Partitioned Rendering Infrastructure for Stable Accordion Navigation

by

James Gerald Alphonso Slack

B.Sc., University of British Columbia, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

Computer Science

We accept this thesis as conforming
to the required standard

The University of British Columbia

April 2005

Abstract

My thesis presents a new rendering infrastructure for information visualization applications that use the accordion drawing navigation metaphor. Accordion drawing techniques use rubber-sheet navigation methods, with the borders tacked down, and provide guaranteed visibility for marked areas of interest.

Our accordion drawing algorithms are based on screen-space partitioning, which eliminates overculling and tightly bounds overdrawing. By eliminating the overculling effects of rendering dense regions of data, we guarantee a correct visual representation of any dataset. Also, our pixel-based drawing infrastructure improves the rendering performance of dense dataset regions with strict drawing constraints, which are based on application-specific drawing requirements. The generic infrastructure provides an interface to numerically stable navigation of datasets, with full support for multiple concurrent regions of navigation motion.

To evaluate our generic infrastructure, I benchmark our tree comparison application with the performance of TreeJuxtaposer, a previous accordion drawing application with identical features. I describe our tree traversal algorithms, which we use for efficient rendering, culling, and layout of tree datasets. I also discuss tree node marking techniques, which offer several improvements over previous range storage and retrieval techniques, reducing memory requirements and increasing rendering speed. Finally, I evaluate tree-specific navigation techniques from our winning entry in the InfoVis 2003 contest, with TreeJuxtaposer supported by an incremental search feature and an improved user interface.

Contents

Abstract ii

List of Figures vi

Notation x

Acknowledgements xi

1 Introduction 1

1.1 Motivation 1

1.2 Information visualization techniques 5

1.2.1 Guaranteed visibility 5

1.2.2 Focus+Context 8

1.2.3 Progressive rendering 9

1.3 Thesis contributions 10

1.3.1 Accordion drawing contributions 10

1.3.2 TJ2 contributions 10

1.3.3 TreeJuxtaposer Evaluation from InfoVis 2003 Contest entry . 11

1.4 Thesis Organization 11

2 Related Work 12

2.1 Visualization, interaction, and perception 12

2.2 Phylogenetic tools and tree visualization 21

3 TJ2 **24**

3.1 Node layout . . . 25

3.1.1 Mapping nodes to grid . . . 27

3.1.2 Placing horizontal node edges . . . 29

3.2 Rendering trees . . . 33

3.2.1 Node seeding . . . 34

3.2.2 Drawing trees . . . 35

3.3 Marked ranges . . . 46

3.3.1 Marked ranges in TJ1 . . . 47

3.3.2 Marked ranges in TJ2 . . . 50

3.4 Topological picking . . . 54

4 Accordion Drawing **59**

4.1 Split line infrastructure . . . 62

4.1.1 Abstracting split lines . . . 65

4.1.2 Separate horizontal and vertical split lines . . . 66

4.1.3 Tree hierarchy for split lines . . . 68

4.2 Generic AD rendering infrastructure . . . 70

4.2.1 Partitioning stage . . . 71

4.2.2 Seeding stage . . . 72

4.2.3 Drawing stage . . . 72

4.3 AD navigation . . . 73

4.3.1 Moving one split line . . . 74

4.3.2 Moving several split lines . . . 79

5 Evaluation and Discussion **83**

5.1 Preprocessing . . . 85

5.2 Scene rendering . . . 88

5.3 Memory usage . . . 93

5.4 Marking efficiency 95
5.5 Evaluation summary 96

6 Future Work and Conclusions **98**
6.1 Future work 98
6.2 Conclusions 99

Glossary **101**

Bibliography **107**

Appendix A TreeJuxtaposer Task Evaluation **112**
A.1 Contest dataset 113
A.2 User interface 114
A.3 Incremental search 117
A.4 Contest results 122
A.4.1 Tasks suited for TJ1-contest 122
A.4.2 Tasks not suited for TJ1-contest 163
A.5 Contest conclusions 170

List of Figures

1.1 A rectilinear, right-aligned tree 2
1.2 Comparison of two trees 3
1.3 Accordion navigation 4
1.4 Highlighted data has priority, context is important 6

2.1 The SeeSoft software analysis tool 13
2.2 The document lens visualization tool 15
2.3 The H3Viewer visualization application 16
2.4 The Pad++ system 17
2.5 The SequenceJuxtaposer application 19
2.6 The PowerSetViewer application 20
2.7 The TreeWiz application 22

3.1 Naming conventions for edges and directions 26
3.2 *doGridding* function 27
3.3 Gridding: example subtree 28
3.4 Tree placed in grid 28
3.5 Horizontal edge placement can be anywhere 30
3.6 Relative placement of horizontal node edges in cells 31
3.7 Leaf range width less than block width 38
3.8 Leaf range width less than half-block width 39
3.9 Ascent rendering horizontal gaps 40
3.10 Finding highest subtree in a leaf range 41

3.11 *ascentRender* function 42
3.12 Half-block gaps in leaf ancestors 44
3.13 Rendering problems with ascent width less than segment width . . . 45
3.14 Example node key assignment in a small tree 46
3.15 TJ1 mark storage with indirect marks 49
3.16 Incorrect indirect marking using simple methods 52
3.17 Direct marking of indirectly marked nodes 53
3.18 *Picking* function 56

4.1 Initial uniform split line layout 61
4.2 Vertical stretch of interaction box 62
4.3 Interaction box stretch in both directions 63
4.4 A Quad-cell, from Accordion Drawing in TJ1 64
4.5 Separate horizontal and vertical sets of split lines form grid 67
4.6 Split lines stored in a balanced binary tree hierarchy 68
4.7 Split lines boundaries expressed as lines and regions 69
4.8 *AbsolutePosition* function 70
4.9 *moveSingleSplitLine* function 75
4.10 Absolute distances move with split line 76
4.11 Labeled regions for single split line movements 78
4.12 *moveSplitLineSet* function 81
4.13 The three cases of function *moveSplitLineSet* 82

5.1 Parsing time 86
5.2 Preprocessing time 87
5.3 Rendering time performance for TJ2 89
5.4 Number of nodes rendered for TJ1 and TJ2 90
5.5 Average time to render a node in TJ1 and TJ2 92
5.6 Memory performance of TJ1 and TJ2 94

5.7 Table of marking performance of TJ1 versus TJ2 95

A.1 TJ1 User Interface 115
A.2 TJ1-contest User Interface 116
A.3 TJ1-contest `Groups` panel 117
A.4 TJ1-contest `Settings` panel 118
A.5 TJ1 drop-down selection box 119
A.6 TJ1-contest `Found` panel 121
A.7 Differences marked in mammalia trees, with Latin names 124
A.8 Differences marked in hcil trees 125
A.9 Differences marked in phylo trees 125
A.10 Differences of genus *pteropus* and family *pitheciidae* 126
A.11 Differences of directories *counterpoint* and *iv03contest* 127
A.12 Movements of *cebidae* and *pitheciidae*. 128
A.13 Similar topological properties in both phylo trees 129
A.14 Path between *animalia* and *homo sapiens* in animalia_A 130
A.15 The *users* subtree marked in logs_A, showing subtree fan-out 132
A.16 *mammals* and *bony fishes* marked in animalia_B 134
A.17 Result of *giraffe* search in animalia_B 136
A.18 Result of browsing for cute animals in animalia_B 139
A.19 Top three topological similarities marked in phylo trees 141
A.20 The *rodentia* subtree marked in both mammalia trees 143
A.21 The animalia_A tree with common node names 144
A.22 Result of *dolphin* search in animalia_A 145
A.23 Result of *horse* search in animalia_A 146
A.24 *marmots* subtree expansion in mammalia trees, common names . . . 148
A.25 *marmota* subtree expansion in mammalia trees, Latin names 148
A.26 Differences marked in mammalia trees, with common names 149
A.27 Result of *Townsend* search in animalia_A, with Latin names 150

A.28 Result of *Townsend* search in $\mathsf{animalia}_\mathsf{A}$, with common names 151
A.29 *spirulida* subtree expansion in $\mathsf{animalia}_\mathsf{A}$, Latin names 153
A.30 logs_A file system tree 155
A.31 *building* subtrees and *shankar* subtrees marked for comparison . . . 156
A.32 *class* subtree expanded in logs_A 158
A.33 *project* subtree expanded in logs_A 159
A.34 *users* subtree expanded in comparison of logs_A and logs_B 160
A.35 Differences in *cmsc434-0101* between logs_A and logs_B 161
A.36 Differences in *cmsc838p* between logs_A and logs_B 161
A.37 Differences show new courses added between logs_A and logs_B 162
A.38 Differences in *jazz-chat* directory between logs_A and logs_B 163
A.39 Differences in *counterpoint* among all hcil trees 164
A.40 Differences in *iv03contest* among all hcil trees 164
A.41 Differences in *spacetree* and *timesearcher* among all hcil trees 165

Notation

In this document I use the first person when referring to work done either entirely or primarily by myself, and the third person when referring to collaborative work with colleagues.

Also, this thesis uses several different typeface conventions that are used to convey meaning, when applicable. No type-faces are combined since each type convention may only apply to at most one special word. Each type is listed here for convenience:

- An occurrence of a glossary entry is in **Roman, upright, bold** type. The glossary will only contain a page reference to either the first use of an entry or to a meaningful place where the entry is defined in a meaningful context, and that occurrence will be **in this type**.
- The names of dataset names are in sans serif, upright, medium type. Dataset names used for the InfoVis 2003 contest are defined in Section A.1. Similar datasets, such as $\mathsf{animalia}_{\mathsf{A}}$ and $\mathsf{animalia}_{\mathsf{B}}$, are differentiated by lowered type from animalia, which refers to the datasets in a comparison environment.
- The specific names of dataset nodes are in *Roman, italic, medium* type. Nodes may be: scientific, Latin species names, such as *marmota*; common, English species names, such as *marmots*; or file system names, such as *iv03contest*.
- The descriptive names of components of TreeJuxtaposer applications are in `typewriter, upright, medium` type. The components are the menu options and panels of TreeJuxtaposer; panels, such as `Settings` can be used by accessing them through the menu options, such as `Tools`. Other named options within panels are also `in this type`.

Acknowledgements

I'd like to keep this short, since my thesis seems so very long and I could very well go on for pages and pages about stuff... so, for everyone who I've missed, it's not out of spite, it's an indication that we need to hang out more.

First, here's a randomized list of grads and other students, many of whom are in a boat similar to mine, that I've run into in Imager while starting my thesis: Qiang Kong; Vlad Kraevoy; Fred Kimberley; Ciarán Llachlan Leavitt; Ken Alton; Ben Forsyth; Chen Yang; Matt Trentacoste; Kristian Hildebrand; Lewis Johnson; Andrew Chan; Dan Archambault; Jason Harrison; Dave Westrom; Abhijeet Ghosh; Dave Burke; Dmitry Nekrasovski; Simon Clavet; Hamish Carr; Adam Bodnar; Yongying Zhu; Melanie Tory; Heidi Lam; Mark Hancock; and Matt Williams. Not all that happens in Imager is fun and games, but my lab-mates of Imager have been a never-ending source of entertainment, research ideas, coffee breaks, runs to the village for Curry Point, and the occasional multiplayer slaughter-fest.

Second, the all important list of colleagues I've worked with on papers and other conference submissions, whom I have worked with night and day, yet still don't understand my sleeping schedule: Tamara Munzner; Kristian Hildebrand; Ciarán Llachlan Leavitt; Katherine St. John; Qiang Kong; and François Guimbrètiere. For the most part, this thesis either refers to work done mostly by me, in the first person, or with others who were forced to work with me. A large amount of our work in analysis of accordion drawing methods, future work considerations, and other tree rendering tricks, I attribute to collaborative work with Kristian Hildebrand, who

always provides the necessary encouragement for me to GBTW on a regular basis. Of course, my patient supervisor, Tamara Munzner, has been super supportive of many of my crazy ideas, shows interest in a majority of my not-so-well-thought-out algorithms, and continues to help me improve in all those pesky areas I need improvement on, which is saying *a lot*.

Finally, but of course most importantly, I must give the biggest shout-out to my family, who are always on my mind when they least expect it. First, my very understanding parents, Cliff and Kathy Slack, who deserve much more than a simple thanks, I've dedicated this thesis to you (it's on the next page) for many good reasons. Hopefully you'll enjoy reading it as much as I enjoyed writing it, perhaps even more. Next, my far away, but never forgotten, sister Erica and my brother-in-law Stu Morgan, who make a 20 hour flight worthwhile, even when it doesn't quite make it to its destination. I'll be sure to see you in Rarotonga in the near future, to, um, check on your palm tree, which must be hundreds of feet tall by now. My traveling buddy, and otherwise outstanding grandma Ella MacNeil: I think I'm way overdue for a visit, and not just for the pies I've been missing out on. How about some bingo? Finally, the sudden passing of my grandma Edith McGinnis last year was a sad time for me, and the closeness of our family has remained strong throughout all we've been through since then. Although we only get together a few times every year, I always look forward to seeing anyone I call family again.

James Gerald Alphonso Slack

The University of British Columbia
April 2005

This is for my parents

Chapter 1

Introduction

My thesis presents two key contributions to the field of information visualization: a generic infrastructure for accordion drawing as a malleable two-dimensional surface; and new rendering techniques for tree visualization on accordion drawing surfaces. Our generic accordion drawing infrastructure accommodates any dataset that can be laid out and meaningfully partitioned into smaller objects on a grid structure, such as the rectilinear, right-aligned tree shown in Figure 1.1. Applications provide the bidirectional mapping from a grid surface to objects in the dataset, and our infrastructure supports the key operations of rendering, mouse-over picking, and navigation, while guaranteeing the visibility of marked data. My tree rendering algorithms for accordion drawing surfaces involve operations which perform according to the number of pixels used to display a scene, not on the input dataset. With support from the generic rendering infrastructure, I provide efficient tree-based algorithms for: tree-to-grid mapping, tree-rendering traversal, node marking with guaranteed visibility, and accurate picking on the tree.

1.1 Motivation

Visualization of datasets with more data than available on-screen pixels is a challenging problem. Without information visualization techniques, the number of on-screen pixels limits the amount of displayable data. For example, recent advances in phy-

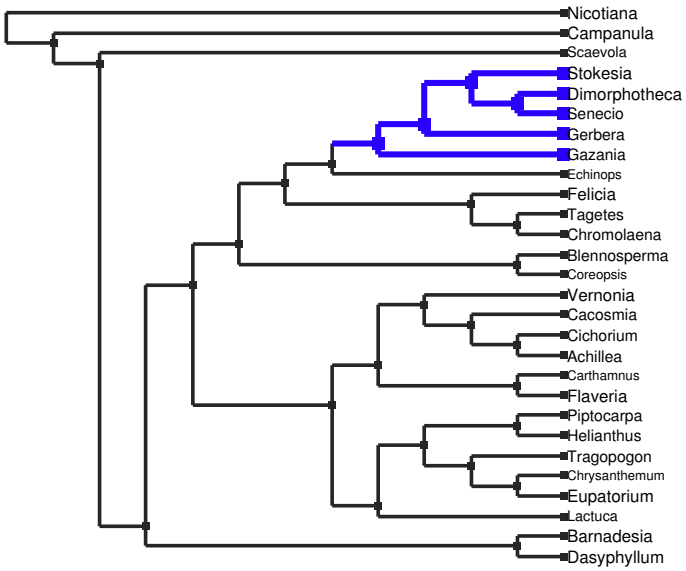

Figure 1.1: A rectilinear, right-aligned tree layout descends a tree dataset topology horizontally and aligns all terminal nodes, or leaves, to the right boundary. Our generic accordion drawing infrastructure supports this orthogonal grid-based tree layout.

logenetics, the study of evolutionary relationships between organisms, produce very large tree dataset topologies with a set of organisms at the terminal nodes. Since any two methods of constructing phylogenetic trees may produce two structurally different topologies for the same set of organisms, evolutionary biologists use many techniques to investigate structural similarities between pairs of tree datasets. The primary focus of my thesis investigates our interactive visualization infrastructure for stable accordion drawing navigation, with an example application capable of comparing large tree datasets, such as phylogenetic trees, of up to two million tree nodes.

TreeJuxtaposer (TJ1)[24] is an information visualization application used to navigate and compare several rectilinear trees, such as the pair of trees in Figure 1.2, by using two important properties: rubber-sheet navigation and guaranteed visibility. Rubber-sheet navigation is the metaphor we use to describe how users

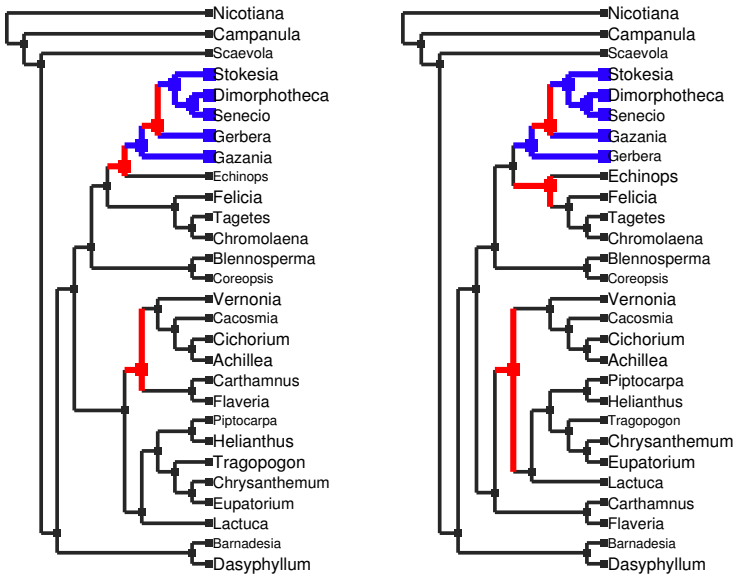

Figure 1.2: TreeJuxtaposer is capable of comparing two trees, side-by-side as shown in this figure. Regions of structural difference are marked in red and other marks, such as the blue subtree, are user-defined.

interact with the data. Users stretch and shrink regions of **accordion drawing (AD)** surfaces, the subclass of rubber-sheet navigation upon which TJ1 is built, much like the trees are drawn on a malleable rubber-sheet, as shown in Figure 1.3. The AD rubber-sheet has its borders tacked down so the entire dataset remains visible at potentially many different levels of magnification.

Guaranteed visibility is another necessary property of AD surfaces that ensures important data will remain visible at all times on its malleable surface. Marked objects on AD surfaces move with other objects when the surface undergoes movements and may be squished and stretched like any other dataset object. Guaranteed visibility implies that marked objects never shrink out of sight.

TJ1 introduces AD surfaces, but its implementation of AD is only for datasets that are tree-specific, has navigation stability problems for complex movements, and does not allow other application domains to use its AD infrastructure. The primary

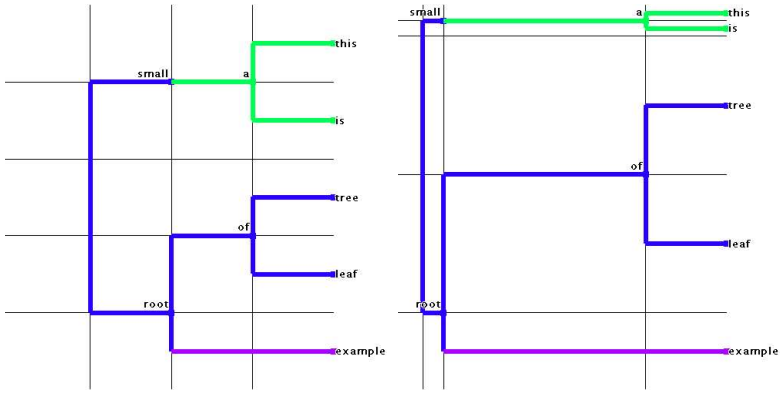

Figure 1.3: Accordion navigation works by distorting a two-dimensional surface, by stretching and shrinking regions, to allocate more screen space for regions of interest. In the left image, a small tree appears undistorted, with no regions stretched. The right image shows the same tree topology with a stretched region, which squishes other regions such as the green subtree.

goal of this thesis is to introduce a new type of AD infrastructure that allows any new application domain to render a scene using a more efficient, stable approach to rubber-sheet navigation. Furthermore, this thesis describes several new tree-specific functions to support TJ1 functionality on a generic AD surface, including: correct tree rendering, efficient marking with guaranteed visibility properties, efficient node layout on a partitioned grid structure, and accurate picking of tree nodes on the grid. This thesis presents several techniques for improving the rendering quality, rendering speed, and memory usage of TJ1 with **TJ2**, our new AD application that is functionally, visibly equivalent to TJ1.

Although current graphical processors are capable of rendering billions of pixels per second, a standard-sized monitor with a commodity video card is not capable of refreshing the display during animated transitions with a brute-force method of drawing every node.Our new infrastructure provides a rendering framework with an efficiency that does not depend on the size or structure of the dataset, but on the number of pixels on-screen. Application-specific algorithms interface with our

AD infrastructure, which abstracts away the details of partitioning, grid structure, and navigation, to allow development of new AD applications that access the infrastructure with generic grid algorithms. With support for dataset sizes beyond the number of pixels available on-screen, AD techniques allow for rapid-prototyping of new applications that render datasets of well over one million items with smooth animated transitions.

1.2 Information visualization techniques

In my thesis, I employ several information visualization techniques, including: guaranteed visibility, Focus+Context, and progressive rendering. These techniques allow users of TreeJuxtaposer, or accordion drawing (AD) applications in general, to better understand large and complex datasets, locate important information, and smoothly navigate large amounts of information without getting lost. In this section, I discuss the properties of guaranteed visibility in Section 1.2.1, Focus+Context in Section 1.2.2, and progressive rendering in Section 1.2.3.

1.2.1 Guaranteed visibility

Guaranteed visibility is a property, first introduced by TreeJuxtaposer [24], used by information visualizations to ensure important data is always visible. Applications developed with our AD infrastructure have two classes of data: **normal** and **marked**. We mark data that we consider important with object marking, either through direct interactive selections or results from computed functions. Other normal data objects provide context and overall dataset structure. Guaranteed visibility of marked data means that marked objects always take precedence over normal objects when rendering views of datasets, meaning that marks are always guaranteed to be visible, at the expense of unmarked regions of data, if need be.

AD applications provide two variations of guaranteed visibility: **static** and **progressive**. The former is used to display rendered scenes with guaranteed display

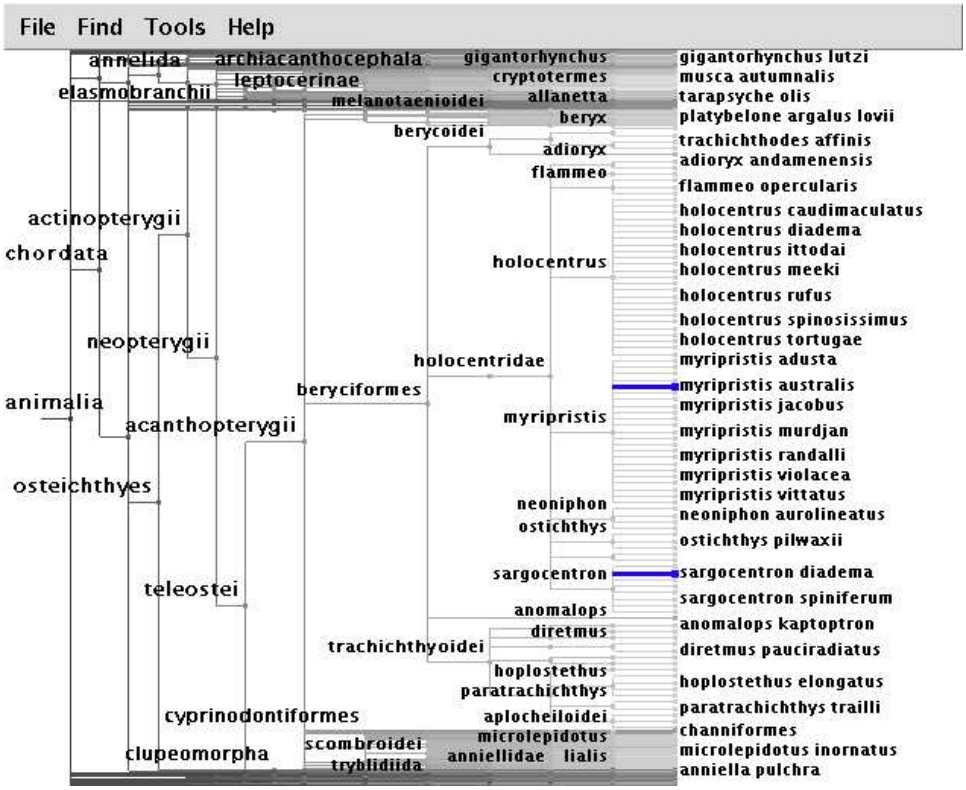

Figure 1.4: The user-selected data in this TreeJuxtaposer figure, *sargocentron diadema* and *myripristis australis* are marked in blue. However, the locations of these tree nodes are not understood without the location of many other nodes, such as *sargocentron spiniferum*, which provide context for the guaranteed visible, marked data.

of all marks, and the latter guarantees marks appear first during transitional frames of an animation. Static guaranteed visibility is a property of an application that is capable of displaying marked data with a higher priority than normal data. Static visibility properties are essential when visualizing a large dataset, where marked data might be occluded by surrounding data in a rendered scene. However, the surrounding data is still important enough to consider drawing; guaranteeing to show marked data within the context of un-occluded, peripheral data provides important visual landmarks. For example, in Figure 1.4, we mark species *sargocentron diadema*

and *myripristis australis* as important tree nodes. TreeJuxtaposer may still draw species *sargocentron spiniferum*, which should be close to *sargocentron diadema*.

AD applications cull regions of datasets when a region is shrunk smaller than the size of a drawable unit, typically a hardware monitor pixel. To draw in dense regions of objects, the application determines what to draw using either object aggregation or object selection, from a set of culling objects. For large dataset visualizations on high-resolution monitors, such as the IBM T221 with over 200 pixels per inch, a pixel-sized feature is sometimes too small to be useful. Our AD infrastructure provides novel methods of producing a lower resolution visualization, using our culling techniques. If we cull regions at larger than pixel sizes, we effectively draw less information on the display, but may reveal patterns with aggregation methods; high-level dataset features may be clearer with a larger minimum feature size. We call the minimum feature sized cells for an application **blocks**, where feature sizes are integer multiples of a pixel.

To solve static view visibility problems, AD rendering methods must examine an application-specific culled region for marks that we guarantee to be visible. Accordingly, marked data points must be stored such that regions of the dataset topology can be examined quickly during a rendering step. Section 3.3 provides more details about the compact representation and storage methods of the dataset topology as ranges of nodes for TJ2.

Guaranteed visibility is also important to consider during the rendering phase of animated transitions. Progressive guaranteed visibility is a component of progressive rendering, discussed in Section 1.2.3, that renders the most important interesting nodes before rendering the rest of the dataset. This type of guaranteed visibility uses a drawing order that favors marked data over all other data, providing landmarks when rendering navigation and animation frames for large visualizations.

As an example of progressive guaranteed visibility, consider marked nodes in TJ2. For the first frame in its scene rendering phase, TJ2 draws the roots of marked

subtrees, the path from that subtree to the root of the tree, and a path from the subtree root to one of its leaves. Rendering an overview of the dataset that is formed by this first rendering step, which produces a skeleton of useful topological features, is exponentially faster, in the common case, than methods in TJ1 that rendered entire marked subtrees. This progressive guaranteed visibility is useful in TJ2 since the location of marked subtrees is quickly represented and the marks are visible as landmarks during animated transitions.

1.2.2 Focus+Context

Focus+Context is a technique information visualization systems use to display areas of interest as focus regions, while still displaying the rest of the dataset in less detail, which presents context for the focus regions. Many of these approaches magnify the areas of interest and concurrently minify other regions, which leads to visual distortion of the original dataset topology. The usage of Focus+Context distortion techniques in visualization applications may cause some disorientation if visual aids such as landmarks or topological features are not present. Additionally, the user may be confused if the application does not provide smooth transitions from one view to the next.

TreeJuxtaposer uses Focus+Context effectively since the tree layout is visualized as a hierarchical structure, where the location of the root and direction of growth is known to the user, even after repeated navigations. The tree structure is able to convey relative node density and the path from any node to the root is well known by manually following the node ancestor path. Along with node labels and smooth animated transitions, navigations in TreeJuxtaposer prevent the user from losing orientation. However, AD itself does not provide topological reference for navigation and requires developers of applications to include navigational semantics based on their particular application-domain requirements.

As an aid to automated movements, smooth transitions help Focus+Context

applications both during navigation and in the resulting static views. When an area is stretched, a smooth transition provides a correspondence between the original state and the final transition state. Progressive guaranteed visibility also aids in making transitions easy to follow, since the interaction box or set of marked nodes are rendered first. Finally, the static view is easier to comprehend after following a smooth transition rather than after direct jump cuts [30].

1.2.3 Progressive rendering

Progressive rendering is a graphics technique for displaying a meaningful, partial scene in systems, allowing real-time user interaction, even for scenes that render slower than the time available to draw a transitional frame. When a user demands a response from the system, the appropriate action must be performed and scenes that render slowly must respond to the user actions with no noticeable delay. Simple scenes with a few thousand nodes in TreeJuxtaposer, for example, may render fast enough such that checking for user action during a rendering is not necessary. However, to scale to millions of nodes, progressive rendering approaches allow for smooth navigation and keep rendering from being a bottleneck that prevents immediate system feedback to user demands.

Although rendering fewer nodes and guaranteeing that the number of nodes rendered depends on the number of screen pixels allows scenes to be drawn faster, progressive rendering is still necessary for many large datasets due to increased time for handling larger datasets. However, progressive rendering overhead can potentially negatively impact the performance of applications that use progressive rendering. Therefore, systems that provide progressive rendering should allow either automated or manual control over the usage of progressive rendering of scenes that do not require several frames.

1.3 Thesis contributions

This section details the contributions of my thesis, presented in order of importance by chapter. Results for each performance claim are presented in Chapter 5. The conclusions for my thesis contributions are given in the final chapter, along with some recommendations for future work.

1.3.1 Accordion drawing contributions

- We developed a generalized infrastructure for accordion drawing applications that does not depend on extra spatial subdivision layers, but uses an inherent dataset topology, when available.

- We refined our generic generic rendering infrastructure by formalizing a three-step rendering process, which includes: partitioning in $O(b)$ time; seeding in $O(b + m)$ time; and rendering in $O(b + m)$ time, where b is the number of on-screen blocks and m is the number of marked groups.

- We created a numerically stable AD navigation algorithm that is capable of resizing many AD split lines concurrently. This means that we do not move a split line in our resizing algorithm more than once per scene, unlike AD for TJ1.

1.3.2 TJ2 contributions

- We developed new algorithms for topological tree rendering in our new AD infrastructure, in comparison to TJ1 quadtree-based AD methods. TJ2 rendering is pixel based and renders a scene five times faster than TJ1.

- We optimized storage and retrieval of ranges of marked data, which we use to perform marking operations on tree datasets in TJ2. Marking is eight times faster when marking a full $190,265$ node tree being compared to another

198,623 node tree, and rendering the fully marked tree is still faster in TJ2, without the caching techniques used in TJ1.

- We created a new TJ2 tree layout technique for our AD infrastructure. Compared to TJ1, TJ2 preprocessing is ten times faster, and overall memory usage is five times more efficient.

- We replaced spatial subdivision picking, used in TJ1, with topological picking that is nearly as time efficient with most tree topologies and is able to pick tree nodes in regions of datasets where TJ1 is known to fail.

1.3.3 TreeJuxtaposer Evaluation from InfoVis 2003 Contest entry

- We analysed a modified version of TJ1, TJ1-contest, with a standardized set of real tasks. This analysis helps understand the strengths and weaknesses of our AD approach to investigating tree-based dataset queries.

- We added an incremental search function to TJ1, which allows a user to quickly identify similarly named nodes. When a small number of search results are selected, our algorithm automatically marks their location in the tree topology with guaranteed visibility.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 1 presents motivation, contributions, and organization; Chapter 2 includes relevant previous work; Chapter 3 discusses our new tree navigation application, TJ2; Chapter 4 discusses our generic accordion drawing infrastructure, AD; Chapter 5 presents analysis of TJ2 and compares the performance of TJ2 with TreeJuxtaposer; and Chapter 6 concludes my thesis.

Chapter 2

Related Work

Navigating large datasets has long been recognized as an important problem in the information visualization community. AD navigation is in a class of information visualization techniques with an interesting, evolving history in human-computer interaction, computer graphics, and other fields of study. Tree visualization also has a past in information visualization, and there are several systems created specifically for biologists that continue to influence our development of TreeJuxtaposer features.

Previous work related to my main thesis contributions is presented as follows: Section 2.1 includes information visualization and other human-computer interaction and perception work; and Section 2.2 describes some common phylogenetic evaluation software as well as other tree-specific work in information visualization systems.

2.1 Visualization, interaction, and perception

Visualization of datasets with more data than available on-screen pixels is a problem since the size of datasets increases faster than the resolution of monitors. Furthermore, human perception limits the feasible amount of information that can be displayed even with infinitesimally small pixels. Obviously, there is very little advantage to monitors with pixels at the microscopic scale. An excellent resource for generic methods of graph visualization and navigation is a survey by Herman et

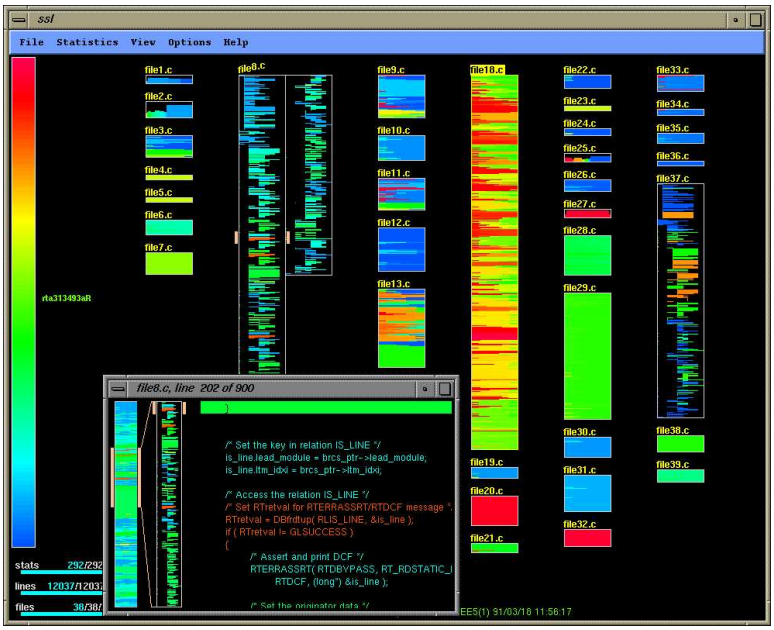

Figure 2.1: The SeeSoft [1] software analysis tool provides an overview of large software structure with meaningful color-encodings. The entire structure is visible and users may zoom into regions of interest to investigate details.

al [13].

Approaches such as the pixel-based software analysis tool SeeSoft [1], as shown in Figure 2.1, display an overview of an entire dataset with millions of items, and users may enlarge regions of interest to view details. The number of displayable items is impressive and gives insight into the global structure, which is especially useful when meaningful color codings are used. However, small details may be overlooked and culled out of view by the rendering process when datasets become larger than the number of pixels, or when the feature size is too small to perceive differences between adjacent pixels.

Applications that attempt to render extremely large datasets of thousands of items are usually either non-interactive, or require acceptable rendering techniques for displaying important details first, such as progressive rendering introduced by

Bergman [6]. For 3D scenes, this approach displays vertices, edges, and other higher level surface features in order. Since an overview of the dataset may be available with visualization of only the vertex positions, interactive techniques, such as camera positioning, can be used with a simple rendering, and details can be filled in when interaction stops. Other progressive rendering approaches may render landmarks or other important features first if such a scene decomposition is not available or does not provide visual benefits during interaction.

Magnification techniques, such as the fish-eye lens [7, 11] and related non-linear magnification fields [18], can be used to view local detail for data too densely packed to be clearly represented in full detail. A fish-eye lens uses the idea of a simple magnifying lens, but, unlike a conventional physical lens, uses distortion around the center of focus in place of an occluding boundary. This means the lens is used to distort some context around the magnified focus region and not hide useful contextual details. We use the term Focus+Context to refer the visualization technique where the focus is shown within surrounding context.

A related magnification technique is the document lens [31], shown in Figure 2.2, where a full text document is shown arranged in a grid, with each page in a different cell. An overview of the entire document is available but the text is too small to read all at once. Users can select a rectangular magnification region, approximately the size of a page of text, and the remaining document is drawn as it would appear on the sides of a truncated, skewed pyramid with the magnified region as the frustum. This approach requires less computation than the fish-eye lens, and is more suitable for viewing full pages of documents undistorted, practical for reading text. Text down the sides of the pyramid is also legible close to the magnified frustum region.

Hyperbolic geometry [19, 22, 23] visualizations remove the traditional Cartesian two dimensional context and use fish-eye visualization techniques for the entire scene. Users perform navigation through the hierarchy by changing the node in

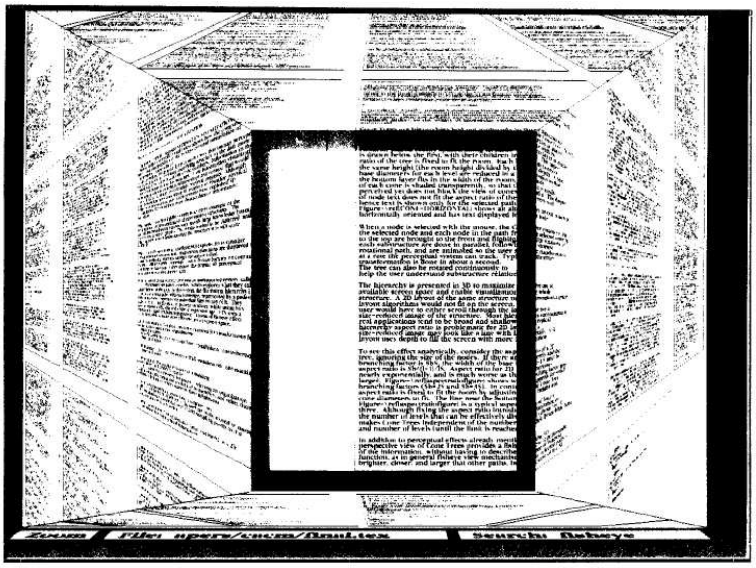

Figure 2.2: The document lens visualization tool [31] shows a page of undistorted text from a large document, and applies distortion to the remainder of the document. Distorted text near the undistorted region is legible.

focus; users may step through each level of the hierarchy or jump to any other visible node and the visualization responds with smooth animated transitions. This class of distortion-based visualizations is limited to tree hierarchy visualizations or other connected graphs, however, since users require structural, visual cues like the edges of a tree structure for navigation. Generic visualization of text objects, like the document lens, or side-by-side cells in a grid structure would not be visually pleasing with these techniques. In Figure 2.3, I show the H3Viewer visualization application rendering of a tree structured dataset.

Several systems use semantic zooming, which is based on generic level of detail (LOD) methods. Semantic zooming aggregates minor features into larger structures to reduce clutter from global overviews, and replaces larger features with their component minor features when focusing in on regions of interest. The Pad++ [4, 27] system, as shown in Figure 2.4, renders objects with infinite precision in an abstract,

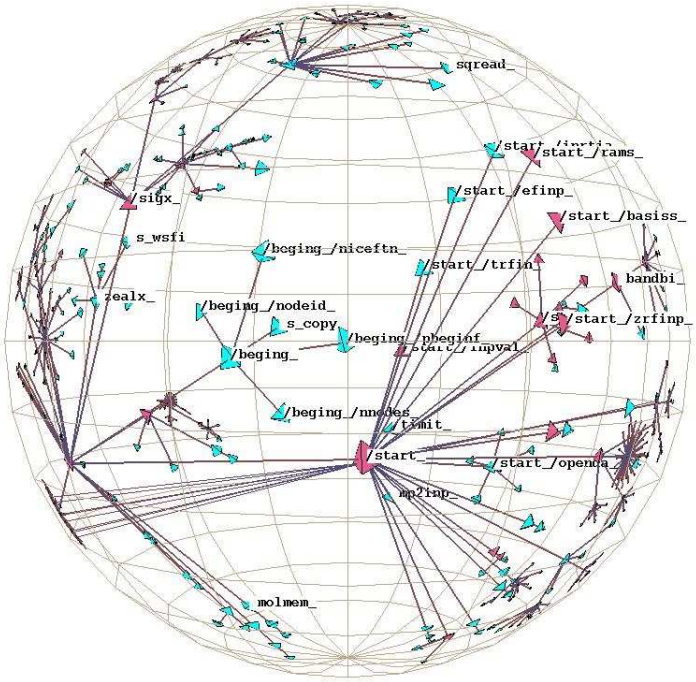

Figure 2.3: H3Viewer [23] is a hyperbolic geometry visualization application for navigating connected graph datasets. Hyperbolic distortions allow any data point, even far away from the focus, to be visible and have relative position with respect to other data.

semantic zooming world. Items are assigned a minimum and maximum visible size and smooth animation provides transitions between levels of detail. Semantic zooming has also been investigated in space-scale visualizations [12], where panning and zooming are used to give intuitive animated transitions. As the viewpoint zooms out, semantic zooming while panning allows a user to track global landmarks, so certain familiar features give much needed navigational context.

