualization, and in September 2001 started on an NWO funded PhD project entitled “Graph Visualization” at the same university under the supervision of prof.dr.ir. J.J. van Wijk. This thesis contains the results of that project.

As of February 2006, he will be continuing information visualization research at IBM’s T.J. Watson research laboratories in Cambridge, MA, USA.
Publications related to this work

The major part of this thesis is based on the following publications:


