Master's Thesis

Browsing Hierarchical Data with Multi-Level Dynamic Queries and Pruning

by H.P. Kumar
Advisor: B. Shneiderman

CSHCN M.S. 95-3
(ISR M.S. 95-5)

Abstract

Title of Thesis: Browsing Hierarchical Data with Multi-Level Dynamic Queries and Pruning.

Name of degree candidate: Harsha Prem Kumar

Degree and year: Master of Science, 1994

Thesis directed by: Professor Ben Shneiderman
Institute for Systems Research

Users often must browse hierarchies with thousands of nodes in search of those that best match their information needs. The *Tree-browser* visualization tool was specified, designed and developed for this purpose. This tool presents trees in two tightly-coupled views, one a detailed view and the other an overview. Users can use dynamic queries, a method for rapidly filtering data, to filter nodes at each level of the tree. The dynamic query panels are user-customizable. Subtrees of unselected nodes are pruned out, leading to compact views of relevant nodes.

The software architecture, data structures and algorithms used to achieve this behavior are specified. Usability testing of the *Tree-browser*, done with 8 subjects, helped assess strengths and identify possible improvements. The *Tree-browser* was applied to the Network Management (600 nodes) and UniversityFinder (1100 nodes) applications. Future research directions are suggested.

Browsing Hierarchical Data with Multi-Level Dynamic Queries and Pruning.

by

Harsha Prem Kumar

Thesis submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1994

Advisory Committee:

Professor Ben Shneiderman, Chairman/Advisor
Professor Michael Ball
Dr. Catherine Plaisant, Associate Research Scientist

Dedication

To my parents

Acknowledgements

I am thankful to Prof. Ben Shneiderman for being so supportive throughout my stay at the HCIL. I am continuously inspired by his boundless and infectious enthusiasm for work and life in general, and his creative mind that helps us develop our ideas through to completion and then some! One very important skill I have tried to learn from him is to always say what's nice about something before saying what's missing and suggesting improvements.

Dr. Catherine Plaisant has guided and taught me a lot over the last 2 years. Her cheerful manner and sharp mind helped keep our sometimes long and abstract design discussions interesting. She contributed enormously to my thesis work.

Prof. Michael Ball seems to always be able to find loopholes in the "thorough" network management designs that we GRAs put together after days of work! It was a pleasure explaining my *Tree-browser* design to him, and have him provide fresh perspectives and improvements that had never crossed my mind.

Fellow members of the HCIL have contributed significantly to this research through their constructive suggestions. I have learnt a lot from fellow HCIL graduate and undergraduate students. I shall cherish the late nights spent goofing off at the lab... being at the HCIL was a unique experience in that we were all able to do good work and yet have a lot of fun!

My family (mother, father, sister and grand-parents) has always been the main source of my inspiration and encouragement. Given the amount of sacrifices they have made for me, they deserve most of the credit for this work. My friends have always helped me see the brighter side of things and have been there whenever I needed them.

I am extremely grateful to every one of these people.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Scientific Visualization allows scientists to make sense of intellectually large data collections. It does so by exploiting the human perceptual system, using animation and visualization to stimulate cognitive recognition of patterns in the data. A similar set of problems exists in the information world, where the volume of information we deal with expands at an astonishing rate. Information access and management is difficult in large information spaces, because it is hard to visualize what is there and how the various parts are related." [RMC91]

Decisions are an integral part of human life. From deciding what movie to see, to what universities to apply to, one is constantly faced with decisions. A number of decisions in everyday life and in engineering applications require the selection of one or a few elements from many, possibly thousands of elements. In such cases, we humans often find ourselves trying to make a "good" choice by deciding first what we do **not** want, i.e. we first try to reduce the data set to a smaller, more manageable size. After some iterations, it is easier to make the final selection(s) from the reduced data set. This **iterative refinement** of data sets is sometimes known as a hierarchical decision-making process.

A hierarchical data set (HDS) is one in which the data points are organized into a hierarchy. HDSs are common in the real world and in many applications in Engineering and Computer Science. For example, a computer file system consists of either files or directories; a directory can have more directories and files within it. A hierarchy is also known as a tree. A data point in a hierarchical data set is also known as a node. Any node in a tree has a unique parent node (except for the root node of the tree, which has no parent node). A node might have sibling nodes and children nodes. If a node has no children, it is called a leaf node. A HDS can be filtered (queried and reduced) very effectively using a hierarchical-decision making process because the data is inherently hierarchical.

1.1 Motivation

The original motivation for this work was our research on user interfaces for network management (See Section 2.1 for details). While working on a user interface for satellite network configuration, we were faced with the task of choosing one leaf node from a large tree of thousands of nodes. Further, the task was such that the number of interesting leaf nodes could be reduced drastically based on selection criteria at various levels in the tree. This problem prompted us to design and implement the *Tree-browser*. In the remainder of this thesis, *Tree-browser* is used to refer specifically to our design and implementation, while Tree-browser is used to refer generally to tree browsing visualizations.

1.2 Contributions

The major contributions of this work are as follows:

- Extension of dynamic queries to hierarchical data sets: Using the parent-children interrelationship between nodes to facilitate browsing of the HDS,
- Dynamic query environments: A more flexible and powerful mechanism to specify selection subsets of large data sets.

1.3 Content

Chapter 2 presents the original motivation for this work and a review of literature. Chapter 3 describes the design of the *Tree-browser*, its operation, and generalizations to the design. Chapter 4 describes the software system architecture, data structures and algorithms used for tree drawing, dynamic querying and tree pruning. Chapter 5 is about the Usability Testing of the *Tree-browser*, and finally, Chapter 6 summarizes and suggests directions for future work.

Chapter 2

Analysis

This chapter explains the original motivation, specifies *Tree-browser* system requirements and analyzes the previous work done in this field.

2.1 Original Motivation : Network Management Application

The original motivation for this work was our research on user interfaces for network management. We worked closely with **Hughes Network Systems** to design and develop task-oriented user interfaces which provide data overviews and task overviews to the network operator [KPTS94].

While working on a user interface for satellite network configuration, we were faced with choosing one leaf node from a large tree of thousands of nodes. This is basically a **needle-in-a-haystack** problem where the haystack is hierarchically structured! Further, the task was such that the number of interesting leaf nodes could be reduced drastically based on selection criteria at various levels in the tree. This problem prompted us to design and implement the *Tree-browser*. Therefore, let us first look at the satellite network configuration problem in some detail.

2.1.1 Simplified Network Structure

In the Hughes architecture, the satellite network consists of a centralized hub and many (thousands) of remotes. Each remote communicates with the hub via satellite (Figure 2.1). This communication is established by setting up a **session**. Any 2 remotes communicate via the hub.

The hub and each remote consist of a complex hierarchy of hardware and software objects (Figure 2.1). A remote has many DPCs (Data Port Clusters), each DPC has many LIMs (LAN Interface Modules), and each LIM many ports. The hub hardware has a similar but larger containment hierarchy: many network groups, many networks, many DPCs, many LIMs and many ports. A hub typically has thousands of ports.

2.1.2 The Task : To set up a Session

A session is established between a port on the hub and a port on the remote. Without getting into details, (see [KPTS94] for details) suffice it to say that in order to set up a session, the network operator has to make at least two selections from large hierarchical data sets:

1. Select a port from hundreds on a remote.

2. Select a port from thousands on the hub.

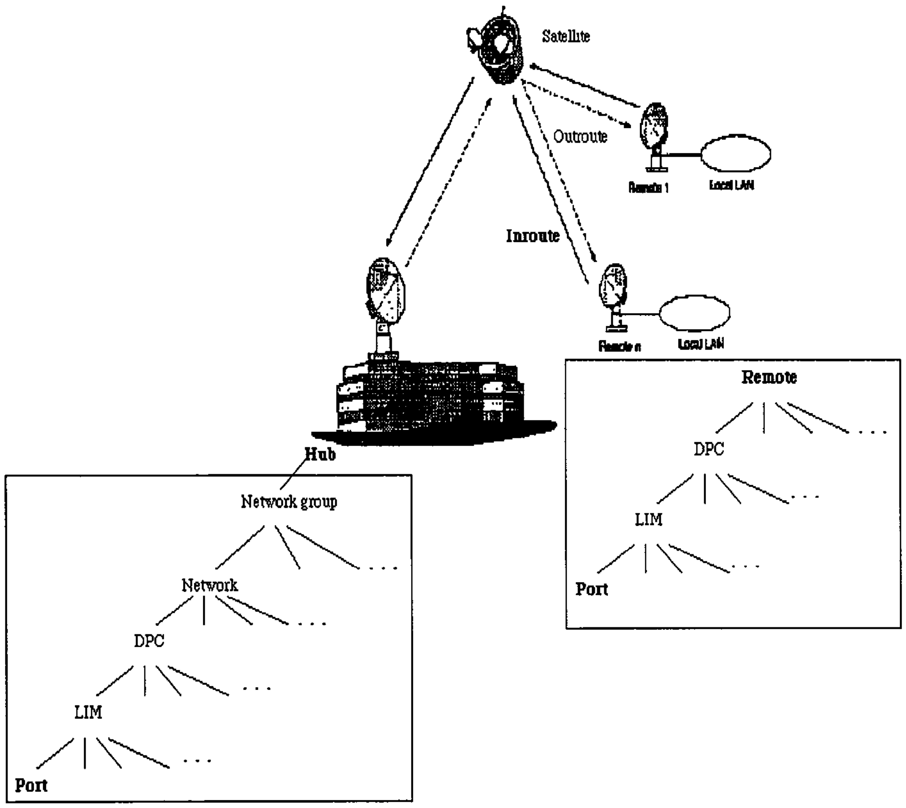


Figure 2.1: Simplified satellite network structure

In both cases, certain choices are better than others. This depends on some port attributes as well as some attributes of the port's corresponding LIM, DPC, etc.

One current network configuration management system that we studied did not address this user requirement at all. Thus, operators made somewhat ad-hoc choices of ports resulting in deterioration in network performance.

The need for a tool to aid in this selection process is clear. The goal is that by making a well-informed decision to begin with, the need for rework and troubleshooting will be reduced and the network will also maintain better performance levels.

Once suitable ports are selected, the operator proceeds to enter a number of configuration parameters needed to set up the session.

2.2 *Tree-browser* System Requirements

The *Tree-browser* needs to meet the following system requirements:

1. **Requirement 1**: Ability to browse the entire tree and view it at different levels.

2. **Requirement 2**: Ability to query nodes at all levels on the basis of attribute values. Querying mechanism should be easy, yet powerful. The interface should make it easy for the operator to specify selection subsets on large trees.

3. **Requirement 3**: Ability to hide uninteresting nodes and branches of nodes and thus reduce the data set progressively. Ability to easily iterate by un-hiding hidden nodes / branches.

The *Tree-browser* design needs to address these basic issues:

1. How will the tree be visualized? i.e. as a 2D node-link diagram, as an outline, as a treemap, or as a cone-tree?

2. How will selection subsets be specified on the original data set?

3. How will the visualization update / respond as the selection criteria are continuously refined? And how will it change as the level at which the tree is visualized is changed?

Figure 2.2 shows our *Tree-browser* applied to the network management application. Details of the system design and implementation are presented in the next 2 chapters.

2.3 Literature Review

Papers relating to user interface design and visualization were surveyed in order to better design the *Tree-browser*. The discussion of literature reviewed is organized as follows:

1. Visualizing Trees: The different visualizations of tree structures that people have used and evaluated, e.g. node-link diagrams, treemaps etc.

2. Automated node-link tree layout: Algorithms to automatically generate node-link layouts for trees.

3. Browsing Trees and Graphs: Strategies to browse trees and graphs. Graph structures are a more general class of data structures than trees. Many interesting browsing issues are common to tree and graph structures.

4. Dynamic Queries: The use of direct manipulation widgets to incrementally, reversibly and continuously control a real-time visual display of the data.

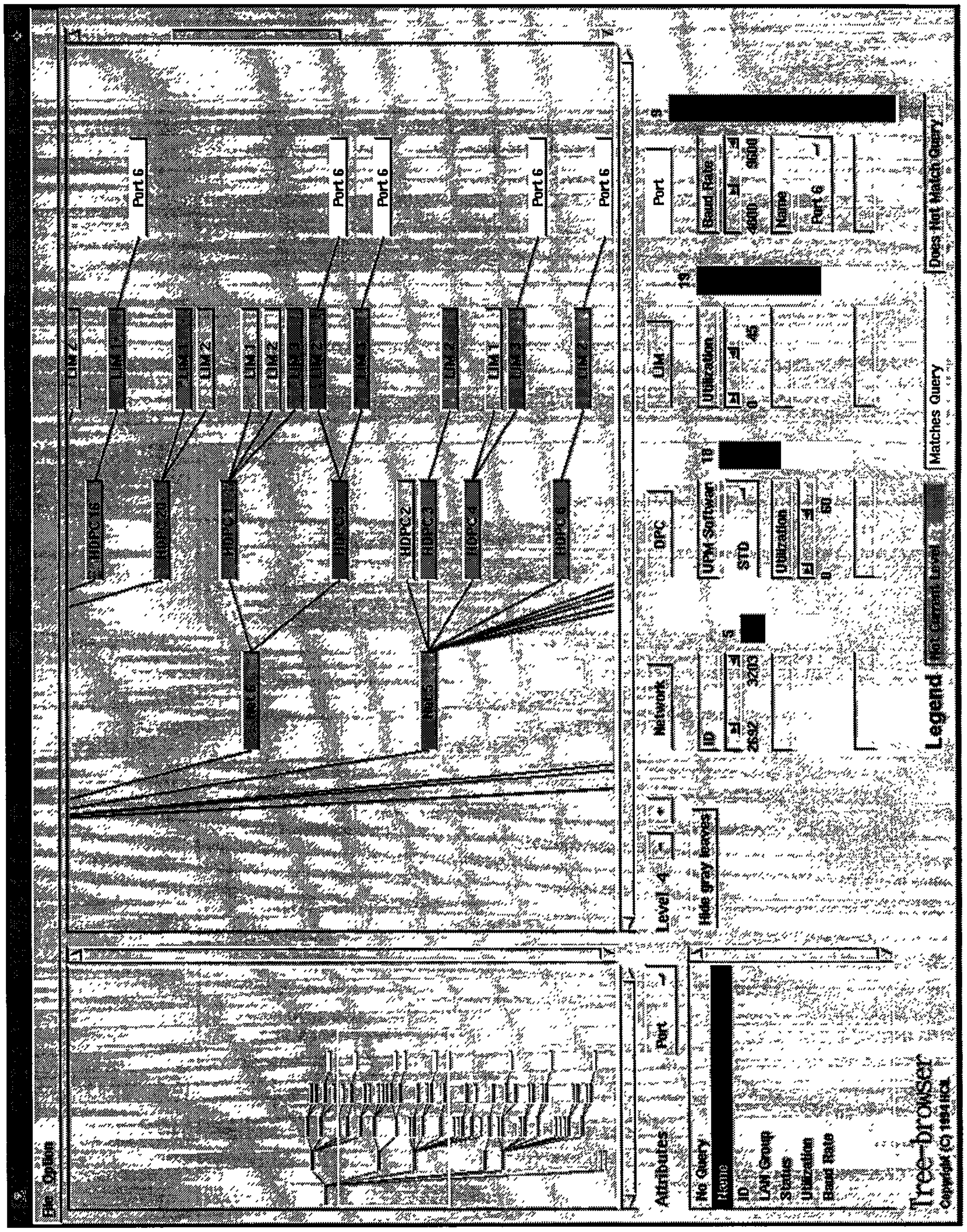


Figure 2.2: The *Tree-browser*

2.3.1 Visualizing Trees

The goal of surveying literature in this area was to make a choice of the visualization method.

The most obvious visualization for any tree structure is the classical node-link diagram (Figure 2.3), which usually uses rectangles to represent nodes in the tree and lines to represent the links (parent-children interrelationship between nodes).

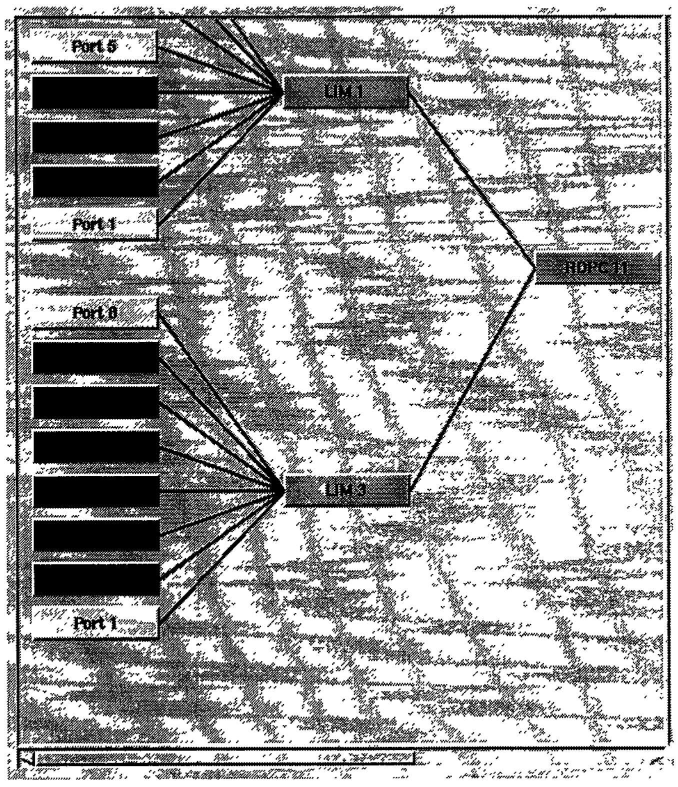


Figure 2.3: Node-link visualization of a tree (Treeview)

We specified and implemented the *Treeview* widget which provides a node-link visualization of any m-ary tree [CJK+94]. Treeviews have traditionally been used for a number of applications, for example:

1. Hierarchy Visualization Applications: Treeviews have been used to visualize hierarchical information such as geographical data, computer file systems, object inheritance hierarchies, and org-charts.

2. Non-Hierarchy Visualization Applications: Treeviews are widely used in the areas of probability and set theory, reliability engineering and systems engineering. Event trees and fault trees are two examples.

The advantages of node-link visualizations of trees are that they are a familiar mapping of structured relationships and therefore easy to understand. They can display attributes of links by color or size if required. However, it is well known that node-link diagrams make inefficient use of screen space, and even trees of medium size need multiple screens to be completely displayed. This necessitates scrolling of the diagram and global context is lost, since only a part of the whole diagram is visible at any given time.

David Beard and John Walker [BI90] used a *map window* - a miniature of the entire information space with a wire-frame box to aid users in remembering their location (Figure 2.4). The map window is better known as an *overview* and the entire information space shown in full size is better known as the *Detailed View* and the wire-frame box is better known as the *field-of-view* or the *panner* [PCS95]. The field-of-view can be dragged around in the overview to pan the detailed view. Similarly, scrolling the detailed view updates the position of the field-of-view in the overview. Hence, the overview and the detailed view are said to be *Tightly-coupled.*

Beard and Walker [BI90] compared three interfaces to browse 2-D spaces: The first provided only scrolling of the detailed view and no overview, the second added an overview and the ability to pan, and the third also added the ability

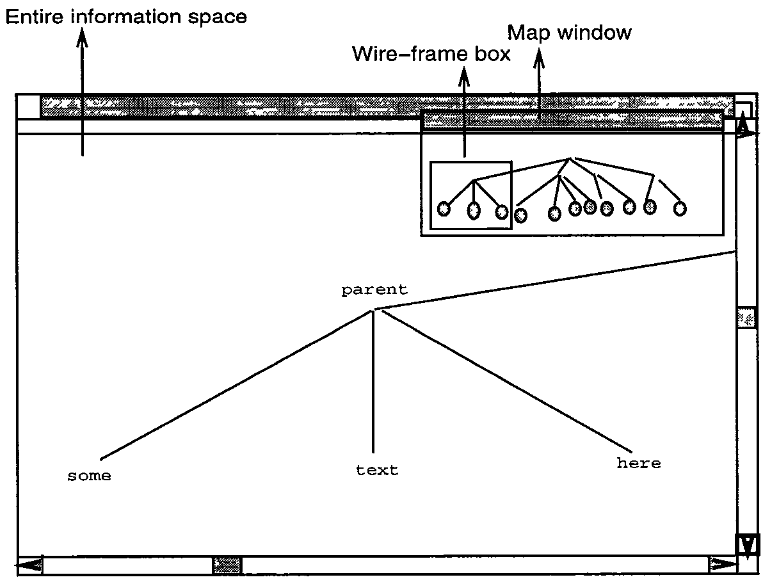


Figure 2.4: The 2D-browser of Beard and Walker

to zoom into parts of the information space. They found that the overview significantly improved user performance. Further, the pan technique and the zoom and pan technique were significantly faster than the scroll technique.