A closely related hierarchical zooming method is multiscale visualization [37]. This method presents aggregation, or selection, of underlying data instead of feature filtering approaches used in traditional semantic zooming applications. Multiscale visualization assigns implicit semantic representation to zoomed-out data, which either may be useful if the underlying data is similar, or may be detrimental if the

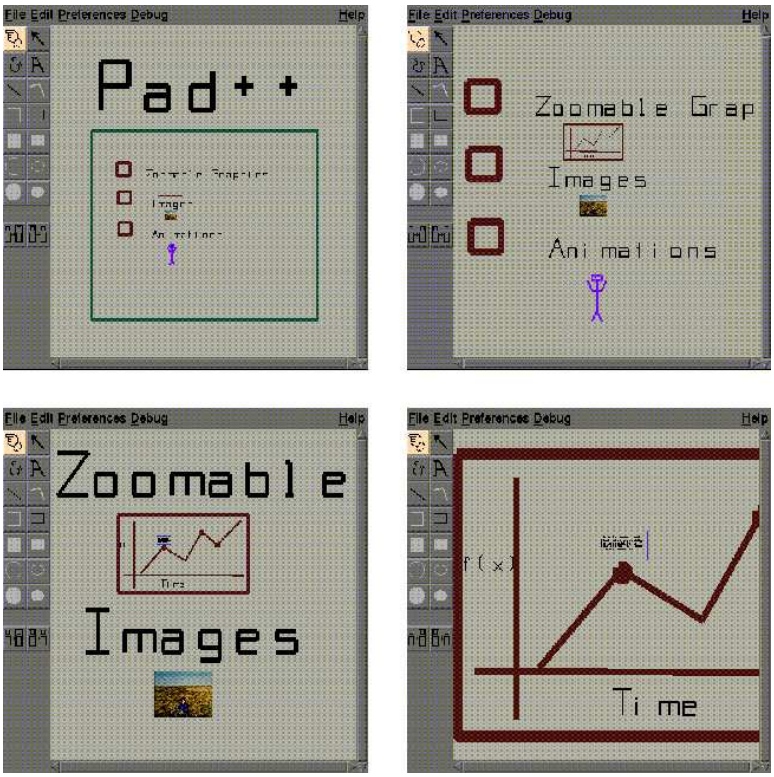

Figure 2.4: The infinite-precision world-space for objects in Pad++ [4] allows development of zooming features at several scales of magnification. This figure, from left to right, top to bottom, shows an example of repeated zooming on features. Semantic zooming also allows Pad++ to restrict visibility of rendered objects at user-specified magnification scales.

aggregation techniques hide data regions with high variability.

Degree of Interest (DOI) systems like continuous zoom [2] offer another type of semantic zooming. Groups of objects are assigned a percentage of the screen, so that when one object is focused on with a fish-eye magnification lens, other objects in the same group are shrunk at the same magnitude, to preserve the screen area devoted to the group instead of the entire context shrinking uniformly. The semantic zooming aspect arises from when objects reach an assigned threshold size, and with some cases, multiple foci are possible as groups of objects "open," and display more

detail.

Rubber-sheet navigational metaphors [34] introduce orthogonal, and polygonal convex hull, distortions where objects drawn on a two dimensional grid can be stretched as if they were drawn on a rubber-sheet. Areas of interest on the rubber-sheet can be stretched out, essentially magnifying them without the occlusions of traditional magnifying lens effects. Navigation with a rubber sheet is typically user directed with continuous zooming capabilities, as users pull defined region boundaries to increase the space allocated to an object. The borders of the rubber-sheet are tacked down, meaning that the context regions are squished to small regions but always visible, although compressed similar to [2]. With no semantic zoom, regions of the context need to be culled, or otherwise aggregated, as they are shrunk.

Landmarks, or regions of interest, in semantic zooming applications may not be visible while at extremely zoomed-out views. It may be desirable that certain characteristic objects never disappear from displays where the entire dataset is always visible. Implementations of critical zones [17] extend infinite precision visualization systems, such as Pad++, with methods of guaranteeing certain objects will always be visible at any level of magnification.

TreeJuxtaposer [24] (TJ1), shown in Figure 1.4, introduces accordion drawing, which combines rubber-sheet navigation with concepts of guaranteed visibility for select regions of data. TJ1 provides a scalable alternative to side-by-side analysis of trees, previously done by hand on paper printouts. The layout of TJ1 is quadtree based, and uses accordion drawing techniques derived from rubber sheet navigation. When the objects in context are shrunk or culled, highlighted landmarks are given rendering priority, by drawing above every other node in their context, with minimal feature size as space permits with other landmarks. Context nodes are given second-class treatment and not limited to how small they can be drawn; ranges of context nodes are considered landmarks in themselves, however, and cannot be squished completely out of sight. TJ1 scales to over $500,000$ nodes [24], and

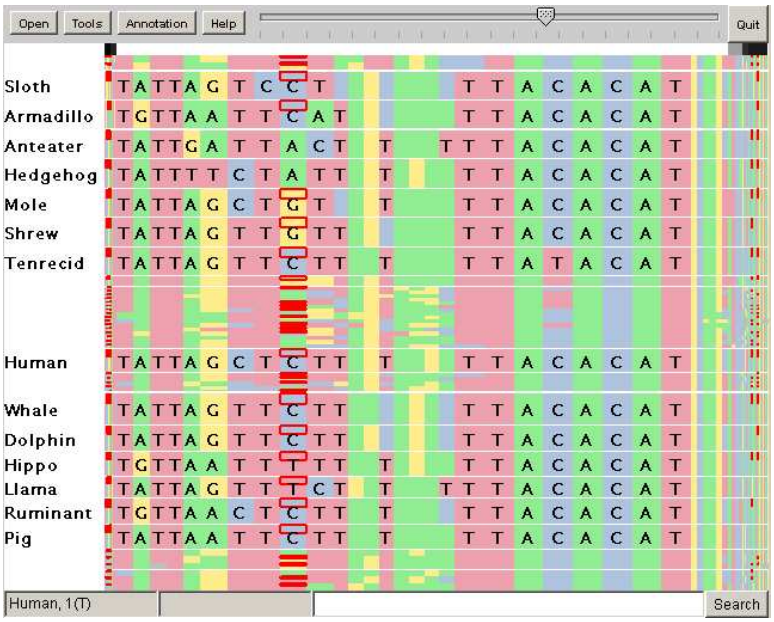

Figure 2.5: SequenceJuxtaposer [35] is an aligned sequence visualization tool that uses accordion drawing navigation. Each sequence is drawn horizontally and base pairs of each aligned sequence create visible vertical columns when there are no differences. Simple difference heuristics appear as red guaranteed visibility marks.

animated transitions are necessary to maintain context during continuous zooming. TJ1 uses a best corresponding node (BCN) criteria [39] to correlate matching nodes from pairs of trees under analysis, so selecting one node also selects relatively similar corresponding nodes in other trees under comparison.

More scalable tree analysis with TJC [5], is capable of rendering up to 15 million node trees in under one second. TJC removes the quadtree hierarchy, uses a simple grid-based structure, and optimizes data structures to dramatically increase memory performance. TJC also renders dense regions of trees without gaps and eliminates many of the rendering inefficiencies of TJ1.

Beyond TreeJuxtaposer, there have been several accordion drawing applications: SequenceJuxtaposer [35], shown in Figure 2.5, and PowerSetViewer [25], shown in Figure 2.6, which render rectangular regions of color to represent their

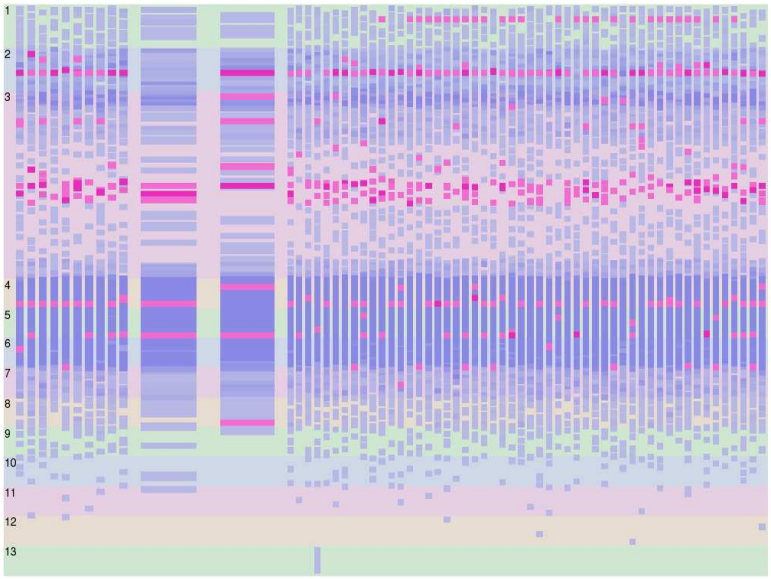

Figure 2.6: The PowerSetViewer application [25] is a visualization system for powersets. Powersets are drawn as a single enumerated sequence of nodes. PowerSetViewer line-wraps the world-space at the end of each column and visually separates cardinalities by alternating the background color.

data. Because they layout datasets on simple grid structures rather than trees, these applications impose a hierarchy on their datasets. SequenceJuxtaposer aligns its input data vertically, since it assumes sequences that are drawn horizontally to be somewhat aligned vertically, in a large stretchable grid.

PowerSetViewer is a grid-structured accordion drawing application that displays powersets, or the set of all possible sets of nodes, as a single enumeration [25]. An interesting aspect of PowerSetViewer is its ability to add or delete data from the grid over time, and modify the grid accordingly. Furthermore, PowerSetViewer does not require allocation of a grid large enough for all addressable space in the powerset. Instead, it builds a sufficiently large grid to draw a sparsely distributed set of sets, on the order of up to two million, into the powerset domain.

2.2 Phylogenetic tools and tree visualization

Although not constrained to tree topology datasets of a biological nature, TreeJuxtaposer is designed with many desirable features for phylogenetic research, which are briefly described in Section 1.1. However, since several tree analysis systems are used to investigate phylogenetics, the evolutionary history and relationships between organisms, it is important to describe a few of the most influential systems here.

Currently, in the field of evolutionary biology, efforts are underway to categorize every organism with a single tree called the Tree of Life [9], which shows hypothesized relationships between existing organisms and their proposed, or otherwise extinct, ancestors. The research effort is broken into small pieces, such as fungi [14], and further by research lab, such as the Hibbett lab that studies *homobasidiomycetes* [15], the mushroom-forming fungi. Once data is collected from each group, small trees are combined into supertrees [33], which would culminate with a hypothetical set of trees of all known organisms.

Since methods of determining the organism relationships are subject to error for several biological reasons, evolutionary biologists use several statistical models to reconstruct evolutionary trees, the most common being parsimony or Bayesian inferences in relationships. Parsimony-based tree reconstruction [38] relies on minimal characteristic changes between species identifying close ancestors, while Bayesian techniques [16] use Markov Chain Monte Carlo simulation techniques to estimate tree topologies. Both of these methods are statistical inferences and are subject to error, which therefore require humans to analyze and add their professional knowledge to the results.

Manual investigation of data is time-consuming and understandably complex, so several software packages are available to visually investigate the results from evolutionary tree construction software. MacClade [20], and more recently Mesquite [21], are two well-known and useful software packages built by evolution-

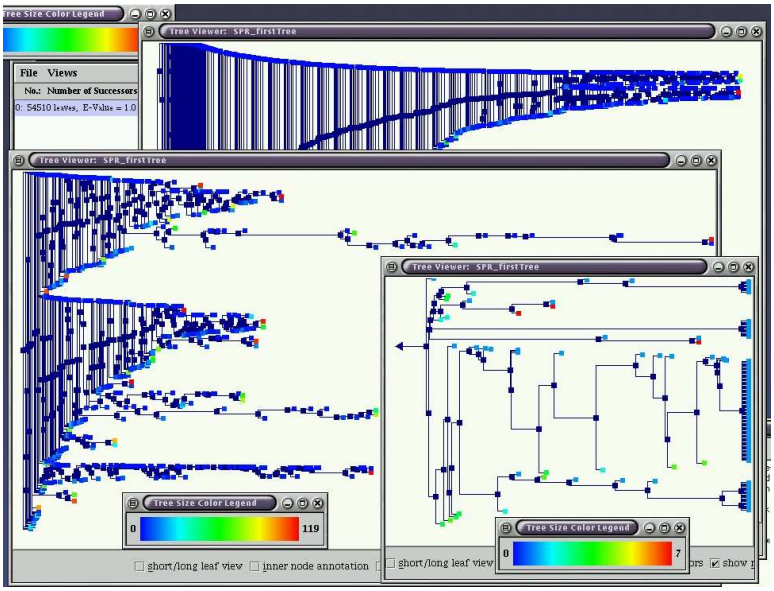

Figure 2.7: TreeWiz [32] is a scalable phylogenetic tree visualization system capable of supporting 50,000 nodes. Each viewpoint change for navigation opens a new display window.

ary biologists. They offer a set of useful editing and analysis functionality, but lack in scalability. Some of the more interesting features of MacClade are the ability to annotate and edit the properties of tree data. A simple panning canvas is used to display a visual representation of a tree of several nodes at a time, which is sufficient for many tasks.

A more scalable system called TreeWiz [32] supports up to 50,000 leaf nodes in a Java application. Subtrees that do not fit onto the visualization are collapsed into their parent nodes and assigned a color from a density map. However, navigation is limited since each change of viewpoint creates a new display window. Aggregation of subtrees into parent nodes is also a feature of the SpaceTree [29] browser, which supports automated subtree collapsing and several other pan and zoom type modes for navigation in collapsed trees.

Cheops [3] is another scalable system capable of browsing tree structures of

up to one billion nodes, and while suitable for a concise index, it is not well-suited for displaying details of the topological structure. For deep subtrees, Cheops occludes information from sibling subtrees to show one region of focus, typically as a path from the root to a target node.

Alternative marking techniques have also been introduced. Carrizo [8] introduces a color-filling approach to tree annotations. Instead of coloring tree edges, Carrizo colors the regions under subtrees to provide much larger colored regions to indicate the properties of a subtree.