Hence, if the node-link visualization is chosen as the visualization technique for our *Tree-browser*, it would have to provide a tightly-coupled overview and detailed view, in order to provide global context and a more convenient browsing mechanism.

Plaisant et al. [PCH92] found that in some cases, it might even be useful to provide an intermediate view in addition to the overview and the detailed view, when the detail-to-overview zoom ratio is high. An experiment showed that users performance significantly degraded when no intermediate view was provided for a detail-to-overview zoom ratio over 20:1.

The treemap visualization of tree structures uses a two-dimensional (2-d) space-filling approach in which each node is a rectangle whose area is proportional to some attribute such as node size [Shn92]. Shneiderman notes that "even designers of family trees, animal species trees, or organization charts found that a large wall was necessary to give the whole picture", if a node-link visualization is used.

On the other hand, the treemap algorithm utilizes 100% of the designated space (Figure 2.5). Sections of the hierarchy containing more important information can be allocated more display space while portions of the hierarchy which are less important to the specific task at hand can be allocated less space.

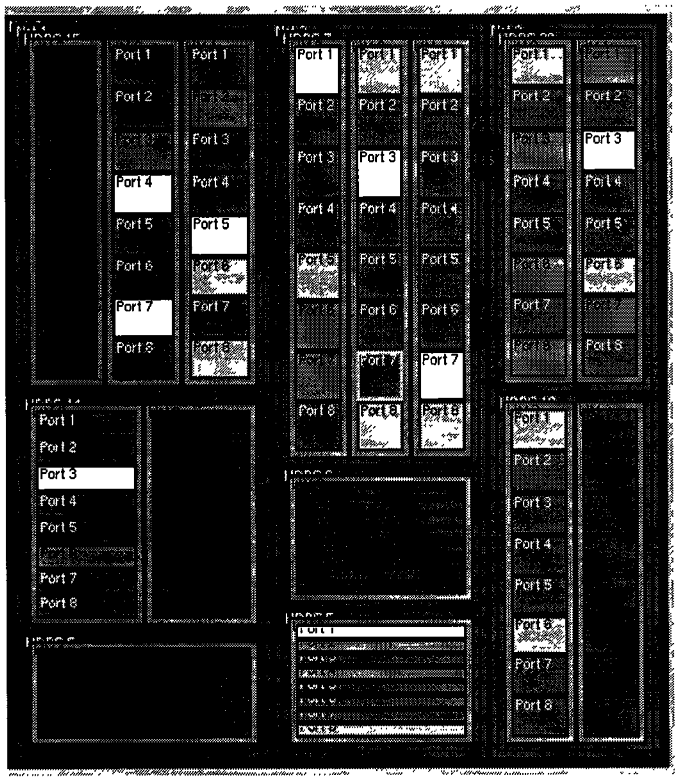


Figure 2.5: Treemap visualization of a tree

Treemaps have been used for file management [JS91], stock portfolio visualization [JT92] and network management [KPTS94].

A similar visualization for hierarchical structures is described by Ritchie [Rit91], in which he describes the Rectangular Structure Chart (RSC) as an abstract diagramming method for drawing hierarchical structures. These charts are also compact, nestable, space-filling mosaics which are strictly rectangular, with no curves, slants, line crossings or tees.

One advantage of the treemap is the fact that leaf nodes get display space proportional to their respective weights. This feature, while being an advantage over traditional node-link diagrams, also leads to the following disadvantages:

- It is assumed that there is at-least one meaningful numerical attribute shared by all nodes at all levels in the hierarchy. Further, it is assumed that the value of this attribute for a parent node is the sum of the values for all its children. This assumption does hold for applications such as a file manager, where the size of each file / directory can be used as the weight in the treemap layout algorithm. However, in many applications like the network management application, this is not a valid assumption. Most node attributes are level-specific; only some are valid at all levels in the hierarchy. The way the treemap gets around this limitation is by allowing each leaf node to occupy unit space.

- Since it is assumed that nodes occupy a display area **strictly** proportional to their weights [JS91], nodes that users are currently not interested in might end up wasting screen space. As specified in Section 2.2, users should be able to filter out nodes that they are not interested in currently. One implementation of the treemap, called the WinSurfer (used for file management in the Windows environment), does allow users to manually hide one node at a time, but a faster and more convenient mechanism for hiding and un-hiding nodes is clearly needed.

Hence, the treemap can be used as the visualization method for the *Tree-browser*, after making some changes to the treemap algorithm.

Robertson et al. [RMC91] and Chignell et al. [CZP93, CGPZ93] used 3D node-link diagrams in order to visualize tree structures. Robertson et al., in the Information Visualizer project at Xerox PARC, developed a tool called Cone-Trees that allows for animation of 3D trees (Figure 2.6). They claim that interactive animation can effectively shift cognitive processing load to the perceptual system. However, while there is considerable literature on search within static 2D displays, there is relatively little prior literature on searches within and manipulation of complex, dynamic 3D displays.

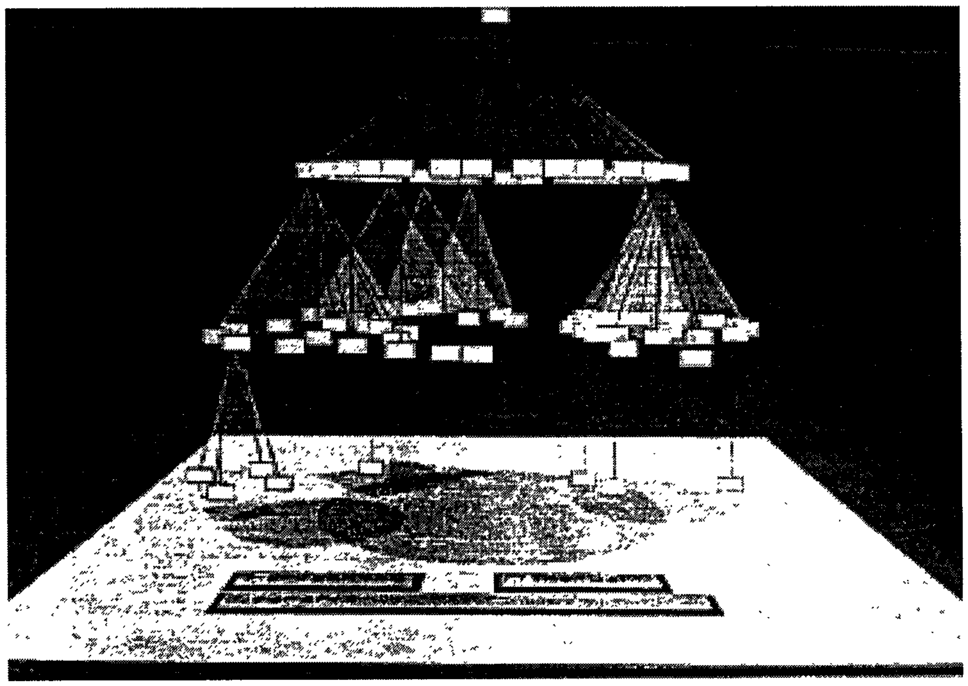


Figure 2.6: Cone-Tree visualization of a tree

Chignell et al. developed the Info-TV tool. Spatial models of information capitalize on the fact that people are very familiar with tree-dimensional spaces and with the visual cues associated with them [CZP93]. The visual scene of a 3D world is supposed to be a more "natural", "ecological", or "compatible" representation than that provided by 2D displays. However, while people are highly skilled at viewing a 3D world, they may require some training to navigate effectively in three dimensions.

Info-TV allows two styles of pruning:

1. The sub-branch(es) for which the chosen node is the root can be removed from the screen

2. The nodes and labels are removed, but the links remain

The authors however, do not describe at all how the nodes whose sub-trees are to be pruned are specified by the user. It is assumed that the user makes these specifications manually by selecting these nodes. This kind of specification is too slow and cumbersome to match our system requirements. Clearly, a better selection subset specification mechanism is needed.

Similarly, Robertson et al. [RMC91] describe "gardening operations" where the user can manually prune and grow the view of the tree (Figure 2.6). Prune and grow operations are done either by menu or by gestures directed at a node. While these gardening operations do help in managing and understanding large, complex hierarchies, this manual mechanism of pruning or growing one subtree at a time, is clearly not sufficient nor very effective in specifying complex searches spanning several levels of the tree. Cone Trees do provide for searches like wild-card string searches, but the updates are not dynamic. In fact, when a search starts, all nodes are made invisible.

Ware and Franck [WF94] tried to quantitatively assess the worth of presenting abstract data in 3D. They report that "true" 3D viewing can increase the size of the graph that can be understood by a factor of 3; using stereo or head-coupling alone produced less advantages.

3D trees were rejected as the visualization method for our *Tree-browser* implementation because of the need for high-end workstations to support 3D animated visualizations.

A 2D node-link diagram was chosen as the visualization method for this implementation of the *Tree-browser*. It is worth mentioning however, that a treemap, a Cone-Tree, or any other tree visualization could also have been used.

2.3.2 Automated node-link tree layout

Researchers have proposed guidelines that lead to pretty, easy to read, aesthetically appealing trees. Wetherell and Shannon [WS79] and later Reingold and Tilford [RT81] formalized notions of tree drawing, and present algorithms based on these. Tilford extended the aesthetics and algorithms to handle m-ary trees. Radack and Desai [RD88] claim to have improved upon Tilford's algorithm.

Charles Wetherell and Alfred Shannon, and Edward Reingold and John Tilford defined five rules to draw an aesthetic binary tree [Moe90]:

1. A parent should be drawn above its children.

2. Nodes at the same level should lie along a horizontal line.

3. A left child should be positioned to the left of its parent and a right child to the right.

4. A tree and its mirror image should be drawn to reflect each other.

5. A subtree should look the same, regardless of where it occurs in the tree.

Moen argues that these rules, while useful, were meant only for binary trees, and general trees are a far more useful class of data structures to visualize. Also, it made more sense to use horizontal trees to diagram user-interface trees because typically, the nodes contained text strings of varying length. The author describes algorithms to draw dynamic trees. His algorithm tries to overcome two drawbacks in traditional tree-drawing algorithms:

1. Tree Nodes of varying shapes and sizes are not supported, and if they are, the overall layout is not compact enough.

2. Layout information is not reused when the trees change, so after every change the layout must be recomputed and the tree redrawn.

Moen's algorithm draws compact trees with nodes of any polygonal shape and the data structure supports insertion and deletion operations on subtrees.

However, the example that Moen describes as a "large" tree is 9 levels deep but only 43 nodes big. In the applications we encountered, that would be considered a very small tree!

2.3.3 Browsing Trees and Graphs

Visualizations have been used traditionally to browse trees and various types of graph structures in different application domains. Examples of graph visualizations include hypertext graphs, finite-state diagrams, flow-charts, parse-trees, pert-charts, dataflow diagrams, wiring diagrams and hierarchical decompositions.

David Gedye puts it very aptly when he says [Ged88]:

> "Browsing is the loosely structured exploration of connected data. It differs from database querying in that exact questions need not be framed; the retrieval of information proceeds incrementally, and by association. In electronic databases, as in written material, browsing is most appropriate when the data being examined contains many explicit relationships, or cross-references. When information is structured as objects and relationships, it often makes sense to browse it graphically, as a two-dimensional picture on the screen. If the data is presented as a graph, with nodes representing the objects, and edges the relationships, the spatial arrangement of the picture aids understanding of its content."

Chimera et al. [CS94] evaluated three interfaces to browse a medium (174) and a large (1296) online hierarchical table of contents. The three interfaces were as follows:

1. A fully expanded stable interface.
2. An expand/contract interface.
3. A multi-pane interface.

Results showed that the stable interface produced the slowest performance for timed tasks as compared to the other two interfaces for the large table of contents. The authors conclude that the expand/contract and multi-pane interfaces offer advantages over the stable interface and that the expand/contract interface's animation characteristics are very important to users' performance and comprehension.

Gedye [Ged88] describes an implementation of a browser for an engineering design database and cautiously attempts to generalize the implementation. The author claims that the research is directly applicable to any organizational structure which is graphical, big, well-connected (N nodes have at least O(N) edges) and typed (there are different types of relationships between the objects).

The proposed solution is to present the user with a list of all the objects. The user can then choose an object, and then upon choosing one particular relationship, a DAG / tree or an equivalence set is created in a new window. So, the tangled web that would have resulted by showing all the relationships of the object together is eliminated by using one window per relationship. The collection of windows for each object is called a context group. If there are more then one context groups up on the screen at any given time, a unique window background color identifies the context group.

When the size of the graph becomes very large, Gedye mentions two existing techniques to reduce the amount of data on the screen to a manageable level:

Zooming: The user is given control of a scale parameter, and depending on the value chosen, either a small amount of the graph is seen in detail, or a large amount is seen in overview. Black box representation is often used to reduce the complexity.

Panning: When the view is zoomed-in, and only a portion of the graph is being displayed, the user is given control of which portion must be displayed.

The authors noted that both zooming and panning work well for graphs that are almost trees, but are inadequate for arbitrary graph structures. Therefore, they implemented pruning in order to assist in the browsing. A subgraph is selected that contains few enough nodes to comfortably fit in the window, and this subgraph is laid out and displayed in its entirety. The subgraph obtained by the pruning procedure is called the "display graph". Rectangular pruning and hourglass pruning algorithms have been described.

Gedye lists the extensions possible to this work. Among them, attribute-based browsing, in which the user does not know the name of the component being browsed, but can easily specify it by a combination of attributes. This feature is similar to one of our *Tree-browser* requirements (Section 2.2).

Several papers deal with the aesthetics of drawing trees and graph diagrams. Ding and Mateti [DM90] identify the following as features of "good" diagramming: visual complexity, regularity, symmetry, consistency, modularity, sizes, shapes, separation, and traditional ways of drawing. This work focuses on identifying features, on a high-level, that make for "nice" data structure diagrams.

The reason there is so much emphasis on automated diagramming algorithms is well-captured by Sugiyama [STT81]: "An automatic method for the drawing of hierarchies by a computer is indispensable because many hierarchies... should be drawn one after another without much lapse of time". The authors give algorithms which take as input, a set of directed pairwise relations among elements of a system and output a readable map. Readability comprises such elements as hierarchical layout, less crossings of edges, straightness of lines, close layout of vertices connected to each other and balanced layout of lines coming into or going out from a vertex.

There are several other works that relate to aesthetic layouts of graph structures [BETT89, HH91, Hen92, MRH91, Mas92, SM91]. According to Henry and Tyson [HH91, Hen92],

"Traditional graph layout algorithms have tried to increase the readability of graphs by focusing on the task of minimizing specific fixed properties. The following list contains some common graph aspects traditional algorithms focus upon:

1. Maximize display symmetry.
2. Avoid edge crossings.
3. Avoid bends in edges.
4. Keep edge lengths uniform.
5. Distribute vertices uniformly.

Unfortunately, minimizing these kinds of aspects tends to be a very difficult computational problem. In fact, finding a layout which simply minimizes edge crossings in a general graph is NP-complete. Approximate solutions can be found by heuristics that run in polynomial time."

According to Henry and Hudson, even when traditional layout algorithms do a good job minimizing such properties, they do not produce the most readable graphs or make the best use of available resources such as screen space, the reason being that the "best layout" depends on the user's current region of interest.

Consequently, a single canonical layout algorithm cannot always produce the best results. Thus, the **ability of users to customize** the layout to meet their current needs and interests is essential. Therefore, users must be provided with **interactive tools** to **iteratively** dissect large graphs into manageable pieces.

[HH91, Hen92] suggested three novel concepts relating to graph visualization. The first two deal with graph layout while the third is a highly interactive mechanism for selecting portions of the graph that match the user's current focus.

The authors suggest that users should be able to create lots of views so that they are able to home in on the aspects of the graph they are currently interested in. While such flexibility, in general, is good, it is clear that integration of multiple views in the minds of users is a non-trivial task, and may carry significant performance overheads with it.

In their system, users can make multiple views of subgraphs. Each view shows one “selection set”. Multiple selection sets are supported, but only one of them is active at any given time. The system provides users with a direct manipulation dialogue box to customize the views by setting parameters like spacing using sliders. Users could create a new view by dragging each node or by copying selected nodes.

In [HH91, Hen92], the authors say that there are two basic types of selection, manual selection in which users select nodes and edges using simple direct manipulation techniques, and algorithmic selection in which users apply an algorithm to the graph. But by classifying manual selection into **only** individual selection or marquee selection, the authors are greatly restricting the range of interesting selection subsets that can be specified by the user.

[HH91] concludes by identifying two primary future directions: firstly, using domain specific graph semantics to guide the layout and the selection, and secondly, creating a methodology that can be used to build **an interactive system for non-programmers to specify selection and layout algorithms**. The analysis of *Tree-browser* requirements has also lead us to the similar problem of a methodology for users to specify selection criteria on data structure diagrams.

Sarkar and Brown [SB92] describe their extensions to fisheye views of graphs, a strategy first proposed by Furnas [Fur86]. A fisheye view of a graph shows an area of interest quite large and with detail, and shows the remainder of the graph successively smaller and in less detail. Since the entire graph structure visualization fits onto the available display space, there is no need to scroll and pan and mentally integrate tightly-coupled views, and global context is not lost. On the other hand, the "map view" strategy [BI90] suffers from the extra space required for the overview, and from forcing the viewer to mentally integrate detail and context.

Schaffer et al. list the many applications where effective graph visualization is not only important, but life-critical!

> "Operators often deal with hierarchically clustered networks such as power grids, machine plants, telephone systems, and gas pipelines. Operators must monitor the network operation. When something goes wrong with the network operation, alarms are sounded. Operators must then quickly isolate and repair problems: these are sometimes due to isolated failures of network components, or they could result from an interrelated breakdown of many components. Failure of these systems could affect large numbers of people, use expensive resources, and even be life-critical (e.g. a nuclear power plant operation). The operator must be able to navigate through these structures quickly and accurately."

Schaffer et al. [SZB$^+$92] take the "Hierarchical clustering" approach to aid the navigation of graph structures. This approach provides a framework for viewing the network at different levels of detail by superimposing a hierarchy on top of it. Nodes are grouped into clusters, and clusters are themselves placed into other clusters. Users can then navigate these clusters until an appropriate level of detail is reached. Links between clusters indicate that the clusters are connected by at least one path in their respective sub-nets.