Chapter 3

TJ2

We made several significant changes to TreeJuxtaposer to make TJ2 work with our fast, general AD infrastructure. The most significant changes were in the TJ2 rendering process, where we developed new algorithms for laying out nodes, placing tree edges, and performing gapless rendering with smaller rendering queues. Our rendering algorithms are now pixel-based, with a rendering time complexity of $O(p)$ where p is the number of vertical pixels, rather than $O(n\ log(n))$ where n is the number of nodes in the topology. This means our rendering is more scalable, since dataset topology does not affect our rendering performance. We characterize three cases of potential rendering gaps in ascent-based rendering, and our algorithm minimizes the amount of drawing required to fix those gaps.

The marked ranges improvements for TJ2 allow for much faster color lookup for marked nodes, as well as deciding when nodes are not marked, by using a tree-based range lookup instead of linear searches through all marked ranges for every node being drawn. Collapsing the ranges efficiently was also an improvement for storing and retrieving large numbers of node differences when comparing trees. Although nodes are stored more than once, looking up node colors quickly is not possible unless each marked node is stored; color lookup time is $O(m\ log(r))$, where m is the number of marked groups and r is the total number of nodes ranges stored by any particular group. Our localized algorithm for finding all indirectly marked

nodes is sufficiently fast and we no longer require node color caching, which allows us to load larger tree datasets. The efficiency of marking depends on the dataset, but we achieve an average marking speed $O(k)$, where k is the total number of nodes in the range marked by the user. Marking the entire InfoVis 2003 Contest [28] dataset $\mathsf{animalia}_\mathsf{A}$ tree of $190,265$ nodes while comparing with $\mathsf{animalia}_\mathsf{B}$ takes less than two seconds to process, as discussed in Section 5.4.

TJ2 also introduces topological picking to TreeJuxtaposer, which allows a user to pick nodes in sparse topological regions of a tree. Although the picking algorithm is $O(h)$, where h is the topological height of the tree, we find it is sufficiently fast for the deepest trees TJ2 can currently load; picking is interactive with trees taller than 1000 nodes.

In this chapter, I present the major improvements of TJ2 over previous versions. In Section 3.1, I describe our node layout algorithm. I discuss our tree rendering algorithms in Section 3.2, which follow the tree topology. In Section 3.3, I discuss our marked range improvements. Finally, in Section 3.4, I describe our topological picking methods.

3.1 Node layout

TJ2 incorporates significant changes to the tree layout algorithms from TJ1-based TreeJuxtaposer applications. Trees in TJ2 are still drawn right-aligned, meaning that leaf nodes are found on the right-hand-side of the tree with the root on the left-hand-side. Due to this orientation, in this section, I will introduce our conventions to describe TJ2 layout algorithms and rendering techniques. In TreeJuxtaposer, the width of the tree is the total number of leaves and the height of the tree is the longest branch length. Horizontal node edges are the height component for each non-root node and vertical edges are the width component for each internal node with two or more child nodes. Refer to Figure 3.1 for a pictorial description of these terms.

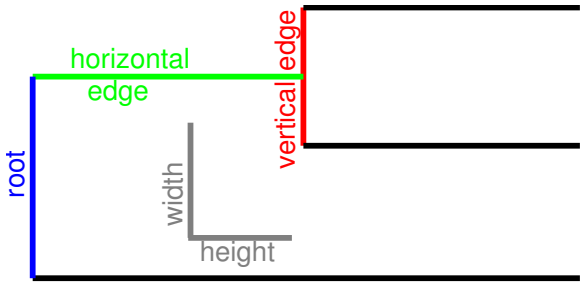

Figure 3.1: The naming conventions used in this thesis. The root node, in blue, is drawn with no horizontal edge. The internal node is marked in green and red for horizontal and vertical edges and the leaf nodes have no vertical edges. The width of the tree is the number of leaf nodes and the height is the longest path from the root to a leaf.

TJ1 algorithms for rendering and node layout create a hierarchical spatial quadtree layout, described in Section 4.1, which is inefficient for trees since most trees have many more leaves than height. The quadtree is built on a **base grid** of uniformly sized **base grid cells**, as shown in Figure 4.1. A base grid cell contains a reference to a node of the topological tree, and a quadtree cell points to up to four children cells, which could be either base grid cells or interior quadtree cells.

TJ1 quadtree subdivisions are built on the base grid to facilitate traversal, so partitions divide the number of grid cells in half in both directions for each layer of the quadtree. This partitioning is inefficient for most cases since the base grid is often not close to square since the width of the topological tree tends to be much greater than the height. The interior quadtree cells are most efficient in the few cases where the topological tree height is almost equal to the tree width, which happens to be the case in pectinate trees, also known as "comb-shaped" trees, that occur in some biological contexts.

However, as introduced in TJC [5], a more efficient technique to store tree nodes in a grid is possible with separate horizontal and vertical binary trees. TJ2 uses the basic idea of separate structures in TJC, but is quite different in all tree

doGridding Function
input: set of nodes N from tree T, in post-order list
grid G large enough to layout T
output: nodes N assigned to rectangle of coordinates in G

$y \leftarrow 0$
while $N \neq \emptyset$
  $n \leftarrow N.pop$
  if isLeaf(n)
    $n.maxX \leftarrow G.maxX$
    $n.minY \leftarrow y$
    y++
    $n.maxY \leftarrow y$
  else
    $n.maxX \leftarrow$ getMinX($n.Children$)
    stretchMinX($n.maxX$, $n.Children$)
    $n.minY \leftarrow$ getMinY($n.Children$)
    $n.maxY \leftarrow$ getMaxY($n.Children$)
  end if
  $n.minX \leftarrow n.maxX - 1$
end while

Figure 3.2: *doGridding* function assigns a grid position in G to each node in T. Leaves are positioned on the right side of G, internal nodes span their children and are as wide as the sum of their child widths, and all nodes initially are in cells one base grid cell high. Cells are stretched for each child of *parent* that does not have a $minX$ value equal to $parent.maxX$.

layout, rendering, and culling algorithms. In the remainder of this section, I describe how nodes are mapped to grid coordinates in Section 3.1.1. Then, in Section 3.1.2, I discuss a necessary modification for placing horizontal node edges during rendering in TJ2 that is not required by TJ1 node mapping.

3.1.1 Mapping nodes to grid

Node layout in TJ2 is quite different from layout in TJ1, but both TreeJuxtaposer applications create very similar-looking tree visualizations with the same base grid

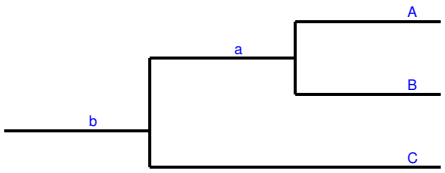

Figure 3.3: A small subtree for our gridding example. The nodes in are added in post-order.

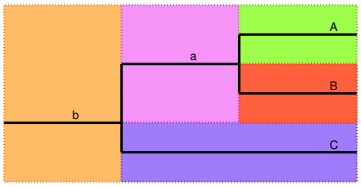

Figure 3.4: The nodes of Figure 3.3 added to the grid. Our tree layout partitions screen-space into a fully covered grid cell layout, as shown.

size. Instead of using a spatial subdivision method, TJ2 partitions the base grid into rectangular regions for each tree node; we call the partitioning process **gridding**.

Topological tree nodes are assigned to cells using an algorithm based on the pseudocode for *doGridding* in Figure 3.2. The cells form the boundary around tree edges for a tree node and distort with respect to the Accordion Drawing methods on the base grid. Each internal node in the topological tree is drawn with two tree edges, one horizontal connection to its parent and one vertical connecting its children. Leaf nodes and the root node are special cases: leaves have only a horizontal edge and the root has only a vertical edge. Cells for each node in both TJ1 and TJ2 are bounded by four accordion split lines, which are movable grid lines described in Chapter 4, each with minimum and maximum lines in the left-to-right X and top-to-bottom Y directions. In TJ2, the leaf-to-root node placement and initialization algorithm is linear in the number of nodes in the dataset.

We must have enough base grid cells in height to support the deepest nodes of

the topological tree, which is the equal to the height of the tree. Since all leaves are of the same vertical weight, we must have enough base grid cells to place every leaf in an individual cell, which is exactly the number of leaves. Therefore, the dimensions of the grid are known after parsing the input dataset and TJ2 can initialize the Accordion Drawer split line structures, creating the base grid.

As an example of the gridding process in Figure 3.2, consider the small tree in Figure 3.3, where nodes are placed in this post-order traversal: A, B, a, C, b. The leaves A and B are: placed in the grid one cell tall and wide; adjacent to each other; and aligned with the rightmost split line in G, as shown in Figure 3.4. Leaf C is placed next to B, but is two cells tall since it must match the height of internal a, which was placed on the other two leaves. Both C and a are attached to internal b, and the internal nodes a and b are as wide as the sum of their child node widths. The time complexity of the insertion per node is on the order of the number of children, since the $stretchMinX$ function processes all children; leaves have no children but require constant time to initialize. Therefore, the complexity of the entire insertion process is $O(n)$, for a tree topology of n nodes.

3.1.2 Placing horizontal node edges

This section deals with positioning the horizontal node edges in TJ2, necessary with the partitioning process from gridding in Section 3.1.1. All edge positions are calculated relative to the width of their subtrees; leaf edges are placed in the center of their cell and internal node positions depend on the positions of their children. In an orthogonal tree layout, the density of horizontal tree edges show the width of subtrees and the height of child nodes, and the positions of some of those edges determine the length of parent node vertical edges.

The placement of horizontal node edges is slightly more complicated in TJ2 than in TJ1, since TJ2 partitions the entire base grid for node domains. TJ1 node to cell mapping places nodes in the base grid, but the nodes are given offsets to

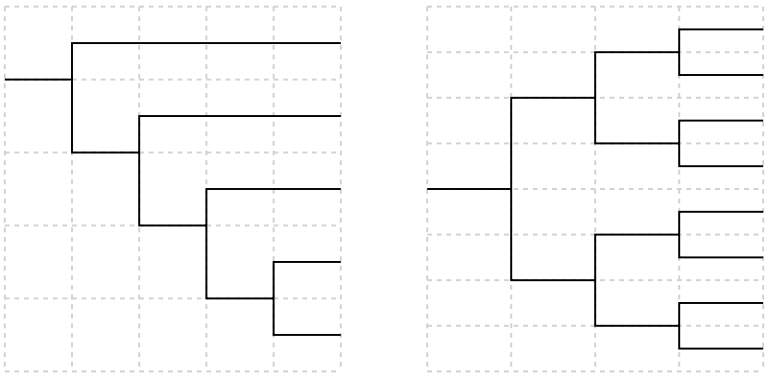

Figure 3.5: The balanced tree on the right places the horizontal edge of its root in the center of its cell width, but the pectinate tree on the left places the same edge much more toward the top of the root cell. The horizontal edge position for any tree node may move anywhere within the cell and, unlike TJ1, cannot be a constant offset since node cells span the entire width of its descendant node cells.

a single base grid cell at the minimum height of all child nodes and somewhere close to the center child position. This mapping differs from TJ2, which maps a tree node to a cell that is as wide as the sum of its children widths. When a node is rendered in TJ1, the horizontal edge position is simply calculated with the offset and grid cell position. We want the same performance and correctness for horizontal edge position computations in TJ2 as in TJ1: computable in constant time and guaranteed to attach to the vertical tree edge that stretches from the first to last child horizontal edge positions.

The horizontal edge position for a subtree may be anywhere in its bounding cell. To understand how a horizontal edge can change according to the underlying subtree structure, consider Figure 3.5. With the class of subtrees called pectinate trees, similar to the tree shown on the left of the figure, we can generate examples of horizontal edges placed anywhere vertically within their cell.

We cannot compute the horizontal edge position in TJ2 with a set offset; if we attempt to use TJ1 methods in TJ2, we quickly see why we need to calculate the

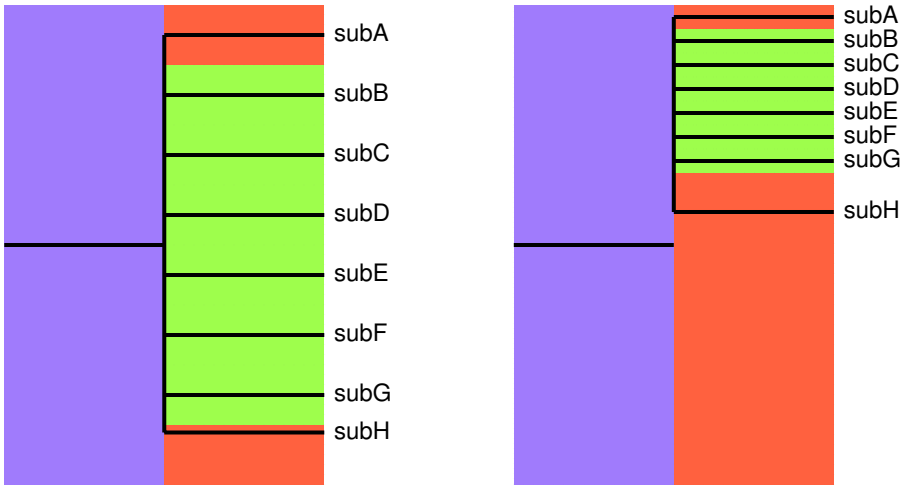

Figure 3.6: The relative position of the horizontal node edge for an internal tree node depends on the position of its inner children cells. This figure shows why resizing where the edge in the blue cell cannot use an offset similar to TJ1 horizontal edge position computations. On the left is a small subtree with the parent horizontal edge in the blue cell calculated from an offset; all other nodes *subA* through *subH* are subtrees with the horizonal edge of *subH* positioned close to its boundary with *subG*. If *subH* is vertically grown without moving the outer boundary for the blue cell, as shown in the right figure, the relative positions for all *subA* through *subH* remain the same and move since their cells change in size. However, the node in the blue cell remains at its initial position since its size has not changed. The horizontal edge for *subH* is drawn on the wrong position relative to its parent and our small example shows a broken subtree.

horizontal edge positions differently. Take for example the small tree in Figure 3.6, where *subA* to *subH* are subtrees of a common parent in the blue cell. If we set the internal node in the blue cell where it is, as a vertical offset in the blue region, the node does not move vertically when the top and bottom cell boundaries do not move. However, if we resize the cell with subtree *subH* towards *subA* to be nearly the same width as the blue cell, it is possible for the horizontal edge of *subH* to be on the wrong side of the blue cell horizontal edge. This is possible since if the horizontal edge for *subH* is close enough to the *subG* cell boundary, then as *subH*

gets wider, it will eventually pass its parent horizontal edge.