The authors describe a controlled experiment which compared the performance of subjects navigating and repairing a simulated telephone network, represented as a hierarchically-clustered graph. Subjects used the full-zoom and fisheye visualizations to navigate the clusters. The results showed that subjects using fisheye views were more efficient at performing the task than when they used the full zoom technique. In particular, they took less time to complete the task, and the amount of navigation (indicated by the number of "zoom" actions) was reduced. From the comments of the subjects, most subjects greatly preferred the extra context provided by the fisheye view. But a few subjects expressed a preference for the simplicity of full-zoom views, because the amount of information presented on the screen was always small.

Hollands et al. [HCMM89], however found users getting somewhat disoriented while using fisheye views for complex tasks. The authors compared a fisheye view with a scrolling view for presenting and browsing a graphical network. Subjects were asked to select optimal routes between stations on a fictional subway network, using both interfaces. Performance with the fisheye view was superior when the destination station was not visible in the initial display; where as performance with the scrolling view was superior when both stations were visible and when more complex itineraries were required.

Plaisant et al. [PCS95] provide perhaps the most complete discussion of image browsers including taxonomy and guidelines. This work attempts to standardize some of the terms being used by researchers today, e.g. Detail View, Field-of-view etc. Different kinds of browsers such as "Detail only", "One window with zoom and replace", and "Tiled multilevel browser" are shown. The authors identify 5 classes of tasks that are accomplished with image browsers, e.g. Open ended exploration, navigation, monitoring etc.

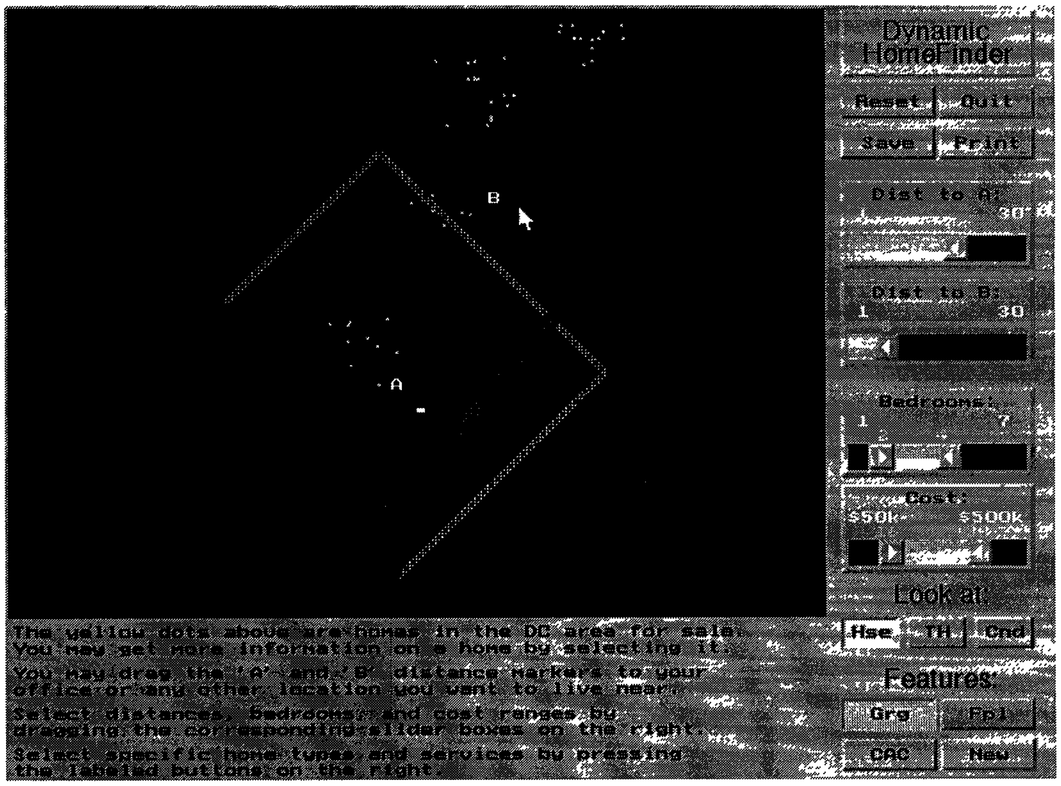
So far, the review of the literature pertaining to tree and graph browsers has not provided a good answer to the problem of specifying selection subsets on large connected data sets like trees and graphs. The specification interface needs to be powerful, flexible and easy-to-use.

2.3.4 Dynamic Queries

This section examines the concept of *Dynamic Queries*, which is a powerful way of specifying selection subsets. According to Shneiderman [Shn94],

> "Dynamic Queries describes the interactive user control of visual query parameters that generates a rapid (100 msec update) animated visual display of database search results. Dynamic queries are an application of the direct manipulation principles in the database environment. They depend on presenting a visual overview, powerful filtering tools, continuous visual display of information, pointing rather than typing, and rapid, incremental, and reversible control of the query."

[Shn94, WS92] describe a system (the HomeFinder - Figure 2.7) for real estate brokers and their clients that allowed them to locate homes by adjusting sliders for the price, number of bedrooms, distance from work, and buttons for home type (house, townhouse, or condominium) etc. Each of the 1100 homes appeared as a point of light on a Washington, DC map. Users could explore the database to find neighborhoods with high or low prices by moving a slider and watching where the points of light appeared. A point of light could be selected to generate a detailed description of the corresponding house, at the bottom of the screen.



The yellow dots above are homes in the DC area for sale.
You may get more information on a home by selecting it.
You may drag the 'A' and 'B' distance markers to your office or any other location you want to live near.
Select distances, bedrooms, and cost ranges by dragging the corresponding slider boxes on the right.
Select specific home types and services by pressing the labeled buttons on the right.

Figure 2.7: The HomeFinder

Dynamic queries have also been applied to explore (databases of) movies, chemical tables of elements, and health statistics [AS94, JS94, Pla93, AWS92].

The advantage of dynamic queries comes from the fact that users can rapidly, safely and even playfully explore a database. The rapid reformulation of queries encourages an atmosphere of exploration. Dynamic Queries achieve these advantages by applying direct manipulation strategies:

- Visual presentation of query components.

- Visual presentation of results.
- Rapid, incremental and reversible actions.
- Selection by pointing (not typing)
- Immediate and continuous feedback

This kind of real-time display enables users to catch trends in the data and spot exceptions. Also, irrelevant information is filtered out. These concepts of dynamic querying and tight-coupling are similar to those of Focusing and Linking [BMMS91].

However, Shneiderman says: "Application specific programming is needed to take the best advantage of dynamic query methods. While we have developed some standardized tools, they still require some conversion of data and possibly some programming". It would be nice to have a flexible and general interface that has no application-specific details coded into the interface.

Chapter 3

Interface Design

In Chapter 2 the *Tree-browser* system requirements were specified. Further, a review of the literature showed that existing tools did not match all of the requirements. Hence, the new *Tree-browser* was designed to meet these requirements:

1. **Requirement 1**: Ability to browse the entire tree and view it at different levels.

2. **Requirement 2**: Ability to query nodes at all levels on the basis of attribute values. Querying mechanism should be easy, yet powerful. The interface should make it easy for the operator to specify selection subsets on large trees.

3. **Requirement 3**: Ability to hide uninteresting nodes and branches of nodes and thus reduce the data set progressively. Ability to easily iterate by un-hiding hidden nodes / branches.

3.1 Theory

Two major concepts that guided the design of the *Tree-browser* both relate to dynamic queries:

- Dynamic Queries on HDSs: This relates to how predefined interrelationships between data points in a HDS influence the querying.
- Dynamic Query Environments: This relates to how the dynamic query control panel could be made user-customizable, in order to accommodate for varying user interests.

This work focussed on the above-mentioned issues, and some other issues, while important, received considerably less attention. For example, we implemented an algorithm to automatically layout trees as node-link diagrams (See Section 4.2 for details), but did not try to optimize the layout in terms of compactness etc.

3.1.1 Dynamic Queries on Hierarchical Data Sets

Dynamic queries have been applied to data sets consisting of independent data points [WS92, AWS92, Pla93, AS94, JS94, Shn94]. In such cases, whether a data point satisfies a given query or not does not in any way affect the outcome for other data points. This is because there are no interrelationships between the data points. Thus, the data set can be thought of as "flat". Queries are merely queries on the attributes of individual data points. For example, in the FilmFinder, each movie is an independent data point. Similarly, in the HomeFinder, each house is an independent data point (Figure 2.7).

In a "non-flat" data set, on the other hand, there are predefined interrelationships between data points. For example, in a hierarchical data set, some nodes are related to some others by the parent-child relationship. Therefore, whether a node matches a given query or not might affect some other nodes. Specifically, while searching a hierarchical data set in a top-down manner (i.e. parent first), it makes sense to prune out all descendant nodes of nodes that do not match the query. For example, if users are looking for ports on low-utilized LIMs, it makes sense to not even consider those ports whose LIMs are over-utilized.

When criteria (like low LIM utilization, low error rate, high data rate, etc...) exist at all or most levels in the hierarchy, **stepwise refinement of the query** can be done to progressively reduce the initial (large) data set into a smaller set, from which good choices may be made.

Dynamic queries applied to stepwise refinement of queries on hierarchical data sets is a step forward from traditional approaches used in many existing file managers / outliners. These systems allow users to explode / expand only one node at a time to reveal the subtree below it. On the other hand, by using dynamic queries, the user can find a suitable set of (more than one) nodes to be exploded / expanded.

3.1.2 Dynamic Query Environments

The HomeFinder and the FilmFinder are examples of systems that provide hard-coded graphical widgets for the user to manipulate in order to dynamically update the visual display. If the user wanted to find homes that were within 2 miles of any hospital, the current HomeFinder interface would have to be reprogrammed (Figure 2.7).

Therefore, the *Dynamic Query Environment* should allow users to customize dynamic query control panels based on current interests. Users should be able to select what combination of attributes they wish to query on, and have the appropriate widgets created, at run-time. The method of selecting the attributes and creating widgets should be easy, and also allow for modifications / backtracking.

3.2 Description of features and operation

A number of interface design issues were considered and resolved during the course of the design. This section describes the features and operation of the *Tree-browser*. Section 3.5 looks at possible design alternatives.

The *Tree-browser* interface consists of two main parts as shown in Figure 2.2:

1. **Data Display**: The tree structure is visualized in two tightly-coupled views. The detailed view on the right shows the node-link diagram fully zoomed-in and the node names displayed, while the overview on the left shows a miniature version of the node-link diagram without the node names. A field-of-view (the yellow rectangle) inside the overview indicates which part of the tree is seen in the detailed view, and can be moved to pan the detailed view. Similarly, when the detailed view is scrolled by using the scrollbars, the field-of-view moves inside the overview to provide global context feedback. If the *Tree-browser* window is resized by the user, the field-of-view shape and size is updated automatically.

2. **Dynamic Query Panel**: This panel consists of two parts, the Attributes List on the left and the Widgets Panel on the right. Initially, there are no query widgets in the Widgets Panel. Users select (by dragging-and-

dropping) up to three attributes from the Attributes List for each level in the hierarchy (except the root level). This causes an appropriate widget (range-slider for numerical attributes and Menu for textual attributes) to be created and initialized (Figure 3.1). Queries (on up to 3 attributes) at each level are AND-ed together. Users can replace an existing widget with another by dropping a new attribute name over the original widget's attribute name. Existing widgets can be deleted by dropping "No Query" onto the corresponding attribute name.

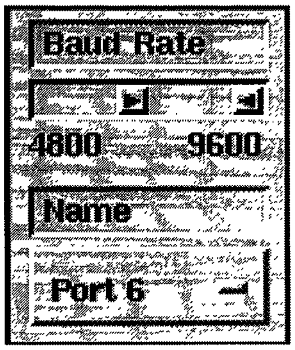


Figure 3.1: Range-slider and Menu widgets

If users manipulate a widget at the current lowest level displayed, the nodes at that level are colored yellow if they match the query, and gray otherwise. These updates of the data display are real-time (within 100 msec of updates to the control widgets) in accordance with the principle of dynamic queries. Buffering was done in order to make the updates as smooth and flicker-free as possible.

If users manipulate a widget at a level other than the current lowest level, the tree visualization first "jumps" to that level, i.e. the level of the widget is made the tree's current lowest level. This is done so that the structure of the tree during direct manipulation of the widgets remains constant and only the colors of nodes change. Then the nodes at the new lowest level are updated (by coloring yellow or gray) in real-time to show whether they match the query or not.

The tree structure changes **only** when the current lowest level of the tree is changed, which can be accomplished by users in any 1 or 3 ways:

1. By clicking on the corresponding level button (i.e. the buttons labeled "Network", "DPC", "LIM" and "Port") just below the data display (Figure 2.2).

2. By manipulating a widget at any level other than the current lowest level, as explained above.

3. By clicking on the + or - buttons to either increase or decrease the levels displayed.

When the current lowest level is changed so as to show more levels, pruning of the tree is done so as to eliminate subtrees of nodes that do not match the query at their own levels. For example, if the user manipulates the range slider for Network ID such that Nets 1 and 2 do not match the query (Figure 3.2), and then increases the lowest level displayed by 1, the children of nets 1 and 2 are not shown, while children of other nets are shown (Figure 3.3). Nets 1 and 2 now appear in gray, while all the other nets appear in orange, simply to show that those nodes did match the query at their own level. This feedback enables users to go back and change queries at higher levels, and thus iteratively refine the selection subset.

As explained above, nodes at the current lowest level are colored either yellow or gray, while nodes at higher levels are colored either orange or gray. Also, the

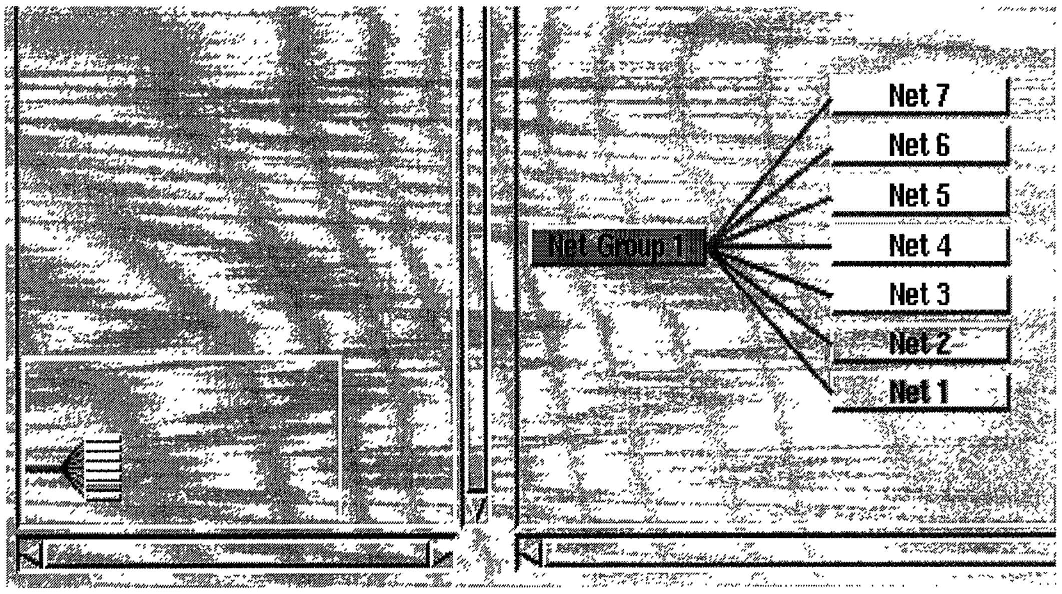


Figure 3.2: Illustrating Pruning - Nets 1 and 2 do not match query

level button corresponding to the current lowest level is colored yellow so as to focus the attention of the user to that level. Buttons at other levels are colored gray.

When the current lowest level is changed so as to show fewer levels, the tree is simply "folded" back to the new lowest level, and then the nodes at that level are colored either yellow or gray based on the query at that level.

Thus, users can easily jump back and forth between levels in order to fine tune their search.

The Attributes List on the left shows only the attributes corresponding to the current lowest level. Users can however access names of attributes at other levels by choosing the appropriate level name (e.g. Network, DPC, etc.) from the attributes menu just above the Attributes List (Figure 2.2).

There are four feedback indicators, one corresponding to each level (other than the root level), that are updated in order to show the number and pro-

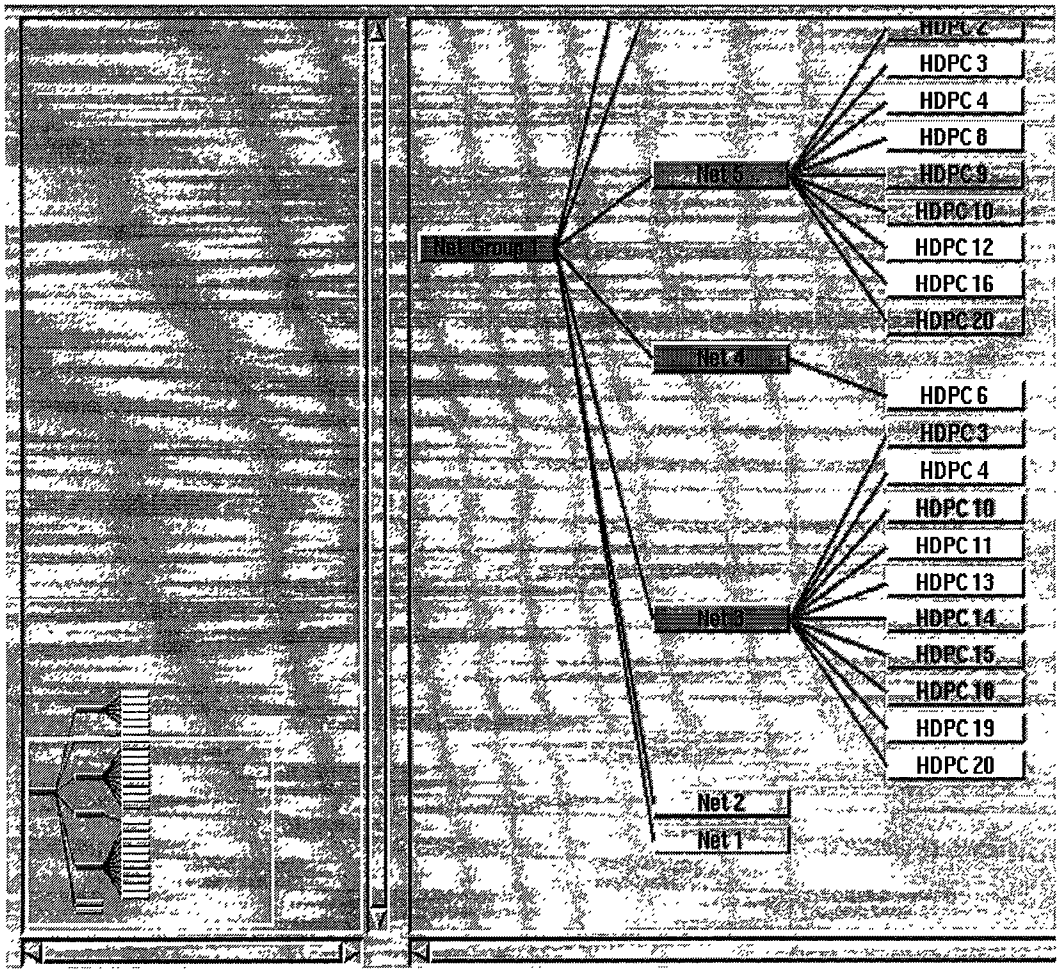


Figure 3.3: Illustrating Pruning - Subtrees of Nets 1 and 2 are not shown

portion of nodes that currently match the query (i.e. number of "hits"). The proportion of the feedback indicator that is colored yellow corresponds to the proportion of hits. This proportion is a proportion of the total number of nodes nodes at that level, not of the number of nodes at that level currently displayed. The actual number of hits is also displayed at the top of each feedback indicator. Proportion indicators are displayed only from level 1 down till the current lowest level, those for deeper levels are hidden (grayed out). This is because calculating the number of nodes that **would** match the query at deeper levels is computationally intensive and would slow down the dynamic queries on nodes at the current level.