The horizontal edge positions in TJ2 are computed by determining the center of the vertical edge that we know to be drawn. In Figure 3.6 on the left, we see that the vertical edge for the internal node in the blue grid cell is drawn from $subA$ to $subH$. But, if we use both $subA$ and $subH$ edge positions to calculate the horizontal edge position of their parent, we see that this becomes recursive when we need the horizontal edge positions of the children of $subA$ and $subH$, with an exponential cost of $O(2^h)$ where h is the height of the edge we wish to compute.

Since the vertical edge is only definitely drawn across the cells from $subB$ to $subG$, the green cells in Figure 3.6, we notice that it is possible to place the horizontal edge of the blue cell using only the width of the green cells. Therefore, ignoring the positions of horizontal edges $subA$ and $subH$, we are left with the remainder of the cells to calculate the horizontal edge position. There are many possible ways to compute a horizontal edge position, but we choose a simple mid-point of the central children cell boundaries, the child cells that are neither first nor last, for example the nodes highlighted in green in the figure.

The length of the vertical edge must use the positions of the horizontal edges for the first and last child, to connect all children to the horizontal edge of the parent. However, this is also a constant time calculation since no recursion is required, each horizontal edge calculation is constant, and only two such calculations are required. To reduce calculations of horizontal edges to only require what is visible, we also cache the results of previous horizontal edge positions, by storing the frame number for each calculation, while no movements have occurred.

In the degenerate case of a node with only one child, the horizontal edge of the parent node is aligned with the child node horizontal edge; no vertical edge is drawn but recursion is necessary to find either the first descendant with more than one child node, or a leaf. Nodes with two children are not a degenerate case, but are simply cases of Figure 3.6 that do not have green internal cells; the grid

line between the red outer cells would be the only location suitable for the blue cell horizontal edge, in this case. The worst case of a horizontal edge placement for a totally degenerate tree, with N nodes and a height of N, where every node has a single child and the tree is a single line, is $O(N)$. However, in practice, the degenerate case is rare; that is, few nodes have single children. The complexity for this typical, non-degenerate case is $O(1)$ per edge, the same as TJ1.

3.2 Rendering trees

Rendering a minimal number of tree edges for any tree topology depends on the minimum feature size of a tree node: the edge width. Since "pixels" are really artifacts of the hardware restrictions of physical monitor pixels, we choose to use the terminology **"block"** to refer to the smallest visible features of our drawing objects. Blocks are always integer multiples of pixels, and are by definition pixel aligned; blocks are simply a coarser screen representation than pixels. This terminology becomes useful when using thicker tree edges than minimal one pixel-wide lines; high-resolution monitors capable of 200 DPI, such as the IBM T221, make single pixel-wide lines hard to see.

As described in Section 4.2, the generalized rendering infrastructure of AD follows the generic three-step pattern of: partition an application-specific base grid into pieces smaller than the minimal feature size, following the hierarchical AD structure; seed the application-specific dataset nodes that correspond to the partitions; and draw the seeded nodes and other necessary "attached" nodes, again in an application-specific manner. Of the steps listed for this pattern, partitioning is described in detail in Section 4.2.1. Seeding and drawing are optimized according to the dataset topology and are discussed in this chapter: I describe how TJ2 performs leaf-range seeding in Section 3.2.1; and how TJ2 draws nodes, beginning with the seeded leaf-ranges and ending at the root node, in Section 3.2.2.

3.2.1 Node seeding

Before rendering starts, we prioritize the order of node and subtree rendering in a rendering queue, with a **seeding** algorithm. The order of rendering is important for large datasets that cannot be completely drawn during animated transitions and rely on progressive rendering techniques to prevent disorientation. Progressive rendering draws pieces of the tree in several frames instead of the whole scene at once, if rendering the scene takes longer than 1/20 of a second. While rendering a small fraction of the tree does not give a user the entire picture, we try to render the most important parts of the scene during the first frame. The important parts of the tree visualization scenes are the marked nodes, mentioned in more detail in Section 3.3, the interaction box being dragged by the user, and, to a lesser extent, the upper sections of the tree.

The seeding process starts by adding the roots of marked subtrees, or otherwise individual marked nodes, to the rendering queue. We render subtrees of marked nodes by drawing the subtree root first, then rendering both up to the topological root and down to some leaf in the subtree. This bidirectional rendering of marked nodes allows the rendering process to draw the most important marked node subtree roots first, as visual landmarks, along with the context of root and leaf node paths. We do not require the leaf node paths to be marked similarly, but it is typical for an entire subtree to be marked in one color, especially if a user manually marks subtrees. The cost of rendering this path from root to leaf is $O(h)$, where h is the height of the tree, but we also cache whether nodes have been rendered for a scene, which somewhat reduces the drawing effort. Marked regions are stored as ranges, which may represent a forest of subtrees, so the seeding process breaks each marked range into subtree components and adds the root of each subtree to the queue. I describe marked regions in more detail in Section 3.3.

After seeding the marked node subtree roots, we add the remainder of the topology with leaf ranges. The entire tree is subdivided with a binary process until

either one leaf remains in a range, or the leaves in a range are smaller than some standard size called a **segment**. Section 3.2.2 discusses segments in more detail, but for the purposes of our TJ2 seeding discussion, segments are typically smaller than a visible, on-screen pixel.

Since we may have several leaves in a segment, the seeding subdivision process is responsible for partitioning the entire set of leaves, knowing the dimensions of the rendering canvas, so the drawing process does not need to do any partitioning. The drawing process is given each piece of the tree and renders only one leaf-to-root path per segment, which is discussed in Section 3.2.2. When adding leaf ranges to the rendering queue, the seeding process places any ranges that are inside the current interaction box at the front of the queue so the drawing process can prioritize these regions.

Unlike previous versions of TreeJuxtaposer that seed rendering with the top cell of a quadtree hierarchy, TJ2 begins to draw the scene with a drawing queue of a certain size, and this size only decreases as the scene fills with rendered nodes. In TJ1, the drawing queue starts with the largest, top quadtree cell and grows the drawing queue by repeatedly adding necessary, monotonically deeper cells of the quadtree hierarchy, which puts stress on the data structures used to store, remove, and order that queue information. TJ2 uses a simple list as its queue, so removal operations are constant, where TJ1 operations are all logarithmic since it uses a binary tree dataset for its drawing queue. The TJ2 rendering results with our new seeding, discussed with details in Chapter 5, show that we can reduce the number of nodes rendered with software and our methods require only a small increase in time to draw per node.

3.2.2 Drawing trees

Tree rendering in TJ2 is based on the tree topology and spatial position of nodes from gridding. This section focuses on turning the input, a list of leaf ranges from

the seeding process, into a fully rendered tree visualization by drawing a minimal set of tree edges. Each leaf range contains either a single leaf, or several leaves in a small vertical range; we guarantee that only one leaf in each range, plus the path from that leaf to the root, will be drawn by the rendering process. Furthermore, the leaf ranges partition the set of all leaves, so there are no gaps in the set of all initially seeded leaf ranges.

In our rendering process, we do not force alignment of leaf ranges to discrete regions of the screen, such as pixel alignment, and we do not force leaf ranges to follow topological features of an input dataset. Either restriction would complicate our seeding subdivision process, which needs to be fast to avoid extra computational overhead from our software solution; our leaf range subdivisions are done with the fast, generic accordion drawing code, discussed in Section 4.2.1.

During the drawing of leaf-to-root paths, we make sure the time spent drawing the frame does not violate our per-frame progressive rendering restrictions, if progressive rendering is enabled. Every 1/20 of a second, the rendering algorithm flushes the current drawing results to display the current, partial tree output and the system checks for grid movements from user interactions. The drawing queue clears and restarts the rendering process either if any user action is detected, or if the current drawing is still undergoing an animated transition. It is worth mentioning here that new user actions force the previous user action to jump cut to its final position before processing new movements. TJ1 animation is not robust in this way, which causes several grid positioning problems from propagation of numerical errors, as I mention in Section 4.3.2.

In order to discuss the issues, the rendering is presented in several sections: choosing a segmentation width in Section 3.2.2.1; ascent rendering in Section 3.2.2.2; and choosing the termination for ascent in Section 3.2.2.3.

3.2.2.1 Choosing a segmentation width

The stopping criteria for the subdivision component of the seeding process is an issue mentioned in Section 3.2.1. Since we want to eliminate drawing gaps in dense regions but not draw too much, TJ2 seeds ranges of leaves that are smaller than a vertical block, if leaf density is greater than one leaf per block, to ensure that at most one leaf is drawn for each range. However, choosing a **segment width**, our partitioning stopping criteria for leaf ranges, of **less than one block**, meaning that ranges larger than one block are subdivided, is not sufficient.

Because we do not know the alignment of blocks to leaf ranges in our final set of seeded ranges, and do not know which leaf in the range will be rendered, we cannot be sure that rendering leaves for adjacent leaf ranges will cover all blocks. Section 3.2.2.2 describes why choosing a leaf to render based on block alignment is not sufficient for solving this problem. Referring to Figure 3.7, knowing that leaf ranges contain many candidate leaves to render, a leaf range L_k may render its single block-wide leaf in block row R_{m-1} while adjacent leaf range L_{k+1} renders its single block-wide leaf in block row R_{m+1}, leaving a gap in block row R_m.

The solution to this poor choice of segment width is to restrict the width of a segment to **less than one-half block**. A tighter restriction with smaller leaf ranges adds more leaf paths to render, but does not add computational complexity with approximately twice the rendering. The benefits of sub-half-block segments include a simple fix to the alignment problems seen with larger segments, and we still do not require direct computations of block alignment and leaf range position. We choose the half-block segment width from observing, in Figure 3.8, that of the partitioned adjacent leaf ranges smaller than a half-block, there is at least one full leaf range in every block. However, the half-block segment width only eliminates drawing gaps at the leaf level, so we must refine the traversal process to eliminate other drawing problems.

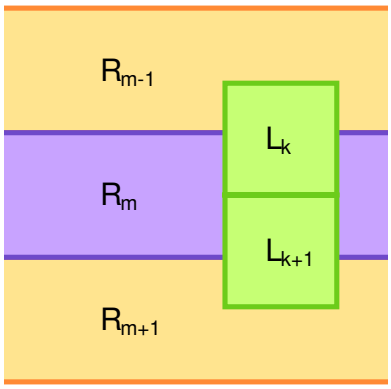

Figure 3.7: Restricting the leaf range width to less than the block width is not sufficient to render in every block at the leaf level. L_k and L_{k+1} are adjacent leaf ranges, both of which may contain several leaves to render, but we only want to render a single leaf in each range. The local blocks rows are R_{m-1}, R_m, and R_{m+1}, we assume a dense leaf layout, and are attempting to draw at least one leaf into each block. Since L_k overlaps with R_{m-1} and L_{k+1} overlaps with R_{m+1}, it is possible that a leaf will not render into R_m from either leaf range. We cannot shift the leaf ranges up or down to align with the blocks since we use a partitioning process from generic accordion drawing functionality. Therefore, the maximum width for leaf ranges is too large for the leaf partitioning process.

3.2.2.2 Ascent rendering

A second rendering problem occurs with our bottom-up rendering technique, as shown in Figure 3.9. When **ascent rendering**, rendering a path from the leaf nodes to the root node, we notice that there may be **horizontal gaps** from naïve path choices. For example, a sub-block subtree attached to a node close to the root of the hierarchy, where drawing is sparse, may not be drawn if its leaf is not chosen. This was not a problem with descent, or root-to-leaf, based methods in TJ1 since all such sub-block subtrees attached in a sparse region of the topology would be drawn. However, TJ1 rendering performance indicates that its methods over-render nodes deep in the hierarchy, exactly what TJ2 attempts to eliminate by ascent rendering.

For dense topological regions, paths from leaf nodes to internal nodes can be

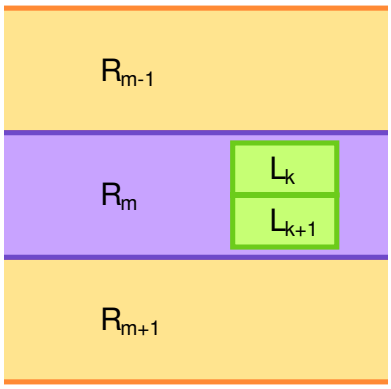

Figure 3.8: Restricting the leaf range width to less than half the block width is sufficient to render in every block at the leaf level. L_k and L_{k+1} are adjacent leaf ranges, both of which may contain several leaves to render, but we only want to render a single leaf in each range. The figure shows both leaf ranges clearly inside block row R_m, but we notice that shifting the leaf ranges up or down so either L_k or L_{k+1} are partially in R_{m-1} or R_{m+1} are exclusive events; one of L_k or L_{k+1} would still be in R_m. We cannot shift the leaf blocks in any way to exclude at least one full leaf range inside any block row. The maximum width for leaf ranges to guarantee rendering leaves in every block, therefore, is slightly less than half the width of a block.

culled into single horizontal lines instead of drawing the complete subtrees under all internal nodes, until paths connect to subtrees larger than the block size. When we assume the rendering paths of a leaf range are single horizontal lines from culled subtrees, our horizontal line rendering gaps occur when we do not draw the spatially highest culled subtree in a leaf range. Since every path of a leaf range under our assumption renders into the same block row, we only need to render the path in a subtree that is not covered by any other subtree. Therefore, our leaf selection in ascent rendering depends on finding the highest subtree possible from any leaf in the range, with a restriction that the subtree width is less than the width of a block.

Finding the highest subtree in a leaf range is not an expensive process. We do not need to examine each leaf in the range; the number of leaves to examine per range is constant, and depends on our ascent checking width. The **ascent width**

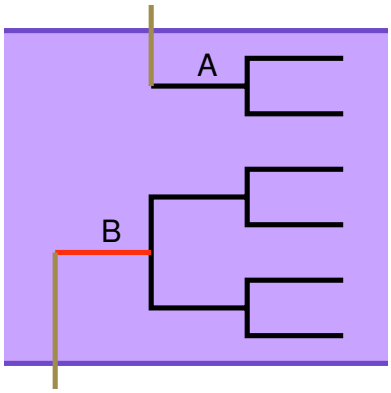

Figure 3.9: Our rendering choices for dense leaf ranges in ascent rendering affects the rendering output for horizontal edges in sparse regions. Given the two subtrees A and B from the figure, both of which are contained in the leaf range highlighted in blue, we need to choose one horizontal line path from some leaf to the root to represent both subtrees. Furthermore, the parents of A and B are large enough to terminate ascent searching since they cannot be represented with the same horizontal line path. If we choose either leaf in A, we render two nodes high, while rendering any of the four leaves of B, we render three nodes high. However, rendering A would prevent us from rendering B, so the line segment marked in red would not be drawn if we make the poor choice of rendering A. Our ascent rendering process must ascend all possible subtrees representable with horizontal line paths to render the spatially tallest subtree, in this case B.

is further discussed in Section 3.2.2.3, which includes choosing an appropriate value for segment and ascent widths.

Without loss of generality, assume that the leaves in range L are enumerated from lowest to highest in some vertical direction, from L_s to L_k, as in Figure 3.10. We begin by following the path from L_s to node A, which is the first node that is wider than the ascent width; B is the child of A along the path to L_s. We store B as the highest subtree, H, for the leaf range, so far, and continue searching L for higher subtrees.

Each internal node stores the widest leaves under its subtree, so we can find L_i, the maximum leaf under A, in constant time. Furthermore, we can find the

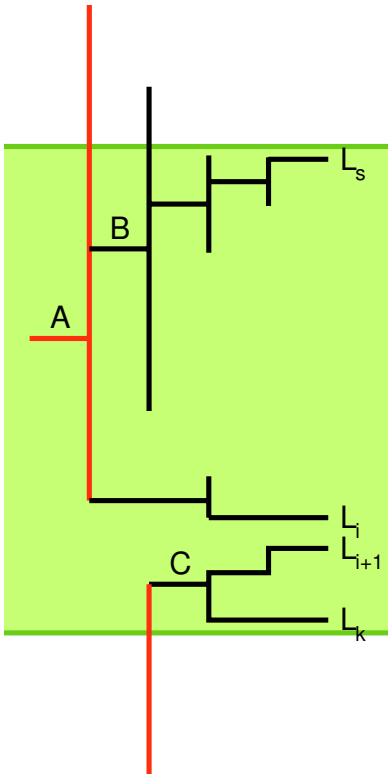

Figure 3.10: Finding the highest subtree in a leaf range, with leaves L_s to L_k, which are not as wide as the segment width shown as the green background. Starting from L_s, we ascend the topological tree until we get to the first subtree wider than the ascent width. In the figure, A is the first subtree wider than the ascent width and B is the child of A along the path to L_s; I do not draw the entire tree in the figure, only the traversed paths. We find L_i, the maximum leaf under A, with a constant time operation, and continue the process with its neighboring leaf, L_{i+1}, which is under C. L_{i+1} is still in the leaf range, so we ascend again, this time finding C as the first node that is not as wide as the ascent width. C is spatially lower in the tree than B, so B is still the node we render for the leaf range. The parent of C has its maximal leaf outside of the leaf range, so the process is finished; we render from L_s to the root of the topological tree.

AscentRender Function
input: L = sub-segment leaf range [with leaves $L_s, L_{s+1}, \ldots L_k$]
output: path P rendered from L to tree root

```
H ← L_s
n ← L_s
while n ∈ L
    p ← getParent(n)
    while subtreeWidth(p) < ascentWidth
        n ← p
        p ← getParent(n)
    end while
    if nodeHeight(n) > nodeHeight(H)
        H ← n
    end if
    n ← getNextLeaf(p)
end while
renderToRoot(getLeafIn(H, L))
```

Figure 3.11: *ascentRender* ascends a range of leaves L to determine the highest subtree node H that is not as wide as $ascentWidth$. Once H is found, a path from L that is in the subtree under H is rendered towards the root, rendering H along the path. Here is a description of all variables and functions used: $ascentWidth$ is a global variable, as discussed in Section 3.2.2.3; subtreeWidth(N) returns the width of the subtree under node N; nodeHeight(N) returns the base grid line coordinate of N closest to the root; getNextLeaf(N) returns the leaf adjacent to the maximum leaf in the subtree under N; getLeafIn(N, L) returns some leaf $L_i \in L$ that is under the subtree of N; and renderToRoot(L_i) renders from leaf L_i to the root.

adjacent leaf in the next subtree, L_{i+1}, to start ascending from next, by using a constant time operation from L_i. If L_{i+1} is not in L, then we are done searching since A covers L and the leaf range adjacent to L. Otherwise, we follow L_{i+1} much like we followed L_s, updating H if necessary.

Once we find H, we render from any L under H to the root, stopping when we arrive at a previously drawn node. Figure 3.11 gives pseudocode for our ascent rendering function.

3.2.2.3 Ascent termination width

In the previous sections, we identify segment and ascent widths as important tree ascent rendering values. The segment width determines how many leaf ranges must be made for a given number of vertical blocks and the ascent width determines how to search for subtrees of a certain threshold width to produce a correct rendering for horizontal edges. Although we may choose any value less than one-half block for a segment width, we have identified neither the limitations for an ascent width, nor the effect of ascent width on our previous segment width restriction. This section identifies one last rendering problem for dense regions and how appropriate segment and ascent widths eliminate those drawing gaps.

In Section 3.2.2.1, we noted that segment widths must be smaller than one-half block to ensure no visible gaps occur in leaves. In Section 3.2.2.2, we use a rendering function that assumes rendering paths from leaf segments render into the same row of blocks as the leaf segment. However, upward paths in a subtree are not straight lines but, depending on the topology, may be very erratic. Similar to our reasons for segment widths bounded by one-half block, we do not know the position of subtrees ascended by our *ascentRender* function relative to on-screen blocks. If an ascent occurs close to a block boundary, there is the possibility of visible gaps in dense regions, as shown in Figure 3.12. We notice that this problem may occur when the sum of segment and ascent widths is larger than one-half block, for exactly the same reasons given for our original choice of segment width in Section 3.2.2.1. When the sum of these widths is less than one-half block, we guarantee gap-less rendering of paths in every block row.

One restriction to the ascent width is that the ascent width a must be at least as large as the segment width s, as shown in Equation 3.1. If the ascent width is smaller than the segment width, it is possible to miss the highest subtree node rooted in the leaf range, as shown in Figure 3.13. We want to maximize the segment width since larger leaf ranges result in fewer leaf ranges to process.

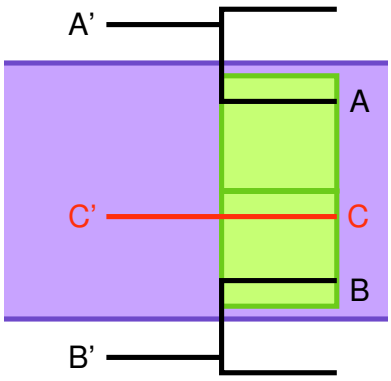

Figure 3.12: Illustration of ascent-related gaps for segment widths of less than one-half block. The blue region represents a block row, green squares represent the position and vertical width of two hypothetical adjacent leaf ranges, black lines represent drawn edges, and the red line represents an edge not chosen for drawing in the corresponding leaf range. If we choose to render leaves A and B, as shown in the figure, there will be no gaps at the leaf level for the blue block row. However, higher in the subtree at internal nodes A' and B', there is a drawing gap, where it would have been possible to draw internal node C'. This gap is possible even when A' and B' are not as wide as the segment width.

Adding the restrictions of Equation 3.1 and Equation 3.2, which states that the sum of the two widths is less than one-half block, we solve for the segment width s, in Equation 3.3, with respect to the block width b. To solve for the ascent width restriction a in Equation 3.4, we need to use the maximal value of s, $b/4$, with both Equations 3.1 and 3.2; a is exactly $b/4$. We arrive at an optimal solution of both segment and ascent width equal to one-quarter of the block width b:

$$a \geq s \quad \rightarrow \quad a - s \geq 0 \tag{3.1}$$

$$s + a < b/2 \quad \rightarrow \quad b/2 - s - a > 0 \tag{3.2}$$

$$b/2 - 2s > 0 \quad \rightarrow \quad s < b/4 \tag{3.3}$$

$$\textit{maximize } s \quad \rightarrow \quad s = b/4 \rightarrow a = b/4 \tag{3.4}$$

Again, similar to restrictions from Section 3.2.2.1, we do not have an increase

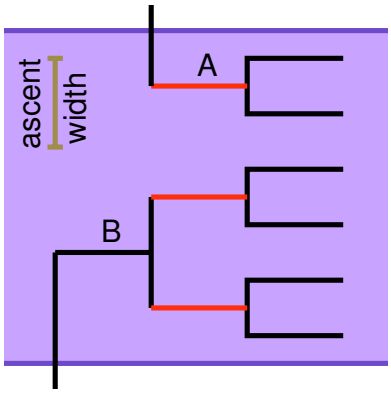

Figure 3.13: If the ascent width is less than the segment width, we may not find the correct horizontal edge in a leaf range. Using the figure, we ascend subtrees under nodes A and B in the leaf range highlighted in blue. If we ascend subtree A using the ascent width, given as the brown line on the left side of the figure, we terminate on the red line at the root of A; we may assume the parent vertical line of A is very long. Ascending B in the same manner, we find two more possible paths, also marked in red; the ascend rendering algorithm would find one of these to render since the root of subtree B is the first node in B that is wider than the ascent width. Our algorithm would choose among all red nodes to render, all equally likely depending on the traversal and layout methods used. However, we know that the root of B is not covered by subtree A, so we would see a horizontal gap where we would expect the root of B to be drawn. Therefore, the ascent width must be wider than the segment width, which would definitely select the highest subtree B that is contained in the leaf range.

in computational complexity by rendering twice as many leaves. Our pixel-based resulting rendering performance with quarter-block segment and ascent widths renders seven times fewer nodes than TJ1 for the large, non-binary *animalia* trees from the InfoVis 2003 Contest datasets [28], with only a small increase in the per node rendering time, as shown in Section 5.2.

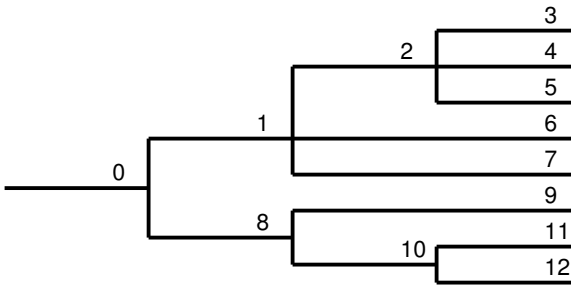

Figure 3.14: A sample node key assignment for a small tree. We can store the subtree under key 1 as the range $[1, 7]$ in a *RangeInTree*, or an individual node such as 2 in $[2, 2]$. Storing a range such as $[1, 8]$ is also valid and represents the subtree range $[1, 7]$ combined with node range $[8, 8]$. In TJ2 *RangeList* collection objects, which store several *RangeInTree* objects, *RangeInTree* objects are neither allowed to overlap nor be adjacent to other *RangeInTree* objects in the same collection; we collapse pairs of such ranges into single ranges when possible.

3.3 Marked ranges

In TreeJuxtaposer, **marked ranges** are necessary to define regions of interest such as computed differences, search results, user marked groups, and even mouse-over highlighted nodes. This section describes the methods used to store marked ranges for efficient performance of updates when marks change and efficient lookup techniques when marked nodes are drawn.

Nodes in TreeJuxtaposer are enumerated with **node keys**: pre-order, consecutive, monotonically increasing integers. This means that for every subtree in TreeJuxtaposer, the subtree root node key is smaller than every other node key in the subtree, and the entire subtree can be represented by a single range of integers, from the value of the root node key to the value of one of the leaf nodes in the hierarchy. An example subtree is shown in Figure 3.14. This numbering scheme allows us to efficiently store large subtrees as a pair of integers, in an object that we call a *RangeInTree*. A collection of *RangeInTree* objects is a *RangeList* and several *RangeList* objects are used in TreeJuxtaposer for operations such as marking,

resizing, and comparing.

For each *RangeInTree* object, nodes are stored either as a subtree or as single nodes with consecutive node key values. This method of compressing the amount of information, necessary to store common topological structures such as large subtrees, is also efficient for range checking operations such as concurrently deciding the color of several nodes. However, the node key assignment is permanent and does not allow keys to change after the initialization step. If a single leaf node is added or deleted, for example, too many nodes would have to be updated to be efficient. Future TreeJuxtaposer versions, which may support tree editing, will require new storage techniques that do not rely on the current node key values.

Each *RangeList* is initially assigned to a marking color, which can be changed with color selection panel, shown as small color swatches in Figure A.3. *RangeList* objects appear as marked with their assigned color in the tree topology; techniques such as guaranteed visibility, progressive rendering and label placement are used to ensure visibility of marked ranges as a priority over the normal nodes in the topology. Highlighted node colors are also priority based, which means mouse-over highlighted nodes are visible over user marked groups that are visible over search results that are visible over automatically calculated differences. When rendering trees, ranges of nodes to be drawn are searched for in each *RangeList* collection. Since the lookup process for determining node colors is common with a potentially large amount of data, storage and recovery of marked ranges for random sets of nodes must be optimal. This section will examine how ranges were handled in TJ1 in Section 3.3.1, and the changes to handling marked ranges in TJ2 in Section 3.3.2.