To summarize, the *Tree-browser* is a visualization tool for hierarchical data that makes use of dynamic queries and pruning. The *Tree-browser* uses two coordinated or tightly-coupled treeviews of the same tree, one a detailed view and the other an overview. The user can select up to 3 attributes (numerical or textual) for querying nodes at each level in the hierarchy. Subtrees of nodes that do not match the query at their own level are pruned out of the visualization. Thus, one can reduce a large data set (with thousands of nodes) to a much smaller set, from which good selections can be made.

3.3 Example Applications

This section describes typical scenarios in two applications and steps through sequences of actions that users might execute in order to solve the problem at hand.

3.3.1 Network Management

Let us assume that the network operator is interested in finding the "best" 3 ports on the hub that match certain criteria. The criteria are as follows:

1. The port should have a baud rate between 4800 and 9600 AND should be the 6th port on its LIM (i.e. the Name of the port should be "Port 6").

2. The corresponding LIM should not be over-utilized, i.e. Utilization should be no more than 45%.

3. The corresponding DPC should be running "STD" UPM software AND its utilization should be no more than 60%.

4. Finally, only networks with IDs between 2692 and 3142 should be considered (because the customer owns only these networks).

The *Tree-browser* initially shows the first two levels of the tree. The root node (Hub) is initialized to orange color while the first level nodes (Networks) are initialized to yellow color. The tree is displayed from left-to-right. The tree is medium-sized, with about 600 nodes, 488 of which are ports (leaf nodes). 24 screen-fulls are needed to show the entire tree in the detailed view, while 5 screens are required to even show the miniature version in the overview. In our *Tree-browser* design, the overview was always to show the entire tree in miniature so that no scrolling of the overview itself is needed. This, however, was not enforced in our implementation. Since the overview itself does not fit within one screen-full, the benefits of having the overview are reduced, but this is still better than having no overview at all.

The operator finds the attribute "ID" in the Attributes List, and drags-and-drops that onto one of the 3 empty text items under the Network button. A range slider is created. The range slider is automatically initialized to the range of IDs of all networks in the database. The user sets the lower thumb to 2692 and the upper thumb to 3142. Then, the user clicks on the DPC button.

This causes the tree to grow to the DPC level. Only networks with IDs between 2692 and 3142 are exploded to the DPC level. Users apply a similar drag-and-drop followed by query strategy at the DPC and LIM levels. At the DPC level, 2 drags-and-drops are required, since a conjunction is required.

At the port level, the operator first performs the query on baud rate and then on Name. The number of ports satisfying all the above criteria is about 10, but a lot of screen space is taken by uninteresting gray ports (about 1.5 screen-fulls of the overview as shown in Figure 3.4). The operator clicks on the "Hide Gray Leaves" button and the gray ports (leaves) are totally removed from the data display (Figure 2.2).

Now, the entire remaining tree fits in about half the overview and in 3 screen-fulls of the detailed view! Hiding the gray leaves has made the visualization much more compact. But there are 9 ports that match the criteria. So, the operator decides that the LIM utilization requirement could be further improved. The upper thumb of the LIM utilization slider is moved from 45% to 29%.

Once again, the operator clicks on the "Port" and "Hide Gray Leaves" buttons. Now, there are only 5 ports that match the query. The operator proceeds to get detailed information about each of these ports by clicking on each and examining the details as they pop-up in a separate dialog box.

The best 3 ports are selected after some review of the detailed information of each of the 5 "finalists".

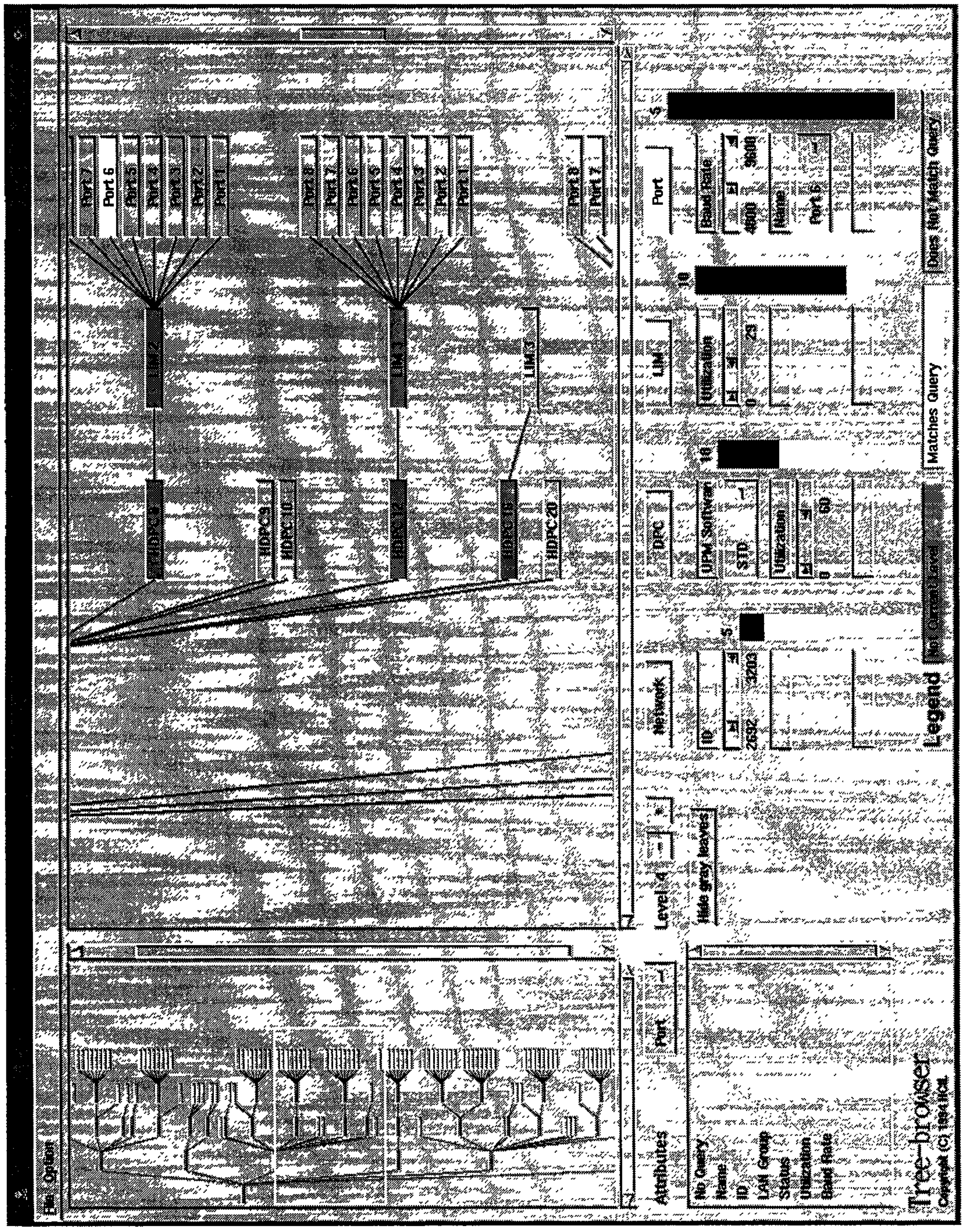


Figure 3.4: Uninteresting leaf nodes hogging up screen-space

3.3.2 UniversityFinder

The *Tree-browser* was also applied to browsing a database that organizes universities hierarchically; we have regions of the world, then states, then universities, and finally, departments. The UniversityFinder demo is available on videotape [Kum95].

Let us say that I am a high-school senior looking for universities that best match my needs. The *Tree-browser* initially shows all the regions in the world. I ask to see the entire tree to get an idea of the size of the database. The feedback indicators show that there are 740 departments in 286 universities in 62 states in 5 regions of the world (Figure 3.5).

I realize that panning this entire tree is no mean task and return to the region level. Since I am only interested in regions where English is the primary language, I create a textual menu of primary languages by dragging-and-dropping the attribute Primary Language on to one of the empty text items under the Region level button. I select English from the menu and the South America turns gray, while USA, Canada, Africa and Europe remain yellow.

I now proceed to the state level. South America's subtree is not expanded. That subtree was pruned out since South America did not satisfy our criterion of primary language. However, South America is still shown in gray in order to provide context feedback.

I would really like to study in a state that is relatively safe, so I choose to query on the level of violence. I manipulate the range-slider to select only states with no more a violence index than 65. 37 states now remain. I further reduce the number of interesting states by choosing only those with good traffic conditions. The number of states has now dropped to only 7 (Figure 3.6).

Now, I can look at the universities, but first I recapture space occupied by uninteresting states by clicking on the Hide Gray Leaves button. This gives me a more compact view. When I now go to the university level, only the remaining 7 states are expanded to show 25 universities.

I now reduce the number of universities by first eliminating those with high tuition and then setting the average SAT scores to closely match mine. I hide gray leaves once again and see that I am down to 7 universities: a couple in Arkansas, a couple in Ontario etc.

Satisfied with this set of universities, I proceed to the department level to closely look at departments (Figure 3.7) that best match my interests and needs in terms of availability of financial aid etc.

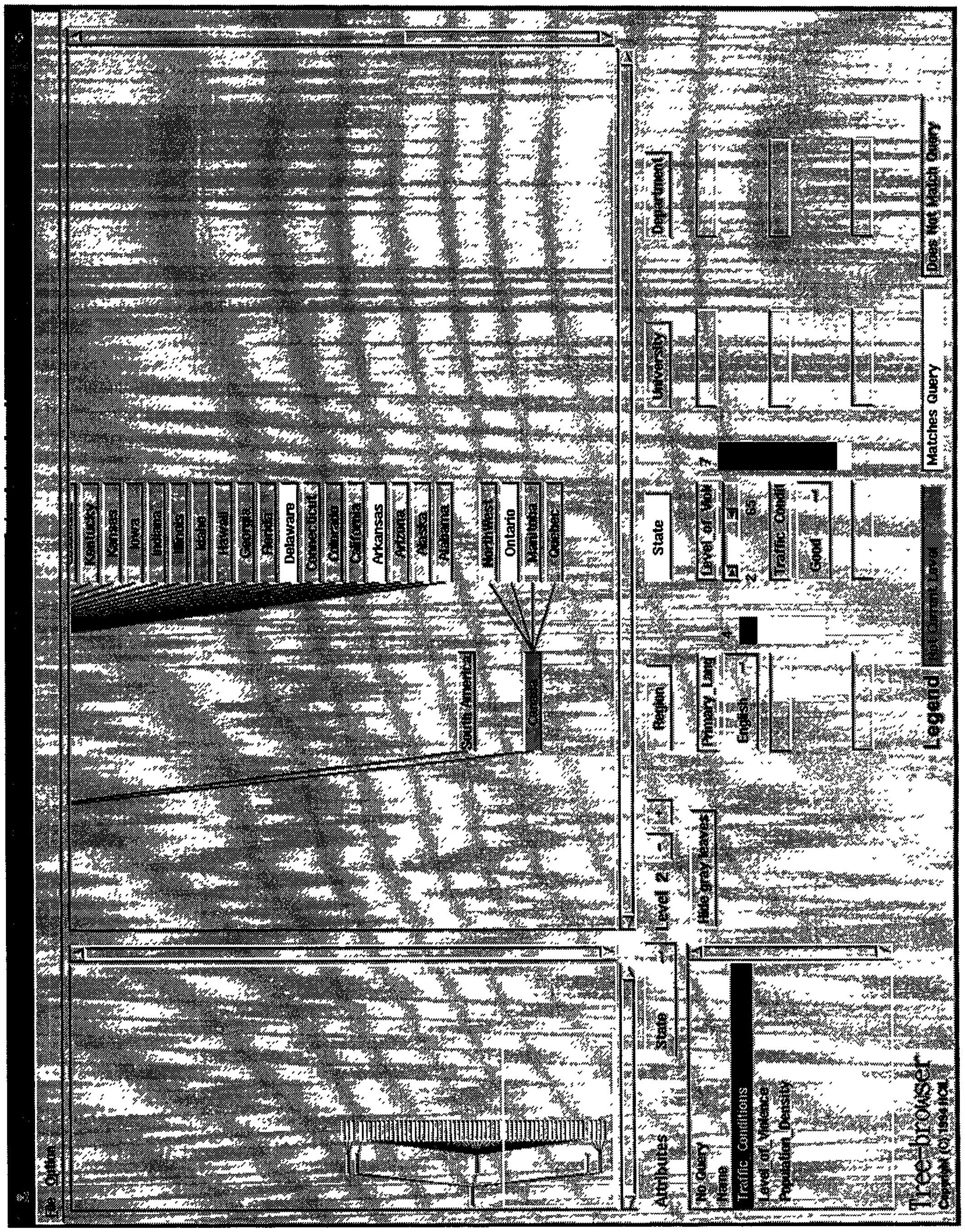


Figure 3.6: The 7 selected states

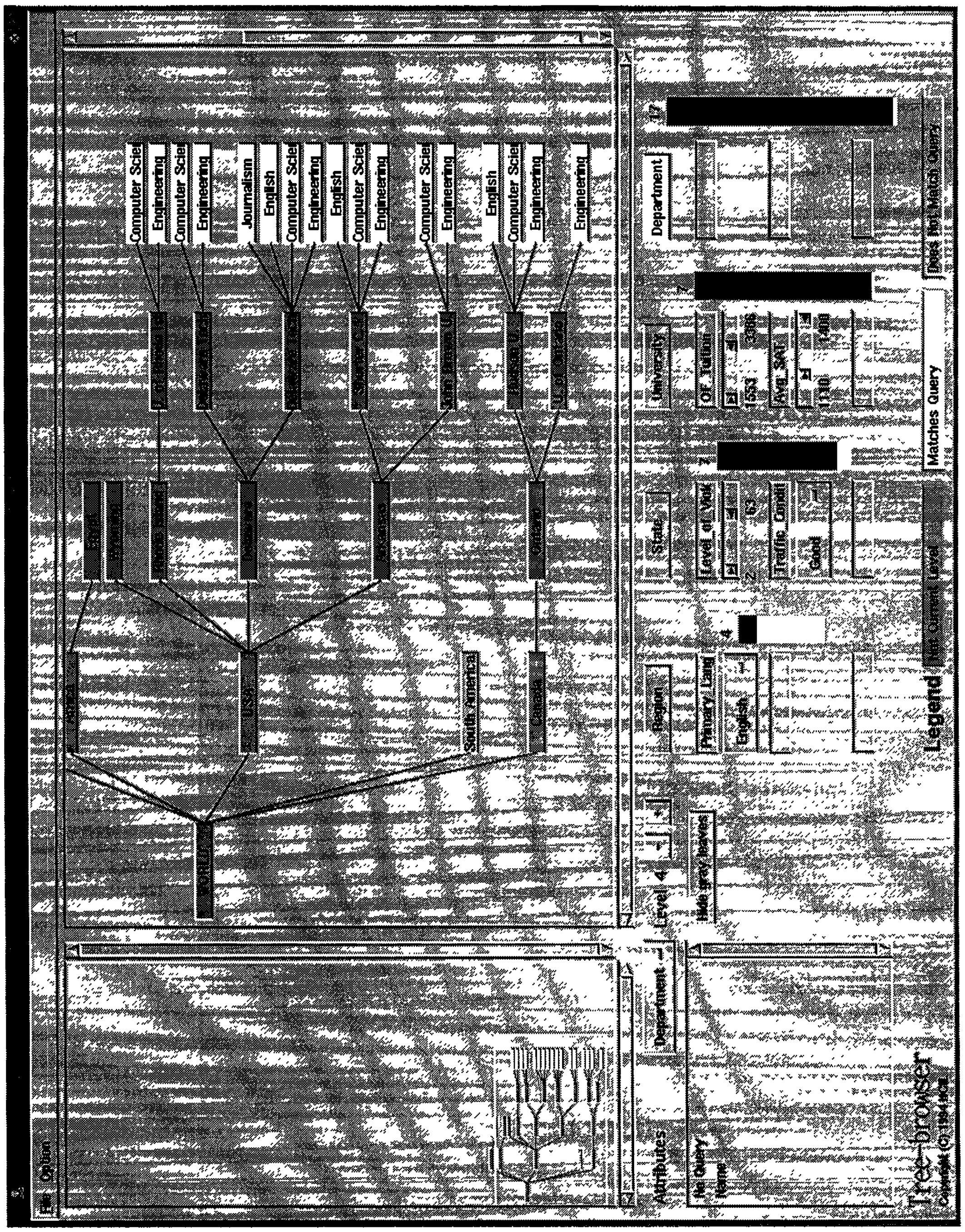


Figure 3.7: The final 7 universities and 17 departments

3.4 *Tree-browser* Limitations

The UniversityFinder scenario demonstrates how the *Tree-browser* can be applied to everyday applications, in addition to complex applications like network management. The tool itself is general and can be used for any hierarchical data set with attributes for nodes at different levels of the hierarchy. However, the current implementation does have the following limitations:

1. The *Tree-browser* interface has been "fine-tuned" for a tree of depth 5. The underlying tree data structure and treeview widget place no constraints on the depth of the tree, but the current interface has been customized to look best for a tree of depth 5, e.g. the levels of the tree in the detailed view align nicely with the corresponding buttons and query widgets.

2. Users can select only up to 3 attributes to query on at each level, and these queries can only be ANDed together. ORs and NOTs are not supported currently.

3. Since the focus of this work was not on layout, the algorithm used produces an aesthetic tidy layout, but not the most compact one. Also, the implementation does not enforce the overview to always show the entire tree (so users might have to scroll the overview).

4. Users can not open subtrees of nodes by manually selecting (e.g. double-clicking) them.

3.5 Possible design alternatives

There are some interesting design issues that deserve special mention here:

1. Pruning vs. graying out: The *Tree-browser* prunes out subtrees of nodes that do not match the query at their own level. Another approach would have been to show the entire subtree, but with all the nodes grayed out. The possible advantage of this approach over the pruning approach would be the increased constancy in the tree structure. Expert users might experience improved productivity as they get more and more familiar with the tree structure. The disadvantage of this approach is that the tree is displayed in its entirety at any level, even when most of the nodes that take up a lot of screen space are uninteresting. This results in increased overheads of scrolling and / or panning.

2. Whether to update the data display when a query widget is created or replaced at a level different from the current lowest level: The *Tree-browser* changes levels whenever widgets at levels other than the current lowest level are manipulated. But it does not change levels when a new widget is created or an existing widget replaced, at a level different from the current lowest level. This design decision was based on the assumption that users might create a number of widgets at different levels at the same time (e.g. at the beginning), and not want the level to change each time. The disadvantage of this approach is that the data display is potentially inaccurate till the next time users go to that level.

3.6 Generalizing the design

This section attempts to generalize the *Tree-browser* design so as to overcome some of its limitations and make it more generally applicable to trees of varying structures and sizes. Ideas on extending the query interface to allow specification of complex boolean queries are discussed. The *Tree-browser* illustrated the advantages of using dynamic queries and pruning while visualizing trees as 2D node-link diagrams. The same advantages are there to be had by other tree visualizations as well, e.g. treemaps and cone-trees. In this section, we illustrate this with examples.

3.6.1 Coping with varying structure and growing size

In this section, we examine some of the issues that need to be addressed in order for our design to be extensible to trees of varying structure and size.

- Trees of arbitrary depth: Figure 3.8 shows how one might extend the current *Tree-browser* interface to handle trees arbitrarily deep. Due to screen space constraints, it will not be possible to see all the query widgets at all levels in the hierarchy. In Figure 3.8, on the lower left corner, is a list of levels in the hierarchy, from which users select the current lowest level (level 2 in this case). The Attributes List then updates to show the list of attributes for level 2. The query widgets for the current lowest level and for the previous level visited (i.e. level 1 in this case) are shown in the query panel.

- Trees of arbitrary size: As the size of the tree to be visualized becomes larger and larger, performance would tend to deteriorate and the browsing mechanism inadequate. But there are approaches that can be taken to alleviate this problem:

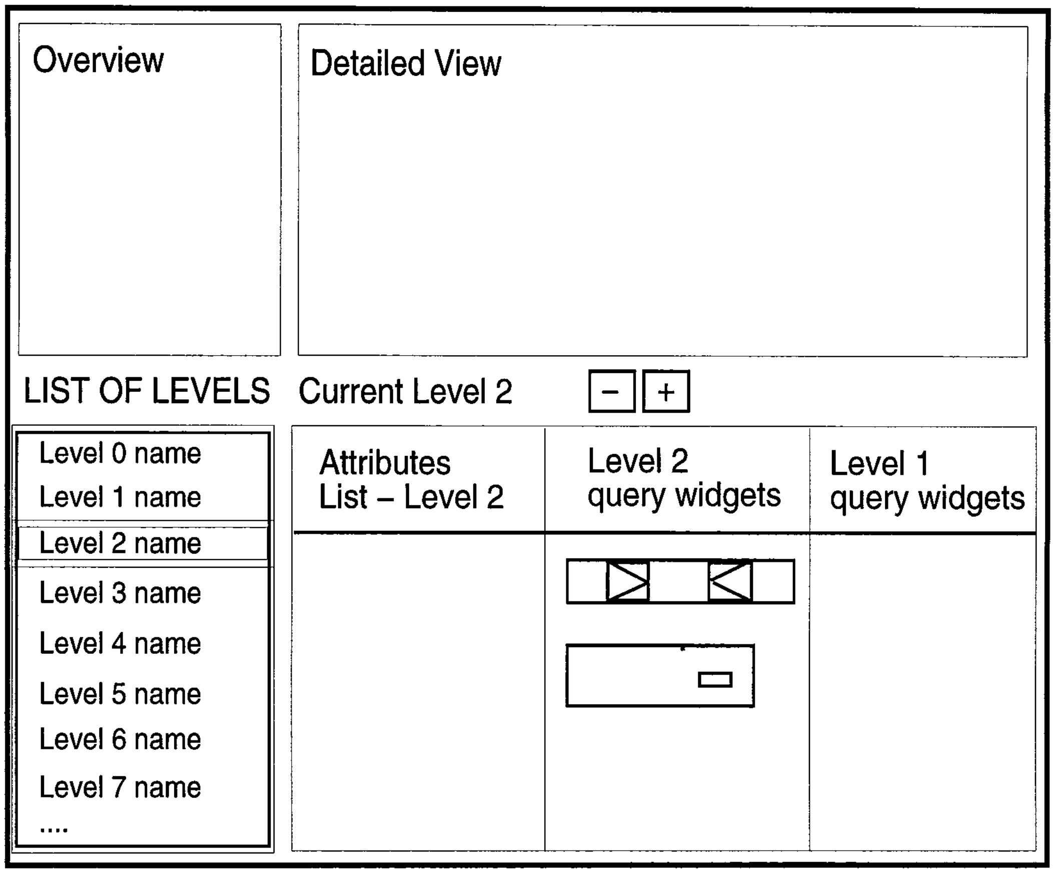


Figure 3.8: Extending the interface to trees of arbitrary depth

1. Performance issues: Section 4.2 describes the algorithms used in implementing the *Tree-browser*. The node-link layout algorithm, in the worst case (i.e. when all nodes are to be shown), requires 2 complete traversals of the tree, Increasing levels displayed requires more than 3 traversals of the entire tree, in the worst case, etc. Dynamically querying nodes at any level is more efficient in that is only requires traversal of each node at that level (See Section 4.1.4 for details). Therefore, the system is likely to slow down appreciably as the size of the tree increases, especially for operations that require re-computation of the

structure and layout.

More sophisticated data structures and algorithms would be necessary in order to minimize the performance deteriorations. For example, one might only traverse the nodes visible in the detailed view to evaluate dynamic queries. But this would mean that the overview would not be in synch with the detailed view. Algorithms for pruning could be improved so that only subtrees that have been dynamically queried since the last structure change need to be traversed.

It is clear that there are several interesting challenges that remain to be solved, with respect to performance issues.

2. Interface issues: Ideally, the overview would always show the entire data display in miniature, even if it means that no details are visible; and the overview provides only global context. However, when the size of the tree gets huge (say 50,000 nodes), there is no way that one overview would suffice. As reported in Section 2.3.1, Plaisant et al. [PCH92] found that in some cases, it might even be useful to provide an intermediate view in addition to the overview and the detailed view, when the detail-to-overview zoom ratio is high. An experiment showed that users performance significantly degraded when no intermediate view was provided for a detail-to-overview zoom ratio over 20:1.

   Another feature that might be useful is to allow users to restrict the nodes to be displayed **before** displaying them. For example, if the user requests to see a new level with 20,000 nodes, the system should present the user with the option of restricting this set (feedback might be provided by displaying the number of matching nodes) before displaying it.

3.6.2 Specification of general boolean queries

The current *Tree-browser* implementation allows users to specify an AND of queries on up to any 3 attributes. This flexibility should be sufficient for many applications, but it would be better to have an interface that would allow any number of ANDs, ORs and NOTs. It would be rather difficult to interpret a set of graphical widgets, boolean operators and parentheses, even for experienced users.

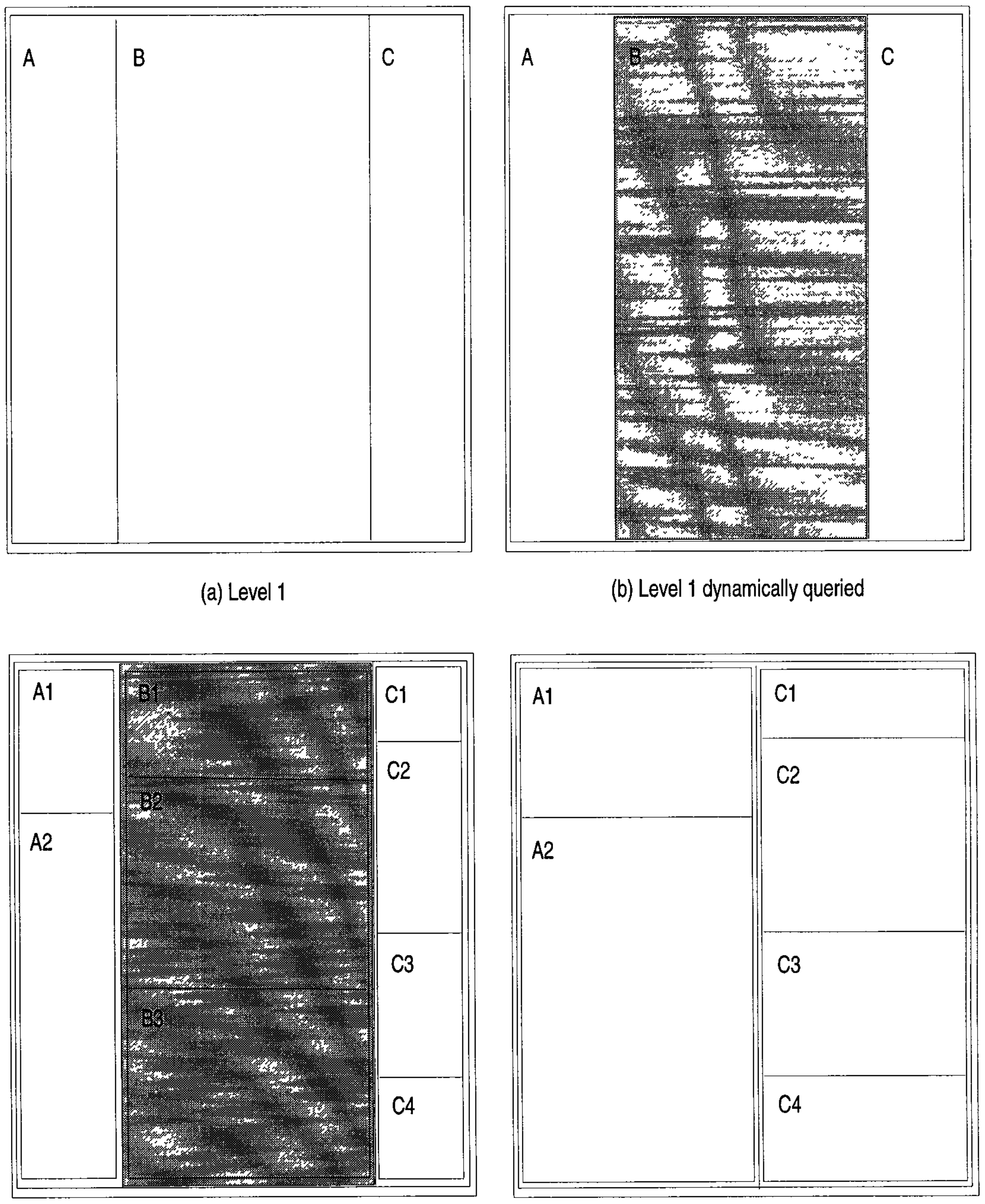
One possible interface that could be used in the *Tree-browser* to specify complex boolean queries was proposed by Young and Shneiderman [YS93]. They note that "extracting information using a boolean query is often too abstract for novice users and problematic for many experienced users". Filter/flow was designed to present the boolean operators using a metaphor that graphically conveys the meaning of the operators. This interface is void of any logical operators, parentheses, or any form of fixed syntax. Queries are shown as water streams flowing through sequential filters for ANDs, parallel filters for ORs, and toggled filters for NOTs. The decrease in the breadth of the water stream while passing through a filter represents the reduction in the data set due to the corresponding query.

Some minor modifications to the interface of Young and Shneiderman would however, be necessary. In their interface, the initial tuples and resulting tuples are displayed in textual lists. Since our data display already shows the results as graphical trees, we can instead place feedback indicators on either end of the filter/flow query panel, and this would further help users to refine their queries based on current needs and interests.

3.6.3 Pruning applied to Treemaps

Figure 2.5 was in fact, one treemap that first highlighted the need for some mechanism to hide subsets of nodes / prune subtrees that were not interesting and recapture screen space. That figure shows a treemap of a network hub, where the user had dynamically queried at the LIM level, resulting in some LIMs graying out. Then, the user had proceeded to the port level, and all ports on gray LIMs had remained gray. These ports, which were uninteresting to the user took up about 40% of the total display space.

Figure 3.9 shows how pruning can help treemaps recapture screen space allocated to uninteresting subtrees.



(a) Level 1

(b) Level 1 dynamically queried

(c) Level 2 without pruning

(d) Level 2 with pruning

Figure 3.9: Pruning applied to treemaps

It is worth comparing at this point, the *Tree-browser* visualization and the treemap visualization. Both have some advantages over the other as listed below:

Advantages of the *Tree-browser* over the treemap:

1. More comprehensible representation of tree structures: The traditional node-link representation is more readily comprehensible to untrained users than the space-filling sliced-and-diced representation of treemaps.

2. More general in terms of attributes at various levels in the hierarchy: The *Tree-browser* requires nodes at each level in the hierarchy to share a common set of attributes. Each level can have a different set of attributes. Attributes can also be shared among levels. Attribute values can be made to aggregate up the hierarchy if desired (i.e. the attribute value for a parent node would be the sum of values for its children), but this is not required. The treemap, on the other hand, requires that there exist at least one meaningful numerical attribute whose values aggregate up the tree. In the absence of such an attribute, one has to resort to unit-sized leaf nodes, and the power of the treemap visualization is reduced.

3. No aspect ratio confusions: Comparing areas of nodes in treemaps can be difficult, especially when aspect ratios of nodes vary widely. Treemaps are not effective for finding medium-valued nodes. Even if size-coding of nodes is used in the *Tree-browser*, there will be no aspect ratio confusions because only one dimension (either width or height) of nodes needs to be encoded.

Advantages of the treemap over the *Tree-browser*:

1. Much more efficient use of screen space: No scrolling or panning of the data display is required since the entire data display is always visible.

Treemaps are good for providing compact overviews and rapid access to detailed information.

2. Treemaps are good for data sets that have meaningful numerical attributes that aggregate up the hierarchy. Significant variance in attribute values of leaf nodes is good. The space-filling representation makes size-coding effective while looking for extreme-valued nodes (i.e. large-valued nodes or small-valued nodes with size inversion applied). Size-coding of node-link diagrams would be helpful but not equally effective (since the representation is not space-filling).

3. While the *Tree-browser* is more general in that it allows different sets of attributes at distinct levels in the hierarchy, such distinct levels do not always exist. For example, a file system is a hierarchy that does not have distinct levels (analogous to Region, State etc. in the UniversityFinder tree). The number of levels itself is not fixed in a file system. In this case, all levels share the same set of attributes. The *Tree-browser* is not as attractive here, because users often are **not** looking for nodes at specific levels, instead they are looking for nodes at any level satisfying certain criteria (e.g. all files with size $>$ 1 Meg).

3.6.4 Dynamic Queries applied to Cone-Trees

Robertson et al. [RMC91] describe "gardening operations" where the user can manually prune and grow the view of the tree (Figure 2.6). Prune and grow operations are done either by menu or by gestures directed at a node. We believe that allowing users to make attributes-based subset specifications in addition to these manual subset specifications will help make Cone-Trees more powerful and useful.

Chapter 4

Software design

4.1 Software System Architecture and Data Structures

The *Tree-browser* is implemented in about 8500 lines of C code, with Application Programmer Interface (API) calls to Galaxy libraries (Version 2.0). It was developed on the SUN SPARCStation / UNIX / X Windows platform.

The *Tree-browser* software system consists of 4 subsystems as shown in Figure 4.1:

1. Tree data structure subsystem
2. Treeview node-link diagramming subsystem
3. Tight-coupling of overview and detailed view subsystem
4. Query specification and evaluation subsystem

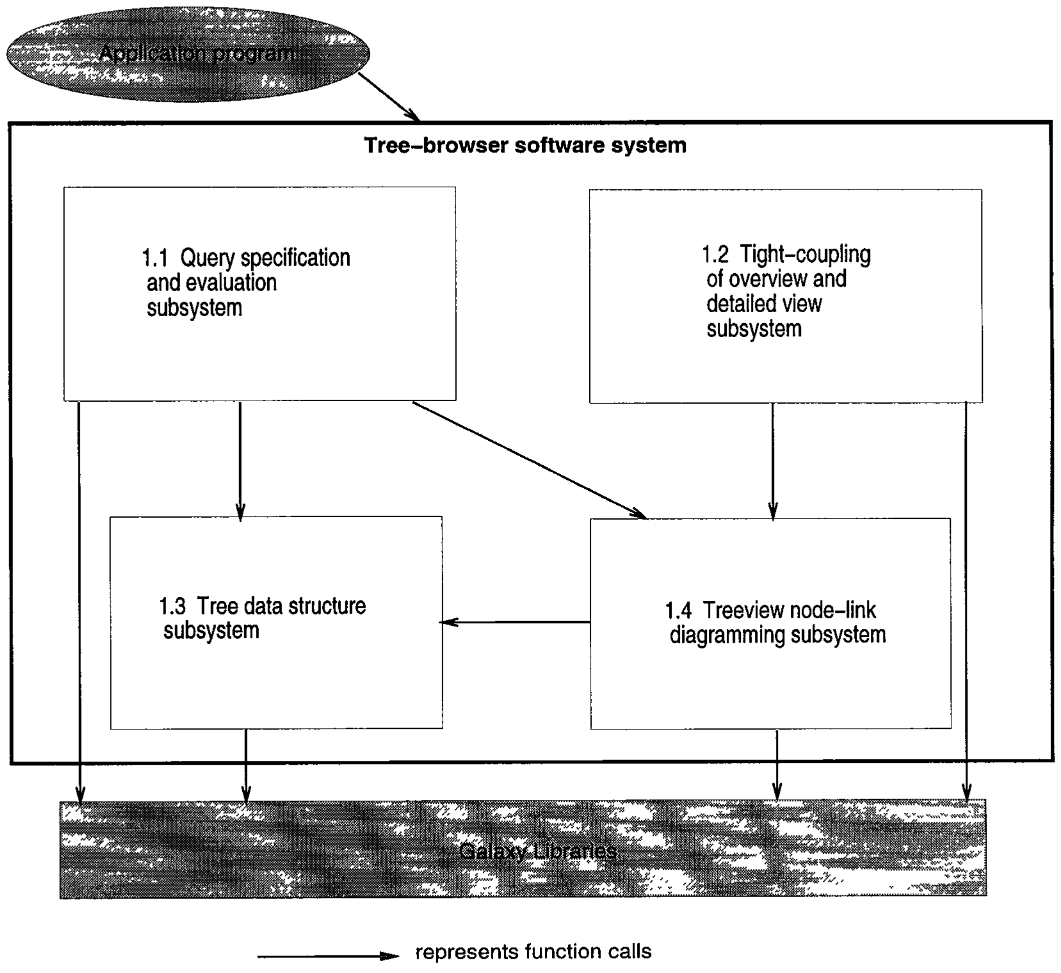

Application program
Tree–browser software system
1.1 Query specification and evaluation subsystem
1.2 Tight–coupling of overview and detailed view subsystem
1.3 Tree data structure subsystem
1.4 Treeview node–link diagramming subsystem
Galaxy Libraries
represents function calls

Figure 4.1: *Tree-browser* software system architecture

4.1.1 Tree data structure subsystem

Since there are numerous applications that make use of trees, and various visualizations (e.g. node-link, treemap etc.), a common tree data structure (henceforth referred to as "tree") was designed and developed. Figure 4.2 shows how the nodes of the tree are stored and linked to parents, siblings and children. Any m-ary tree can be stored in this data structure. Any number of visualizations can make use of the same tree. This enables us to avoid duplication of node data for each visualization. The tree data structure itself is independent of the visualizations, in order to ensure modularity and extensibility. We have developed a node-link treeview widget and a treemap widget on top of this tree data structure (Figure 4.3). More visualizations can easily be built on top of this tree, by making use of its Application Programming Interface (API).

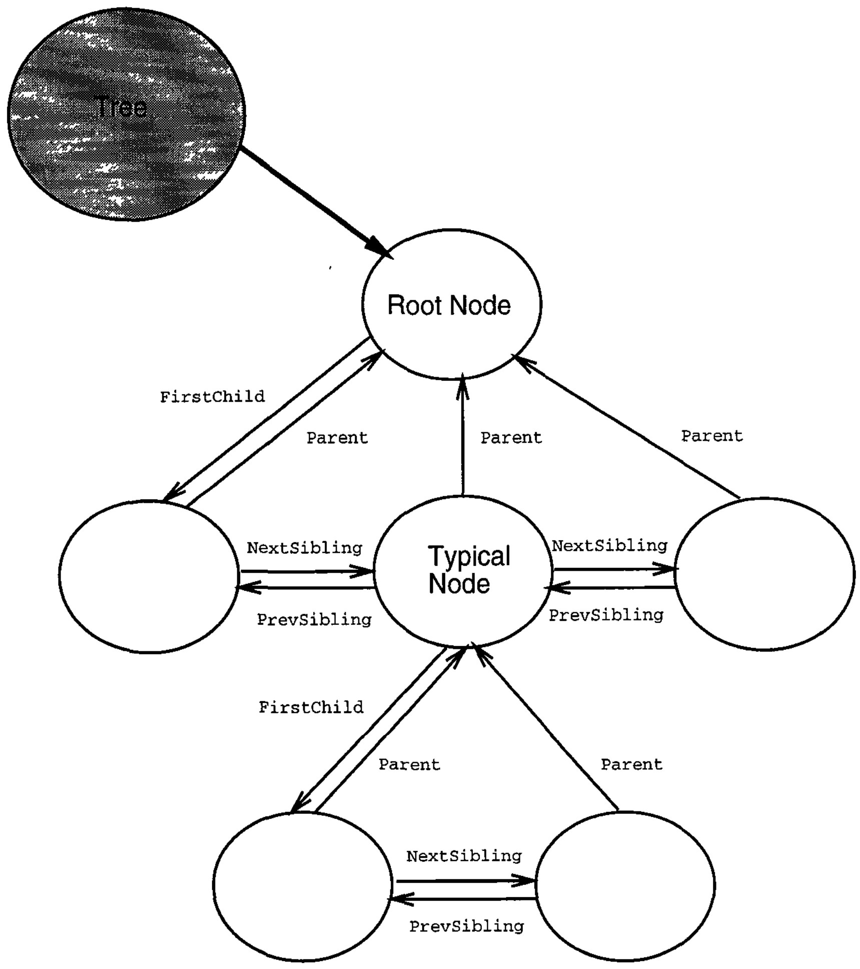


Figure 4.2: Tree data structure

The tree can be loaded from a file, or created dynamically at run-time. The tree data file uses a specific format to specify the tree structure in preorder (i.e. parent first), which is described below.

Tree Data File Format: The tree data file contains two parts, the data

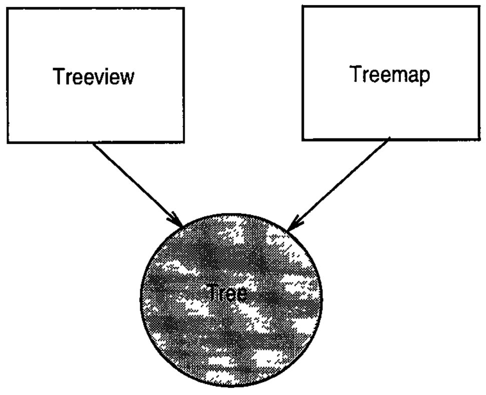


Figure 4.3: Treeview and Treemap visualizations built on top of the tree data structure

definition part, and the actual data part. The data definition part defines the number of attributes for the nodes, and the names and data types of the attributes. The actual data part then specifies the data for each node of the tree.

The data file starts with a file version string which should be "Tree Data File 1.0" (typed without quotation marks). If the file format is later improved, the version number is used to differentiate the versions. This is followed by the tilde character

(~)

which is the field delimiter used throughout the file.

Data Definition: The second field defines the number of attributes for all the nodes. In the current implementation all nodes must have the same set of attributes. If the set of attributes for some nodes is different from the set of attributes for other nodes, the superset of all attributes must be specified for all nodes. For attributes that are not valid for a given node, some distinct default values can be used.

Next the file must list the data type and name of each attribute. This is done by pairs of fields, the first one describes the type and the second the name of the attribute. The data type is specified by a single character, I for integer attribute, F for floating point number attribute, T for text attribute, and M (miscellaneous) for any other type attribute. The fields containing values for attributes of type M (see below, the part about actual data) are skipped, but they must be present in the file. Number "0" can be used as a place-holder for such data. The name of an attribute can be any string, but it must be unique.

Therefore, the data definition part looks like this:

```
Tree Data File 1.0~<Number of Attributes>~(<Attribute Type>
                                           ~<Attribute Name>~)
```

The parentheses () indicate that the enclosed parameters are to be repeated < Number of Attributes > times, one for each attribute.

Actual Data: The actual data is formatted in a nested tree structure (pre-order). Data for a single node is enclosed in brackets: "[" and "]". Inside the brackets the attribute values for the node must be listed in the same order as the attributes were introduced in the data definition part. All the fields, including the last one, must end with the field delimiter

"~".

If the node is a leaf node, the closing bracket "]" follows. If the node has child nodes, they are specified between the last delimiter character and the closing bracket for that node. Each child node is also formatted as described above.

Therefore, the actual data part looks like this:

```
[value1~value2~...~valuen~[child1's value1~child1's
value2~...~child1's valuen~][child2's value1~ child2's
value2~...~child2's valuen~]...]

if <Number of Attributes> = n.
```

Naturally the child nodes could have their own children, but they are not shown in the above example for clarity. Characters

```
"~", "[", "]", and "\"
```

have special meaning and they cannot be used inside the fields as such. They must be quoted using the backslash character

```
"\".
```

So they would appear as

```
"\~", "\[", "\]", and "\\"
```

respectively.

The white-space characters (spaces, tabs, newlines, carriage returns) are stripped from the beginning of the fields so that white-space can be used to make the file more readable. If a field needs to start with white-space (only spaces and tabs allowed), the first white-space character must be quoted using the backslash character

```
"\".
```

White-space can also be used in the file between two delimiter characters

```
("~", "[", and "]").
```

The following example helps understand the format better.

Example: The tree shown in Figure 4.4 would have the corresponding input data file:

```
Tree Data File 1.0~
3~
T~Name~
I~ID~
T~Address~
[Company~0~12345 ABC St~
        [Business~1~12345 ABC St~
                [John Smith~2~9574 XYZ Ave~]
                [Rick Rogers~3~5345 XYZ Ave~]]
        [Accounts~4~12345 ABC St~]
        [Marketing~5~12345 ABC St~
                [Bill Crighton~6~84 D St~]
                [Wayne Palmer~7~8437 P Blvd~]
                [Dan DeVoe~8~84 D St~]]
        [Engineering~9~12345 ABC St~
                [Mark Hunter~10~6805 R Rd~]
                [Harry Chekov~11~3174 P Blvd~]]]
```

4.1.2 Treeview node-link diagramming subsystem

The *Tree-browser* uses two tightly-coupled treeviews, as described in section 3.2. This subsystem is responsible for the automatic layout of each treeview. The *Treeview* widget of the University of Maryland Widget Library^{TM} was used for this purpose [CJK+94].

The treeview widget is a node-link visualization which takes as input a tree data structure (Figure 4.2), generates the layout coordinates for the nodes and links and renders them. The treeview data structure was designed to be flexible, extensible and customizable to specific applications. For example, various layout

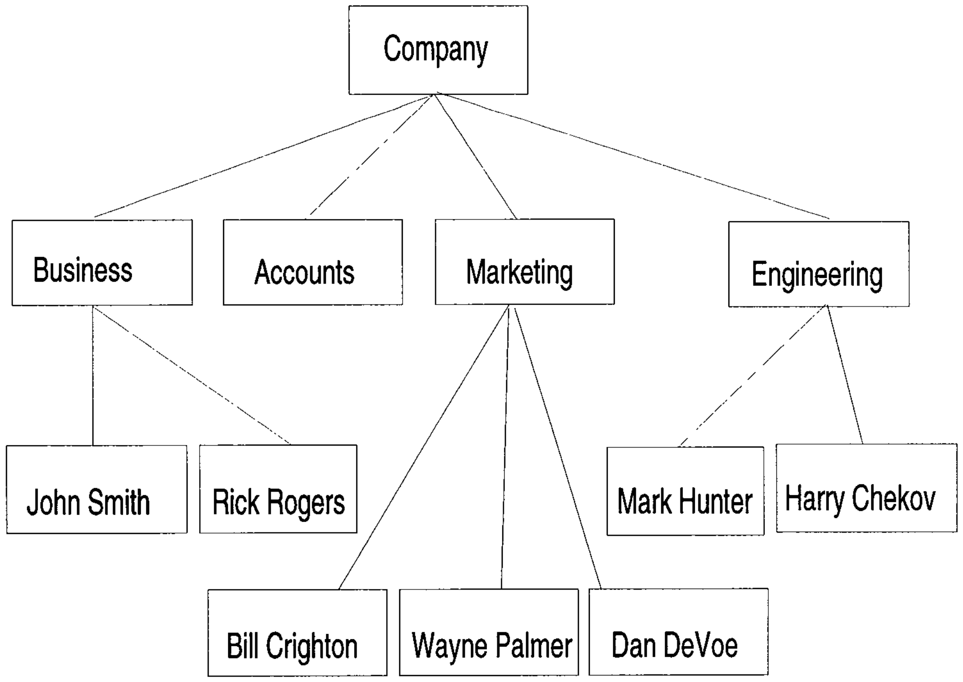


Figure 4.4: Tree Data File Format

parameters, e.g. node width, distance between nodes at adjacent levels, color of nodes, width of links etc. can be manipulated by programmers via the API; the same tree can be visualized in any number of treeviews and treemaps. For details about the treeview widget API, refer to [CJK+94].

The treeview data structure is as shown in Figure 4.5. Each node in the tree is attached to a list of *treeviewNode* structures, one for each treeview (Figure 4.5 assumes 1 treeview). Each treeviewNode contains the treeview-specific information for that node, e.g. node display coordinates and color. Therefore, the node-specific information (which is common) is reused across multiple treeviews of the same tree. Each treeviewNode structure is attached to a *treeviewLink* structure which contains the treeview-specific information for the link that connects that node to its parent, for example, coordinates of link end-points and link color.

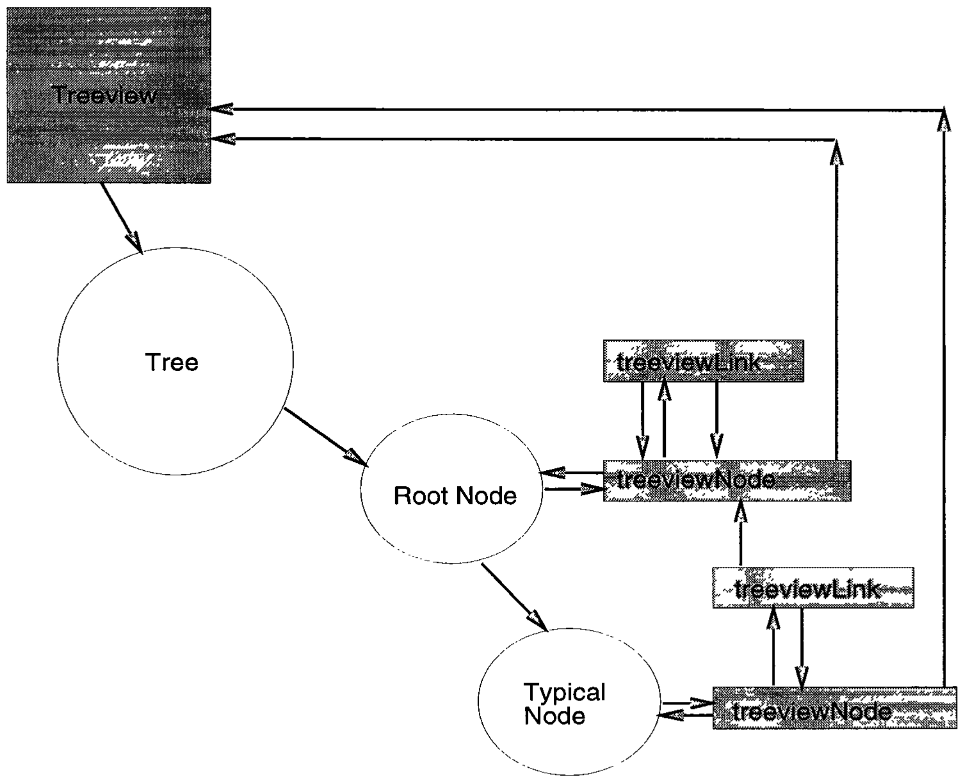


Figure 4.5: Treeview Data Structure

4.1.3 Tight-coupling of overview and detailed view subsystem

This subsystem is responsible for the tightly-coupled behavior of the two treeviews. Whenever the field-of-view is dragged in the overview, the software receives notification of these events. This subsystem then scrolls the detailed view after applying the appropriate scale factor to account for the differences in zoom. Similarly, whenever the detailed view is scrolled by manipulating the scrollbars, this subsystem is notified, and it updates the position of the field-of-view in the overview to provide global context feedback. This subsystem is also responsible for resizing the field-of-view appropriately, whenever the *Tree-browser* window is resized by the user.

The *Tree-browser's* behavior as an image-browser can be specified using the informal specification method proposed by Plaisant et al. [PCS95] (Figure 4.6).

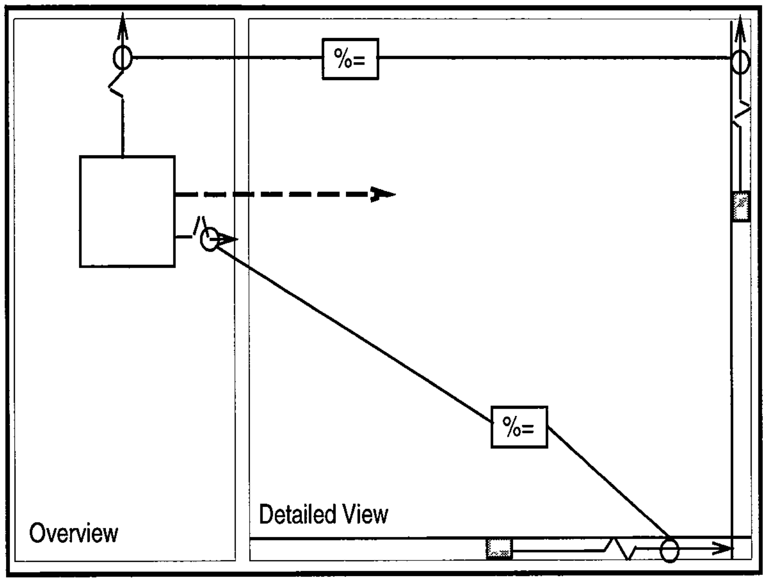


Figure 4.6: The *Tree-browser* as an Image-Browser

As per the notation used by Plaisant et al., the left window (Figure 4.6 is the source view and contains some image. The rectangle within the source view represents the field-of-view. The image enclosed by the field-of-view is projected on a second window which has scroll bars. The horizontal and vertical scroll bars are linked with the corresponding movement constraint for the field-of-view. Thus, moving the field-of-view will not only change the image in the second window, but will change the scroll bar indicator position as well. In addition, moving a scroll bar will change the position of the field-of-view and this change will cause a new projection of the field-of-view to be displayed in the second window.

4.1.4 Query specification and evaluation subsystem

This subsystem provides the functionality for the Dynamic Query Panel described in Section 3.2. There are two main modules comprising this subsystem:

- Drag-and-drop modules for building the query panel: These modules are responsible for updating the Attributes List with those of the current lowest level displayed. Notification of drags of attribute names from the list and drops onto the text items are handled by these modules. Special cases like deleting or replacing existing widgets are also handled. Once the drop is detected, the appropriate widget is created and initialized to the range of values (range-slider) / values (menu) for nodes at that level. The arrays of pointers to nodes stored in NodeArray (Figure 4.7) were used for this purpose.

- Query evaluation and data display update modules: These modules evaluate the queries as the widgets (range sliders and menus) are manipulated, and update the data display accordingly. In order to enable fast updates of the data display, arrays of pointers to nodes at each level are maintained as shown in Figure 4.7. This enables us to search only the nodes at the current lowest level when corresponding widgets are manipulated, without having to search the entire tree. Buffering was also done to make the data display updates as smooth and flicker-free as possible. Changing of levels (and the corresponding pruning operations) is also handled by these modules.

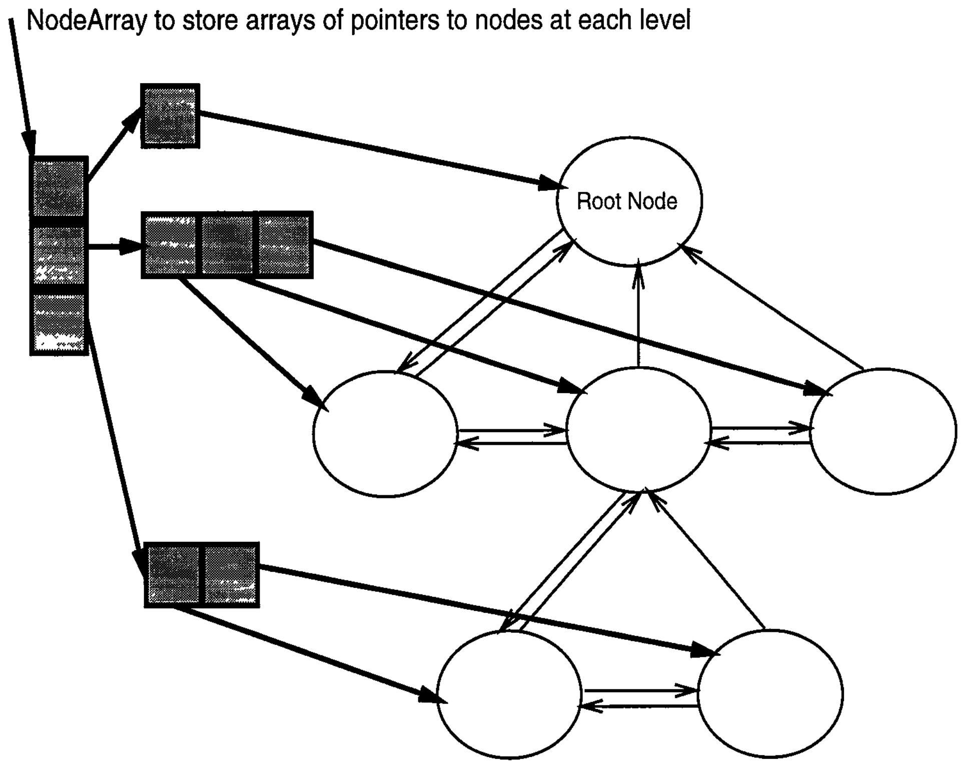

NodeArray to store arrays of pointers to nodes at each level
Root Node

Figure 4.7: Arrays of nodes at each level

4.2 Algorithms

This section presents the algorithms used in the implementation of the *Tree-browser*, in pseudo-code form. Low-level details have been abstracted out and function names have been made extra long and descriptive.

4.2.1 Node-Link Tree Layout

The node and link coordinates are generated by traversing the tree twice in postorder, i.e. children first. The first traversal generates coordinates for each node and the second generates those for each link. The values of coordinates are based upon layout parameters such as node width, distance between leaf nodes etc. (Figure 4.8). These layout parameters can be easily manipulated by the programmer through API calls.

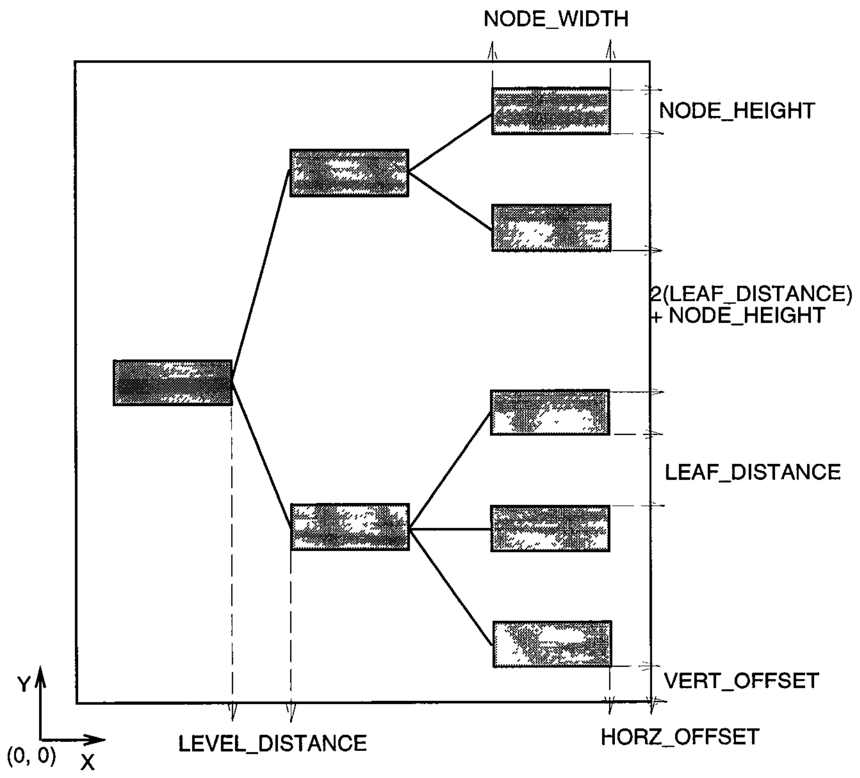


Figure 4.8: Treeview Layout

```
Predefined constants: Let -
  The width of each node be NODE_WIDTH,
  The height be NODE_HEIGHT,
  The distance between leaf nodes be LEAF_DISTANCE,
  The distance between nodes at adjacent levels be LEVEL_DISTANCE,
  The entire treeview maintain minimum offsets from
  the boundaries of the drawing canvas of VERT_OFFSET
  and HORZ_OFFSET.

Variables: Let -
  The coordinates of the lower left corner of each
  node be Node->X and Node->Y,
  The level of each node in the tree be Node->Level,
  The level of the previous node visited be PreviousLevel,
  The Y coordinate of the previous node visited be PreviousY,
  The first node visited be assigned Y = VERT_OFFSET,
  The number of leaf descendents of each node which are
  shown be Node->LeafDesc,
  and Node->ShowAttr be a boolean variable that indicates
  if the node is to be shown or not.

  The link that joins each node to its parent be Node->Link,
  and the coordinates of each link be Link->X1, Link->Y1,
  Link->X2, Link->Y2.

  Finally, TraversalNode and tmpNode are temporary node
  pointers and Done is a boolean flag.

/* Generate the treeview layout, i.e. the coordinates of
   each node and link (assuming left-to-right layout) */

GenerateTreeviewLayout(RootNode)
{

  /* Generate Node Coordinates first */
  Done = FALSE;

  while(Done == FALSE)
    {
      while(TraversalNode->ShowAttr == 1 AND
                TraversalNode->FirstChild exists)
        TraversalNode = TraversalNode->FirstChild;

      if(TraversalNode->ShowAttr == 1)
        AssignNodeCoordinates(TraversalNode);

      if(TraversalNode->NextSibling exists)
        TraversalNode = TraversalNode->NextSibling;
      else
        {
          while(TraversalNode->Parent exists AND
            TraversalNode->Parent->NextSibling does not exist)
            {
              TraversalNode = TraversalNode->Parent;
            if(TraversalNode->ShowAttr == 1)
              AssignNodeCoordinates(TraversalNode);
          }

        if(TraversalNode->Parent == NULL)
          Done = TRUE;
        else if(TraversalNode->Parent->NextSibling exists)
          {
            TraversalNode = TraversalNode->Parent;

            if(TraversalNode->ShowAttr == 1)
              AssignNodeCoordinates(TraversalNode);

            TraversalNode = TraversalNode->NextSibling;
          }
      }
  }

/* Generate Link Coordinates */
Done = FALSE;

while(Done == FALSE)
  {
    while(TraversalNode->ShowAttr == 1 AND
             TraversalNode->FirstChild exists)
      TraversalNode = TraversalNode->FirstChild;

    if(TraversalNode->ShowAttr == 1)
      CalculateLinkCoordinates(TraversalNode);

    if(TraversalNode->NextSibling exists)
      TraversalNode = TraversalNode->NextSibling;
    else
      {
        while(TraversalNode->Parent exists AND
          TraversalNode->Parent->NextSibling does not exist)
          {
            TraversalNode = TraversalNode->Parent;

            if(TraversalNode->ShowAttr == 1)
              CalculateLinkCoordinates(TraversalNode);
          }
          if(TraversalNode->Parent == NULL)
            Done = TRUE;
          else if(TraversalNode->Parent->NextSibling exists)
            {
              TraversalNode = TraversalNode->Parent;

              if(TraversalNode->ShowAttr == 1)
                CalculateLinkCoordinates(TraversalNode);

              TraversalNode = TraversalNode->NextSibling;
            }
        }
    }
}

/************************************/

/* Assign coordinates to given node */

AssignNodeCoordinates(Node)
{
  /* X coordinate is easily determined from level */
  Node->X = HORZ_OFFSET + Node->Level *
              (NODE_WIDTH + LEVEL_DISTANCE);

  if(Node->Level == PreviousLevel)
    /* Single increment for Y */
    Node->Y = PreviousY + (NODE_HEIGHT + LEAF_DISTANCE);
  else if(Node->Level > PreviousLevel)
    /* Double increment for Y since new subtree */
    Node->Y = PreviousY + 2*(NODE_HEIGHT + LEAF_DISTANCE);
  else
    /* Center node above all its leaf descendents
       that are shown */
    Node->Y = PreviousY - (Node->LeafDesc / 2)*
                    (NODE_HEIGHT + LEAF_DISTANCE);
}

/*****************************************************/

/* Calculate the end-points of the link that joins the
```

```
   given node to its parent */

CalculateLinkCoordinates(Node)
{

    Node->Link->X1 = Node->X;
    Node->Link->Y1 = Node->Y + (Node->H)/2;

    tmpNode = Node->Parent;
    Node->Link->X2 = tmpNode->X + tmpNode->W;
    Node->Link->Y2 = tmpNode->Y + (tmpNode->H)/2;

}
```

4.2.2 Dynamic Querying and Tree Pruning

Each node in the tree shown in Figure 4.2 has some attributes. The algorithms for automatic tree layout, dynamic querying and tree pruning make use of two special restricted attributes, *ShowAttr* and *MatchesAttr*, defined as follows:

- ShowAttr: This value of this attribute is 1 for nodes that are to be shown in the treeview. For hidden nodes, the value is 0.
- MatchesAttr: This attribute makes sense only for nodes with ShowAttr equal to 1. The value of this attribute is 1 if the node matches the current query at its own level, else the value is 0.

These attributes are added to the tree in the *Tree-browser* initialization code. Depending on the initial levels to be displayed, all nodes from the root to the lowest level displayed are initialized with ShowAttr and MatchesAttr equal to 1, while nodes at deeper levels are initialized with ShowAttr and MatchesAttr equal to 0.

The need for using these two attributes, rather than deleting the nodes that do not match the query from the tree data structure is that the system has to allow users to back-track and iterate. By using these attributes, we are able to remove nodes from the visualization without actually removing them from the tree data structure.

The algorithms related to dynamic querying and tree pruning are as follows:

1. Callback function executed when widgets (range-sliders and menus) are manipulated. This function searches NodeArray (Figure 4.7) at that level, queries nodes based on the widget states and updates the node colors accordingly.

```
Variables:
  WidgetLevel: Tree level corresponding to widget
  Level: Tree-browser current lowest level

WidgetNotify(WidgetLevel, Level)
{
  if(WidgetLevel > Level)
    /* Increase Tree-browser level */
    IncreaseLevelsProc(Level, WidgetLevel);
  else if(WidgetLevel < Level)
    /* Decrease Tree-browser level */
    DecreaseLevelsProc(Level, WidgetLevel);
  else if(WidgetLevel == Level)
    {
      /* Update node color for all nodes at level Level
         based on query */
      QueryNodesAndSetColor(Level);
      UpdateFeedbackIndicators();
    }
}
```

Note: The function QueryNodesAndSetColor() called from this function is similar to the function QueryNodesAndSetShowAttr(), specified later in this section.

2. Function to execute whenever the levels displayed are to be decreased:

```
Variables:
  OldLevel: Previous level
  Level: New level to change to (Level < OldLevel)

DecreaseLevelsProc(OldLevel, Level)
{
integer i;

  /* Set ShowAttr to 0 for all nodes deeper than the
     lowest level to be displayed */
  for(i = Level + 1; i <= OldLevel; i++)
    SetShowAttrToFalseForAllNodes(i);

  /* Set ShowAttr for each node in the new lowest
     level based on query */
  if(Level != 0)
    QueryNodesAndSetShowAttr(Level);

  /* Grow and prune the tree till level Level,
     one level at a time */
  GrowAndPruneTree(Level);

  /* Recreate the Tree-browser till the level Level */
  CreateTreeBrowser(Level);

  /* Update the color of nodes at all displayed levels
     based on the value of ShowAttr: 1 maps to yellow or
     orange (depending on level), 0 maps to gray */
  for(i = 1; i <= Level; i++)
    SetColorBasedOnShowAttr(i);

  /* Update other interface objects, e.g. level label,
     Attributes List, feedback indicators etc.*/
  UpdateInterfaceObjects();
}
```

3. Function to execute whenever the levels displayed are to be increased:

```
Variables:
  OldLevel: Previous level
  Level: New level to change to (Level > OldLevel)

IncreaseLevelsProc(OldLevel, Level)
{
integer i;

  /*Set ShowAttr for each node at the level OldLevel
    based on its color. Yellow maps to 1 and gray to 0 */
  SetShowAttrBasedOnNodeColor(OldLevel);

  /* For nodes in the new levels displayed, set ShowAttr
     based on the query at the respective levels */
  for(i = OldLevel + 1; i <= Level; i++)
    QueryNodesAndSetShowAttr(i);

  /* Grow and prune the tree till level Level,
     one level at a time */
  GrowAndPruneTree(Level);

  /* Recreate the Tree-browser till the level Level */
  CreateTreeBrowser(Level);

  /* Update the color of nodes at all displayed levels
     based on the value of ShowAttr: 1 maps to yellow or
     orange (depending on level), 0 maps to gray */
  for(i = 1; i <= Level; i++)
    SetColorBasedOnShowAttr(i);

  /* Update other interface objects, e.g. level label,
     Attributes List, feedback indicators etc.*/
  UpdateInterfaceObjects();
}
```

4. Function to grow and prune the tree till level TillLevel, one level at a time.

```
Variables:
  CurrentNode: A pointer to a tree node, initialized to the
  root.
  TillLevel: The level till which the tree is to be grown.
  NumChildren: A variable to store the number of children
  of each node. This is required since this function uses
  a preorder (parent-first) recursive tree search.

GrowAndPruneTree(CurrentNode, TillLevel)
{
integer NumChildren, i;

  /* If node is within levels to be displayed */
  if(Level of CurrentNode <= TillLevel) {
    /* If node is to be shown at all or not */
    if(GetShowAttr(CurrentNode) == 1) {
      /* If the node's descendents are to be shown or not */
      if(GetMatchesAttr(CurrentNode) == 1) {
        NumChildren = GetChildCount(CurrentNode);
        /* Repeat for children */
        For each child do (i.e. from 1 to NumChildren)
          {
            /* Set CurrentNode to point to each child node
               in turn */
            GrowAndPruneTree(CurrentNode, TillLevel);
          }
      }
      else if(GetMatchesAttr(CurrentNode) == 0) {
        /* Show current node */
        SetShowAttrToTrue(CurrentNode);

        NumChildren = GetChildCount(CurrentNode);
        /* Hide all subtrees of current node */
        For each child do (i.e. from 1 to NumChildren)
          {
            /* Set CurrentNode to point to each child node
```

```
               in turn */
             SetShowAttrToFalseForSubtree(CurrentNode);
           }
         }
       }
       else if(GetShowAttr(CurrentNode) == 0)
         SetShowAttrToFalseForSubtree(CurrentNode);
     }
     else
       SetShowAttrToFalse(CurrentNode);
   }
```

5. Function to query all nodes at a given level and set the ShowAttr for each based on results of the query:

```
Variables:
  condition1, condition2, condition3: The results of
  querying each node on the basis of each of the 3
  query widgets. These boolean values are AND-ed
  together in the following function.
  CurrentNode: Current node for which query is being
  evaluated.

QueryNodesAndSetShowAttr(Level)
{
integer j;

  /* Evaluate the query for each node at that level by
     accessing the array of pointers to nodes at that
     level */
  for(j = 0; j < NodesAtThatLevel; j++)
  {
    CurrentNode = NodeArray[Level][j];
    condition1 = DoQueryOnNode(CurrentNode, widget1);
    condition2 = DoQueryOnNode(CurrentNode, widget2);
    condition3 = DoQueryOnNode(CurrentNode, widget3);

    if(condition1 AND condition2 AND condition3) {
        /* Set ShowAttr to TRUE and MatchesAttr to TRUE */
        SetShowAttrToTrue(CurrentNode);
        SetMatchesAttrToTrue(CurrentNode);
      }
      else {
        /* Set ShowAttr to TRUE and MatchesAttr to FALSE */
        SetShowAttrToTrue(CurrentNode);
        SetMatchesAttrToFalse(CurrentNode);
      }
    }
  }
```

Chapter 5

Usability Testing

In order to assess and improve the usability of the *Tree-browser*, semi-formal usability evaluations were performed, in which subjects were given some training and then asked to perform some tasks using the *Tree-browser.* The UniversityFinder data (Section 3.3.2) was used in the testing. This chapter describes the testing goal, methods, subjects, tasks and results obtained.

5.1 Goal

The goal of undertaking the Usability Testing was to assess the strengths and weaknesses of the *Tree-browser.* It was hoped that subjects' comments and suggestions would help refine the *Tree-browser* design and implementation, and enhance our understanding of tree-browsing issues.

5.2 Methods

With each subject, the following procedures were followed:

1. Introduction: The experiment was introduced to the subjects by explaining the UniversityFinder scenario (Section 3.3.2).

2. Description of features: *Tree-browser* features were demonstrated one by one and subjects were given the opportunity to try each one.

3. Tasks: Subjects were asked to perform 7 tasks. The initial tasks focussed on specific features, while later tasks were designed to evaluate the tool as a whole (See Section 5.4 for details). Subjects were encouraged to think aloud while performing the tasks. Comments and suggestions were recorded as they were made. Interesting actions and sources of confusion were also recorded.

4. Subjective evaluation: Subjects were asked to rate specific features of the system (on a scale of 1 to 9), identify what they liked and disliked most about the *Tree-browser* and make suggestions for improvement.

5. Analysis: Finally, all comments and suggestions made by subjects were compiled into one list (Section 5.5). The mean and standard deviation were computed for each rating and some graphs were plotted.

5.3 Subjects

First, pilot testing was done with 2 fellow graduate students at the HCIL, in order to try out the tasks and get preliminary reactions. Based on the pilot tests, the tasks were refined and the subjective evaluation questionnaire was designed.

The software was then tested with 6 subjects. Familiarity / experience with the following were prerequisites for subjects who participated in the study:

- Tree structures and pruning.
- Database querying.
- GUIs and direct manipulation widgets.

The subjects ranged from graduate students and undergraduate seniors in Computer Science / Electrical Engineering to a faculty research assistant at the HCIL. Most of them had some familiarity with the *Tree-browser*, but only one of them had used the *Tree-browser* before.

5.4 Tasks

In order to come up with as complete a set of test tasks as possible, the testing was classified as follows:

- Feature-based testing: Testing the specific features of the *Tree-browser*.
    1. Tight-coupling of overview and detailed view: 2D browsing of the node-link diagrams using panning, scrolling and changing levels features only.
    2. Dynamic Query Environments: Building and modifying dynamic query panels using drags-and-drops. Directly manipulating the widgets to produce real-time color updates of the nodes.
    3. Tree Dynamics: Tree pruning and associated structure and layout changes. Issues relating to getting familiar with the tree structure

and dis-orientation due to structural changes. Ability to iteratively refine queries, by jumping back and forth between levels.

- Task-based testing: Testing queries of varying complexity and type. This classification of queries is general and independent of this particular *Tree-browser* design and implementation.

  1. Simple "attributes-based" queries: For example, how many states have a level of violence index <= 60 and good traffic conditions?
  2. Complex associative "structure-based" queries: For example, how many states satisfy all the following constraints: regions with standard of living >= 50 but <= 95, states with population density <= 72 and good traffic conditions, public universities with out-of-state full-time tuition <= $3000 per semester and average SAT scores of >= 1100 and at least one department.

7 tasks of the above types were used in the testing.

5.5 Results

The subjective evaluations brought out some very interesting results. Subjects agreed on most issues, and that increased our confidence in the results obtained. Several features were liked and some were disliked.

5.5.1 Most liked features

When asked what was the one thing they liked the most about the *Tree-browser*, subjects said things like:

1. "Dynamic querying and pruning to get multiple views based on current interests."
2. "Easy to learn, convenient and straight-forward to use."
3. "The ability to locate interesting parts of large trees."
4. "Easy to create complex multi-level queries."
5. "I liked seeing the tree structure of the data, which would usually be tabular."
6. "Shows adequate information."
7. "Tightly-coupled overview and detailed view."

5.5.2 Subjective ratings

Users rated 22 aspects of the *Tree-browser* on scales of 1 to 9, where 1 was the worst rating and 9 the best rating. They also rated 3 possible features. Table 5.1 gives the mean and standard deviations for each rating (sample size was 6).

ASPECT RATED	MEAN RATING	STD DEVIATION OF RATING
Ease of panning using field–of–view	7.2	1.2
Usefullness of overview	8.5	0.5
Adequacy of Browsing technique	6.7	1.8
Ease of creating / modifying query panel	8	1.5
Ease of range–slider manipulation	4.8	2.4
Speed of range–slider manipulation	5	2.3
Ease of menu manipulation	8.5	0.8
Speed of menu manipulation	8.5	0.5
Usefullness of color updates during manipulation of widgets	8.2	1
Speed of color updates during manipulation of widgets	6.3	0.8
Non–disorientation due to tree structure changes	8.2	1
Usefullness of pruning	8.7	0.5
Speed of changing levels	5.3	2.1
Usefullness of hide gray leaves feature	8.5	0.8
Ease of learning to operate the system	8.5	0.5
How satisfying?	7.8	0.8
How stimulating?	7.8	1
How easy?	7.7	0.8
Adequacy of system power	6.8	1.2
Flexibility of system	7.8	1
Speed of system	6	1.3
How wonderful?	7.8	1
Ability to click on a node to see the subtree below it	8.7	0.5
Ability to also type in query values	7.5	2.7
Usefullness of hide all gray nodes feature	8.3	1.2

Table 5.1: Subjective ratings

5.5.3 Suggestions for improvement

The following is a listing of all the suggestions for improvement made by subjects, ranked by the number of subjects who made each suggestion:

1. Ability to click on a node to open the subtree below it. Subjects were unanimous that the ability to click on a node to expand its subtree was very intuitive and wanted that feature. This can be generalized to the ability to manually specify a subset of nodes to expand, or not to expand (throw out).

2. The range-slider needs to be redesigned and re-implemented. Subjects were unanimous that manipulating the slider was a frustrating experience, especially, while trying to fine-tune the query. The redesigned slider should allow coarse adjustment and fine adjustment and also be faster.

3. Pruning and the Hide Gray Leaves feature were clearly big hits with the subjects. Most of them used the latter frequently to recapture screen space from uninteresting gray nodes and focus the search. Removing nodes like this also made the system faster. Many subjects agreed that a Hide All Gray Nodes feature would be very useful. Some subjects also felt that the default behavior could be to hide gray nodes (when levels changed) and users could ask to see gray nodes (for context) if they were interested. This suggestion should help make the views even more compact. But the prefered default behavior depends on the structure of the tree (fan-out at each level) and also the type of tasks; therefore this requires more investigation. In fact, one subject liked seeing the gray nodes at higher levels and used the Hide Gray Leaves feature only once. He said the presence of the

gray nodes at higher levels helped in visually separating disjoint groups of nodes.

4. Subjects found the response time of some *Tree-browser* features to be slow, especially increasing levels when the number of nodes was high. This was sometimes frustrating to the users. It is important to make changing levels faster, and also allow users to cancel a change level request if needed (sometimes subjects hit a change level button by mistake and then had to wait a long time before they could undo it). The cursor should also change to a busy cursor whenever the user is expected to wait. Updates of the data display as widgets or field-of-view were manipulated were for the most part fast, but sometimes were slow.

5. It was observed that subjects get somewhat disoriented when the level of the tree was changed. This is because the layout algorithm generates a fresh layout whenever the tree structure changes, i.e. whenever more or less levels are requested to be seen. It is felt that this problem can be significantly alleviated by retaining the same current focus. Users could be allowed to choose before changing levels, what part of the tree the detailed view should display. For example, if the user asks to see the university level after specifying Florida as the current focus, the new *Tree-browser* view should be initialized to show universities within Florida.

6. Ability to type in query values, especially when exact values (like violence index $<=$ 61) are to be specified. Even in the case of textual menus, if there are too many items in the list, it might be easier and faster just to type in the string.

7. Panning the detailed view by dragging the field-of-view in the overview was useful, but some improvements to the design are required. One subject emphasized the need to always fit the overview into one screen only, so that no scrolling of the overview is required. As mentioned in Section 3.4, this is what we had designed, but it wasn't enforced in this implementation. Another subject suggested that users should be able to click anywhere in the overview and have the field-of-view jump to that position. This would enable fast coarse navigation. Fine-tuning could then be accomplished by dragging the field-of-view.

8. The catchiest quote received from one of the subjects was this: "Drag-and-drop becomes a drag for experienced users, so drop it!". Some other subjects also echoed the feeling that it might be easier and faster to just replace each drop area with a menu of attributes at that level. Some other subjects enjoyed the drag-and-drop mechanism to create and modify query panels.

9. Another comment related to drags-and-drops relates to the feedback provided to show when it is OK to drop. Some subjects felt that this feedback was ambiguous and needed to be improved.

10. Ability to specify complex boolean queries involving ORs and NOTs, in addition to the ANDs allowed currently.

11. One task (Task 6) required subjects to look for departments within 3 states, Wyoming, Wisconsin and Washington. This is basically restricting states to Wyoming OR Wisconsin OR Washington. The menu allows users to select 1-of-n textual values. A widget to select m-of-n textual values needs

to be designed and implemented.

12. A couple of subjects objected to the fact that widgets at each level were initialized to impossible values sometimes. For example, even if one chose to see only the states inside USA, states of all regions would appear in the menu of state names. This was, in fact, a decision we made during the implementation. Due to the dynamic nature of the remaining nodes at each level, there would be a large run-time overhead associated with keeping the widgets always up-to-date to the range of values taken by the remaining nodes at that level (instead of range of values taken by all nodes at that level).

13. One subject wished that there was a way to search upwards in the tree, i.e. to be able to query on universities and then see the states which contained the selected universities. Another subject made an interesting suggestion, that the *Tree-browser* should allow users to hide certain levels on demand. For example, if users are interested in looking at all universities in USA, and do not care about the states they are in, it should be possible to remove the State level totally and then get it back when desired.

14. Sometimes the field-of-view goes partially off the left edge of the overview. This needs to be fixed.

15. The Attributes List should be sorted alphabetically.

16. The nodes should not be embossed, especially since they are are not clickable. The embossed effect gives users wrong visual cues. A debossed effect would be more suitable. One subject preferred having plain clickable rect-

angles, without neither embossing nor debossing.

17. Ability to do regular-expression searches on strings.

18. Querying on an internal (non-leaf) node should not make the deeper levels vanish from the screen. This was a design decision based on run-time overhead considerations and desirability of tree structure constancy during dynamic querying. It is felt that this subject was surprised at this behavior because it was not explained to him during training. None of the other subjects, who were informed beforehand of this behavior, had any problems with it.

19. Subjects sometimes tried to use the left mouse button (instead of middle) to drag-and-drop, since the left button was otherwise used for clicking. But they always quickly realized their mistake and corrected it. Thus, this is not a real problem but just a matter of getting used to the interface.

One common problem subjects had was that the sibling nodes were ordered reverse-alphabetically, i.e. Virginia appeared above Maryland, instead of below it. However, this order is not controlled by the *Tree-browser*, which merely displays all nodes in the order in which they are specified in the input data file (provided by the application). But this is nevertheless an interesting guideline for application programmers to follow when they prepare the *Tree-browser* input data file.

5.6 Discussion

The Usability Testing provided us with a good insight of what users would like to see in Tree-browsers and in dynamic query interfaces.. Our findings can be summarized as follows:

- *Tree-browser* design refinements: In addition to the ability to specify selection subsets on the basis of attributes, which the users found very useful, users should also be able to specify selection subsets manually: by clicking, marque rectangles etc. Users should also be able to manually override subsets specified via attributes-based queries, e.g. all universities with SAT scores more than 1100, but put the University of Maryland in anyways. Being able to remove certain nodes manually is also required. The user might be able to perform these specifications using ORs and NOTs, or might prefer to do it manually by clicking on nodes. But this would lead to problems of keeping track of these nodes, and overriding the original query might lead to complications later in the query process, if the design is not done carefully.

  The default behavior of the *Tree-browser* could be to hide all gray leaves whenever levels are increased. The user would then be able to see gray nodes at higher levels (for context) upon request. The *Tree-browser* should attempt to retain the current focus of the user while changing levels, so that dis-orientation is minimized.

  It should be possible to click anywhere in the overview (which always shows everything) and have the field-of-view jump to that location. This would allow fast coarse navigation and then users could fine-tune their searches by dragging the field-of-view.

- General GUI refinements: Users want GUI systems to be fast. At the same time, they want to have flexibility in their interactions. For example, they want to be able to specify ranges of numbers using range-sliders as well as by typing in values. The range-slider needs to be redesigned to allow coarse and fine adjustment and needs to be re-implemented to be faster. A m-of-n textual list selection widget needs to be designed and developed.

5.7 A controlled experiment

With my guidance, three students, Robert Ross, Zhijun Zhang and Eun-Mi Choi, conducted a controlled experiment to compare 3 behavior alternatives for the *Tree-browser* [RZC94]. These behavior alternatives related to pruning subtrees of nodes that did not match the query at their own levels. The 3 treatments were as follows: (two of these were discussed in Section 3.5)

1. Subtrees of unselected nodes are shown in their entirety, but are colored gray.
2. Subtrees of unselected nodes are pruned out but the nodes themselves are shown in gray (This is the behavior option that the *Tree-browser* currently uses).
3. Subtrees of unselected nodes and the unselected nodes themselves are pruned out.

The UniversityFinder database (Section 3.3.2) was used in this experiment.

The authors hypothesized that subjects using interface 1 would take longer than those using 2 or 3. Although 1 allows for a static display of the tree, it's inclusion of all irrelevant nodes would probably slow down the response time, add too much useless information to the user's visual field, and create a need for excessive scrolling/panning of the displays to get to relevant nodes. Due to the longer task completion times and more difficult/ complex searching required, they postulated that users would also have lower subjective ratings for interface 1 than for 2 or 3.

They hypothesized that the completion times and subjective ratings for interfaces 2 and 3 would be approximately equal (differences would not be significant). It was postulated that interface 3 promised the most compact views, while 2 offered additional context feedback that might help in tasks requiring several iterations (to refine the query).

24 subjects were randomly assigned to use one (and only one) of these three treatments (between-groups experimental design). They performed a set of seven standard tasks. Afterwards, they filled out an electronic questionnaire (QUIS) to subjectively evaluate the treatment that they had used.

The results showed that the times to complete the set of seven tasks were significantly different at the 0.05 level with interface 1 being slower than 2, which was slower than 3. In terms of subjective satisfaction, 2 rated higher than 3, which was preferred to interface 1.

For total task completion time, an ANOVA gave $F = 25.2$ which was significant at the 0.01 level. Then, using pairwise t-tests, it was found that all results were significant at the 0.05 level. Therefore, in this controlled experiment, 3 was the fastest interface while 1 was the slowest. The differences between 1 and the other two interfaces were also significant at the 0.01 level, but the differences between 2 and 3 were not. For subjective ratings also, the ANOVA produced an F that was significant at the 0.01 level. Also, all of the pairwise t-tests were significant at the 0.05 level. Users liked interface 2 the best and disliked 1 the most. Differences between 1 and 2 and between 1 and 3 were also significant at the 0.01 level although differences between 2 and 3 were not. In addition to the overall times for each interface, the authors also compared the times to complete each task.

The authors note that the poor performance times of subjects using interface 1 might be partly due to poor system response times. Specifically, interface 1 had consistently (noticeably) slower response times, especially when expanding to the lower levels ("University" and "Department" levels).

The differences between interfaces 2 and 3 were not as clear cut; task performance was faster with interface 3 but interface 2 got higher subjective ratings. I feel that the authors failed to consider one important factor while discussing the differences in performance between interfaces 2 and 3, i.e. the fan-out. Specifically, the UniversityFinder tree has a large fan-out (50 states in USA) from the Region level to the State level, and this might help explain why interface 3 had faster performance times than interface 2 (a significant proportion of states were gray and thus took up a significant proportion of screen space). It would be interesting to repeat the experiment with trees of varying fan-out.

The authors conclude that interface 1 is best avoided, and that the system should ideally allow users to switch between behavior alternatives 2 and 3.

Chapter 6

Conclusions

This chapter highlights the major contributions of this work and suggests possible future directions.

6.1 Contributions

The major contributions of this work are as follows:

- Extension of dynamic queries to hierarchical data sets: Using the parent-children interrelationship between nodes to facilitate browsing of the HDS,
- Dynamic query environments: A more flexible and powerful mechanism to specify selection subsets of large data sets.

The concepts of dynamic querying and pruning are general enough that they can be applied effectively to other existing tree visualizations like treemaps and Cone-Trees. These concepts are also extensible to graph structures, but that would require careful thinking and design.

6.2 Future Directions

In this section, future directions are suggested. These include direct extensions to our *Tree-browser* and also other open tree-browsing research issues.

6.2.1 Extensions to our work

- *Tree-browser* refinements:

  1. Ability to manually add or remove 1 or more nodes to the selection subset resulting from the dynamic queries. Thus, the ideal subset specification mechanism would allow both query-based specification as well as manual specification. Some issues relating to keeping track of these nodes and avoiding complications later in the query process need to be addressed.
  2. Query specification: The current *Tree-browser* implementation allows users to specify an AND of queries on up to any 3 attributes. This flexibility should be sufficient for many applications, but it would be better to have an interface that would allow any number of ANDs, ORs and NOTs.
  3. The *Tree-browser* interface has been "fine-tuned" for a tree of depth 5 (See Section 3.4 for details). The interface could be extended to cope with varying structure and growing size by following the suggestions in section 3.6.1 or otherwise.
  4. Further study of which approach is better in terms of hiding vs. showing gray nodes (at higher levels) by default is needed (to extend the work of [RZC94]). This investigation should take into consideration

different types of tasks, applications and tree structures (fan-out, depth, breadth, size) and attempt to identify when either approach is better.

- General GUI extensions: Newer widgets need to be developed to enable users to specify multiple selections on textual attributes easily. Extending the concept of a 2-box slider to a n-box slider-cum-legend would enable users to specify visual queries on several ranges of numerical attributes.

- Extension to Graph Structures: The *Tree-browser* extended dynamic queries to one class of non-flat data sets, i.e. hierarchical data sets. The most general form of any data set, is the *arbitrary graph.* Graphs have nodes connected via an arbitrary number of links of different types. Each link represents a relationship between (among) the connected nodes. We believe that *Tree-browser* concepts of dynamic querying and pruning / selective growing can be extended to graph structures as well. Instead of distinct levels in the hierarchy, we would then think of distinct classes of nodes (and even links). Implementing a Graph-browser that extends the concepts of our *Tree-browser* would help to come up with a more complete theory of browsing graphs that what exists today.

6.2.2 Other open tree-browsing research issues

- Semantics-based browsing: Generic 2D browsers [PCS95] treat the information space being browsed as images only. We believe that browsing of trees can be facilitated by taking advantage of the underlying structure of the tree. Fast navigation between siblings, up to parents and grand-parents etc., without having to manually scroll and pan would be useful options. Traversal of the tree in preorder, postorder and inorder, and tours of nodes marked either manually or by a query are interesting topics for investigation.

- Layout issues: As the size and complexity of a tree increases, the problem of visualizing it effectively becomes more and more challenging. If tightly-coupled detailed views and overviews are used, there is a clear need for layout guidelines. As the aspect ratio (fan-in / fan-out) of a tree varies, new layout strategies are needed. For example, if the tree is wider than deep (as was the case in the network management and University-Finder scenarios) then it makes sense to have the tree drawn from left-to-right (or right-to-left) and the overview to the left (or right) of the detailed view. On the other hand, if the tree is deeper than it is wide, then it might be better to have the tree drawn from top to bottom and the overview below the detailed view. If the tree structure changes frequently and has to be redrawn, then it might be a good strategy to utilize the overview optimally by making the remaining tree occupy the entire overview space. But this might lead to some dis-orientation, as the zoom ratio will keep varying.

Bibliography

[AS94] C. Ahlberg and B. Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 313–317, ACM, New York, 1994.

[AWS92] C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic queries for information exploration: An implementation and evaluation. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 619–626, ACM, New York, 1992.

[BETT89] G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for drawing graphs: An annotated bibliography. Unpublished Technical Report, Computer Science Department, Brown University, October 1989.

[BI90] David V. Beard and John Q. Walker II. Navigational techniques to improve the display of large two-dimensional spaces. *Behavior & Information Technology*, 9(6):451–466, 1990.

[BMMS91] A. Buja, J.A. McDonald, J. Michalak, and W. Stuetzle. Interactive data visualization using focusing and linking. In *Proceedings of the*

IEEE Visualization '91 Conference, pages 156–163. IEEE Computer Society Press, October 1991.

[CGPZ93] M.H. Chignell, G. Golovchinsky, F. Poblete, and S. Zuberec. Information visualization and interactive querying for online documentation and electronic books. In *Proceedings of CASCON '93, IBM Canada Center for Advanced Studies*, pages 1011–1020, October 1993.

[CJK+94] David A. Carr, Ninad Jog, Harsha P. Kumar, Marko Teittinen, and Christopher Ahlberg. Using interation object graphs to specify and develop graphical widgets. Technical Report CS-TR-3344, Department of Computer Science, University of Maryland, College Park, Maryland, September 1994.

[CS94] Richard Chimera and Ben Shneiderman. An exploratory evaluation of three interfaces for browsing hierarchical tables of contents. *ACM Transactions on Information Systems*, 1994.

[CZP93] M. Chignell, S. Zuberec, and F. Poblete. An exploration in the design space of three dimensional hierarchies. In *Proceedings of the Human Factors Society*, Santa Monica, California, 1993.

[DM90] Chen Ding and Prabhaker Mateti. A framework for the automated drawing of data structure diagrams. *IEEE Transactions on Software Engineering*, 16(5):543–557, May 1990.

[Fur86] George W. Furnas. Generalized fisheye views. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 16–23, ACM, New York, 1986.

[Ged88] D. Gedye. Browsing the tangled web. Master's thesis, University of California at Berkeley, May 1988.

[HCMM89] J. G. Hollands, T. T. Carey, M. L. Matthews, and C. A. McCann. Presenting a graphical network: A comparison of performance using fisheye and scrolling views. In G. Salvendy and M. J. Smith, editors, *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, pages 313–320. Elsevier Science, Amsterdam, 1989.

[Hen92] Tyson Henry. *Interactive Graph Layout: the Exploration of Large Graphs.* PhD thesis, University of Arizona, Tucson, Arizona, May 1992.

[HH91] Tyson R. Henry and Scott E. Hudson. Interactive graph layout. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 55–64, ACM, New York, 1991.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the IEEE Conference on Visualization*, pages 284–291, October 1991.

[JS94] Ninad Jog and Ben Shneiderman. User controlled smooth zooming for information visualization with a starfield display. Technical Report CS-TR-3286, Department of Computer Science, University of Maryland, College Park, Maryland, June 1994.

[JT92] W.A. Jungmeister and D. Turo. Adapting treemaps to stock portfolio visualization. Technical Report CS-TR-2996, Department of Computer Science, University of Maryland, College Park, Maryland, 1992.

[KPTS94] Harsha P. Kumar, Catherine Plaisant, Marko Teittinen, and Ben Shneiderman. Visual information management for network configuration. Technical Report ISR-TR-94-45, Institute for Systems Research, University of Maryland, College Park, Maryland, June 1994.

[Kum95] Harsha Kumar. Visualizing hierarchical data with dynamic queries and pruning – the tree-browser. In Catherine Plaisant, editor, *HCIL Open House '95 Video*. hcil, June 1995.

[Mas92] Toshiyuki Masui. Graphic object layout with interactive genetic algorithms. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 74–80. IEEE Computer Society Press, September 1992.

[Moe90] Sven Moen. Drawing dynamic trees. *IEEE Software*, 6(3):21–28, July 1990.

[MRH91] Eli B. Messinger, Lawerence A. Rowe, and Robert R. Henry. A divide-and-conquer algorithm for the automatic layout of large directed graphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):1–12, January 1991.

[PCH92] C. Plaisant, D. Carr, and H. Hasegawa. When an intermediate view matters – a 2d-browser experiment. Technical Report CS-TR-2980, Department of Computer Science, University of Maryland, College Park, Maryland, October 1992.

[PCS95] C. Plaisant, D. Carr, and B. Shneiderman. Image browsers: Taxonomy, guidelines, and informal specifications. *IEEE Software*, 1995. to appear.

[Pla93] C. Plaisant. Facilitating data exploration: Dynamic queries on a health statistics map. In *Proceedings of the 1993 American Statistical Association conference*, pages 18–23, August 1993.

[RD88] Gerald M. Radack and Tejas Desai. Akrti: A system for drawing data structures. *IEEE Languages and Automation*, pages 116–120, 1988.

[Rit91] Ken Ritchie. A general method for visualizing abstract structures. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 1237–1246, October 1991.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 189–194, ACM, New York, 1991.

[RT81] E.M. Reingold and J.S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, March 1981.

[RZC94] Robert Ross, Zhijun Zhang, and Eun-Mi Choi. Using the tree-browser to visualize information for database queries. Term project for CMSC 434, December 1994.

[SB92] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 83–91, ACM, New York, May 1992.

[Shn92] B. Shneiderman. Tree visualization with treemaps: 2-d space-filling approach. *ACM Transaction on Graphics*, 11(1):92–99, January 1992.

[Shn94] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, 1994.

[SM91] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, July 1991.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, February 1981.

[SZB+92] Doug Schaffer, Zhengping Zuo, Lyn Bartram, John Dill, Shell Dubs, Saul Greenberg, and Mark Roseman. Comparing fisheye and full-zoom techniques for navigation of hierarchically clustered networks. Technical Report 92/491/29, Department of Computer Science, University of Calgary, November 1992. 12 pages.

[WF94] Colin Ware and Glenn Franck. Viewing a graph in a virtual reality display is three times as good as a 2d diagram. In *1994 conference on Visual Languages*, pages 182–183, October 1994.

[WS79] C. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE-5(9):514–520, September 1979.

[WS92] C. Williamson and B. Shneiderman. The dynamic homefinder: Evaluating dynamic queries in a real-estate information exploration system. In *Proceedings of the 15th Annual ACM SIGIR conference*, pages 338–446, ACM, New York, 1992.

[YS93] Degi Young and Ben Shneiderman. A graphical filter/flow representation of boolean queries: A prototype implementation and evaluation. *Journal of the American Society for Information Science*, 44(6):327–339, February 1993.