3.3.1 Marked ranges in TJ1

There are several inefficient techniques used to store marked ranges in TJ1. Since the TJ2 rendering process depends on efficient color lookup methods for all nodes being rendered, these techniques are no longer used in TJ2; I identify them here

to clarify the contributions of TJ2. The most notable techniques from TJ1 that we found to be inefficient were: *RangeLists* not combining adjacent or overlapping *RangeInTree* objects for automatically-marked node differences; *RangeLists* storing *RangeInTree* objects in lists; and *RangeLists* not storing nodes for implicitly user-marked nodes.

Overlapping and adjacent *RangeInTree* objects

If the *RangeList* collections were sorted lists, it would be possible to perform color lookup operations in time logarithmic to the number of items in the list with a simple binary search. However, sorting ranges that may overlap in a list is not trivial. One technique that would allow for easier sorting would be to combine all pairs of overlapping ranges into one single range; adjacent ranges such as $[1, 3]$ and $[4, 5]$ would also be considered overlapping and can be combined into the range $[1, 5]$. It is trivial to see the space efficiency of storing one range instead of several ranges for long lists of adjacent nodes, or removing non-unique ranges, but we would also become more time efficient in both searching a sorted list and searching for elements in a combined range.

Unoptimized *RangeList* collections

In TJ1, *RangeList* objects were simple lists of *RangeInTree* objects. Since the lists are not sorted, the color lookup operation, required for each node, has to search the entire list for an overlapping region. Although it is particularly expensive to look up a color for nodes known to be marked, unmarked nodes that are drawn also require color lookups for correctness. The inefficient methods of storage, which lead to inefficient color lookups, do cache color information for any node examined, while the user does not change any marks. However, due to the costs of color updates, this marking scheme does not scale beyond tens of thousands of nodes with many marked regions.

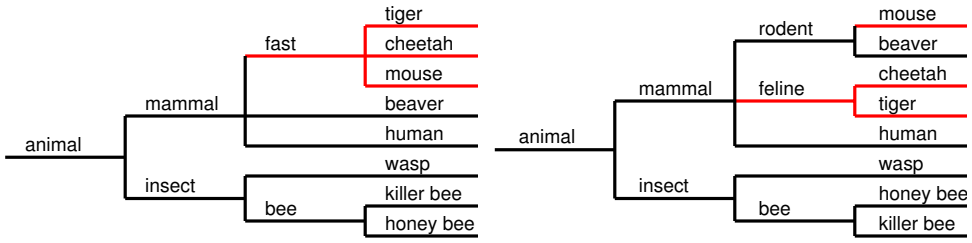

Figure 3.15: TJ1 only stores directly marked nodes to reduce the storage required. In the example, a user has marked the $fast$ subtree on the left, and on the right the indirectly marked nodes appear. TJ1 stores only the subtree $fast$ and does not store the additional two subtrees *mouse* and $feline$ from the right tree, but requires its color lookup code, which refreshes cached values after any marks have changed, to determine the colors of all nodes by searching for the corresponding nodes for each tree in each list of colors. TJ2 stores all three subtrees so determining colors in this way is not necessary. TJ2 color lookup methods are sufficiently fast during the rendering that per-node color caching is no longer necessary.

RangeList collections store only explicit marks

TJ1 only stores marked ranges that are explicitly marked. This means that for a two tree comparison, shown in Figure 3.15, if a user marks the $fast$ subtree on the left tree, only that subtree is stored in the *RangeList*. The $feline$ subtree and all other nodes marked in the right tree are not stored in a *RangeList*. TJ1 determines the marking color for each node when the node is rendered, using the best corresponding node for that node in every tree. Finding the marking color for nodes, after any marks have changed, is a slow operation that must perform a lookup for each node being drawn, but TJ1 caches node colors to prevent subsequent slow operations between marking. Although individual node marking for large numbers of nodes is not a common operation, automated marking that frequently changes the marked nodes, such as tree differences and search results, do not allow for rapid updates of marked regions for large trees.

3.3.2 Marked ranges in TJ2

There were several changes made to improve on the performance of the implementation of marked ranges in TJ1, most notably using a binary tree to sort and store *RangeInTree* objects. TJ2 no longer caches results for each node, since color lookup for ranges of nodes is sufficiently fast; we improve scalability by not caching colors for each tree node. The efficiency issues mentioned in Section 3.3.1 are dealt with in the following topics: *RangeLists* combining adjacent or overlapping *RangeInTree* objects; storing *RangeInTree* objects in binary trees; and *RangeList* storing nodes for implicitly user-marked nodes.

Combining adjacent *RangeInTree* objects

Automated marking from operations like computed differences and search results often return several adjacent, non-unique, or overlapping *RangeInTree* objects, all of which we refer to as **overlapping** ranges. TJ2 combines *RangeInTree* objects by searching the *RangeList* binary tree for overlapping ranges, combining any overlapping ranges with the *RangeInTree*, repeating the process until no more overlapping ranges are found, and finally adding the new non-overlapping *RangeInTree* to the *RangeList*. This repeated searching is necessary with the data structures we use for our binary tree implementation, namely the Java *TreeSet*, which cannot return the entire set of overlapping ranges in a single function call.

RangeList collections as binary trees

We sort the *RangeInTree* objects in a binary tree by their minimum node key values; the sorting criteria could actually use any node key in the range since there are no overlapping ranges in *RangeList* binary trees. Since each *RangeInTree* is accessible in time logarithmic to the number of marked items, the performance improvement is a dramatic improvement for large numbers of marked items, often resulting from hundreds of either differences or search results.

Another drawback to using the Java *TreeSet* class is there is no direct access function to retrieve members of the tree, so we developed a workaround built into the *RangeInTree* comparator function. We use one static *RangeInTree* object in the *RangeList* class, called `matchRange`. The comparison function for *RangeInTree*, *compareTo*(*Object*), stores the value of any overlapping range found in the binary tree by setting the value of `matchRange` to the node passed to the *compareTo* function, before returning `true` to the calling function. By accessing the `matchRange` object, we can get the first overlapping range from the binary tree for removing and further processing, as described in the previous section. This work-around allows us to use the built-in Java *TreeSet* data structures so we do not have to create our own binary tree implementation. Furthermore, we use this work-around in many places where binary trees are used in TJ2 and generic accordion drawing, saving the effort of having to repeatedly re-engineer binary trees.

RangeList collections store all marks

When a user marks a node or subtree, we call that a **directly marked** node or subtree, and the tree that this occurs in is the directly marked tree. An **indirectly marked** node is also possible when we compare with more than one tree, which occurs in all trees not directly marked, the indirectly marked trees. TJ1 does not store the indirectly marked nodes to attempt to save time performing bookkeeping, so it must recompute the indirect marks for each node after any marking changes occur, even if the marking does not affect the user marks. For TJ2, we perform the bookkeeping and attempt to store all marks, direct and indirect, to avoid unnecessary recalculations of node marks for *RangeList* collections.

After changing marks with multiple trees, TJ1 recomputes the cached colors for all nodes drawn in every tree; indirectly marked nodes are no exception. The colors for indirectly marked nodes are determined from their **best corresponding nodes (BCN)** in the directly marked nodes, stored in *RangeList* collections; the

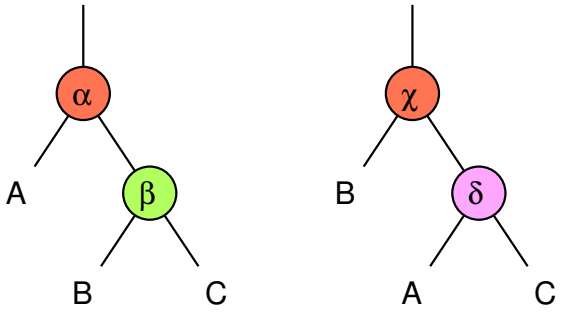

Figure 3.16: The best corresponding node (BCN) relationship between subtrees is not always one-to-one. To compute the BCN for a node from the left tree in the right tree, we need to find the node in the right tree that maximizes the value of number of similar leaves divided by the number of the union of leaves [24]. As an example, consider the figure with subtrees A, B, and C conserved between the two trees. The BCN for each of these subtrees on the left is always the corresponding subtree on the right; each BCN value is maximum at 1.0 since the set of similar leaves is the same as the union of leaves. The same is true of the roots of these trees, α and χ since both trees contain the entire set $\{A, B, C\}$. However, the BCN for β may not be δ but could be C if the size of A, B, and C are certain values. For the BCN of β to be δ, $\beta_\delta = |C|/(|\{A, B, C\}|)$ has to be greater than $\beta_C = |C|/(|\{B, C\}|)$, the BCN value of β for C. Setting $|A| = 1$, $|B| = 3$, and $|C| = 1$, these calculations become $\beta_\delta = 1/5$ and $\beta_C = 1/4$, meaning that the BCN of β is C on the right-hand tree. Therefore, directly marking C on the right-hand tree indirectly marks β as well as C on the left hand tree.

single color that appears for a given node is found by prioritizing the *RangeList* collections. This means that when TJ1 draws a node, it checks the BCN for each tree for a mark; we assert that there is either a single BCN or no corresponding node for each tree.

A simple, but incorrect, approach would be to find each BCN in the indirect trees for each node marked in the directly marked tree, and proactively mark those corresponding nodes prior to rendering. This method does not mark all nodes in the indirectly marked trees, as shown in Figure 3.16. Directly marking node C on the left tree should indirectly mark both the identical node C *and* its parent δ on the

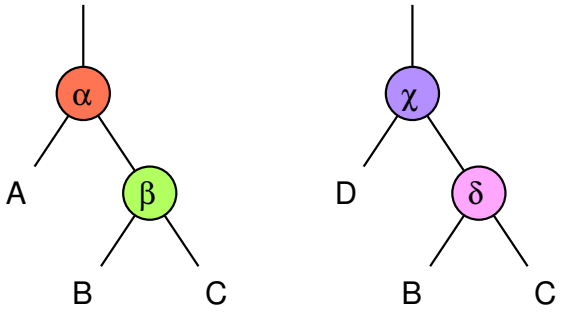

Figure 3.17: When comparing two trees with the same set of leaves, a directly marked single node may have a BCN in the second tree, but that second tree may not have *any* node that has the directly marked node as a BCN. For example, if we mark node α in the figure, its BCN in the right-hand tree is node δ ($\alpha_\delta = |\{B,C\}|/|\{A,B,C\}|$). However, δ has a BCN in the left-hand tree of β ($\delta_\beta = 1.0$ and $\delta_\alpha = \alpha_\delta$). This means that for the implementation of marks in TJ1, we would have not marked any node in the right-hand tree if α was marked. In TJ2, we perform a back-check from the BCN of α, which is δ, by determining if the BCN of δ, which is β, is marked in the left-hand tree. Since β is not marked, TJ2 does not mark δ; both TJ1 and TJ2 follow the same marking rules.

right tree. The BCN criteria is not one-to-one, so TJ2 requires a slightly different approach to mark the correct set of nodes.

Since the BCN relationship is not a one-to-one relationship, each potentially marked node must be examined in the indirectly marked trees for a correspondence to the directly marked nodes. We can avoid checking every node of all indirectly marked trees, by searching the neighborhood around the BCN for directly marked nodes in every tree. If we examine the BCN for all directly marked nodes, we expect to get a close correspondence to the set of indirectly marked nodes. However, there are cases when a directly marked node is not the BCN of any node from another tree, even with identical leaf sets, as shown in Figure 3.17. This means that when we examine the BCN of all directly marked nodes, we must back-check the potential indirectly marked nodes to some directly marked node.

The TJ2 marking process successfully finds all indirectly marked nodes with

an algorithm that performs back-checks on the BCN of all directly marked nodes. If an indirectly marked node is found in this way, the parents and children of that node are also examined recursively for BCN correspondences; indirectly marked nodes that are already marked are not processed further. The local checks are necessary due to the nature of the BCN method used. We notice that indirectly marked nodes are typically in the neighborhood of other indirectly marked nodes, since the BCN does not change dramatically in localized regions of a tree structure. An explanation of why the BCN is well conserved in localized regions is that the leaf set in a parent node includes the leaf set in all of its child nodes. Therefore, the BCN of a directly marked node is typically topologically close to a sufficiently related neighborhood of nodes, which simply means the BCN value of a directly marked node is never zero. When a node in the indirectly marked tree is found to not correspond with any directly marked nodes, such as the example in Figure 3.17, we do not process the parent or child nodes of the BCN.

Note that it is also possible to mark a single node on one tree, and have more than one indirectly marked node appear on a second tree. Referring to Figure 3.16 with the subtrees A, B, and C the same size as in the original example, if we mark node C on the left, C *and* δ on the right are marked. This behavior is correct given our definitions, although it may appear confusing when this is not expected, especially to those who do not know the subtleties of our BCN algorithm. This behavior never appeared in TJ1 due to faulty marking in the color caching process that showed computed differences over all marked nodes, and no colors for indirectly marked nodes when differences were turned off.

3.4 Topological picking

Users perform navigation in TreeJuxtaposer using a mouse, so when the cursor is close enough to a tree node, we want to indicate to users that the node has been selected, or picked. We treat node picking as a simple case of node marking

by highlighting all BCNs of a selected node when we compare trees; unlike real marking, we do not perform the back-checking operations, described in Section 3.3. Since a tree node may be drawn as a single pixel, in either horizontal or vertical directions, we allow picking to be within a margin of error, which we call the **picking fuzz**. The picking fuzz deals with the speed versus accuracy tradeoffs associated with the exact aiming a pointer at a target, which is known as Fitts' law [10], after the famous study by Paul Fitts.

We allow the user to be within a distance of five pixels from a selectable target with our picking fuzz, but also understand that it may not be possible to disambiguate an intended target in regions where many possible selections are valid. Therefore, we rely on a user to stretch the region of interest if a desired node is not pickable with our technique. Our main concern is that users should always be able to pick a node if there is a single, definitive choice for selection when a mouse pointer is in a screen location close enough to pick it.

TJ1 is able to pick most nodes of trees, using quadtree structures, but could get stuck trying to pick certain sub-pixel tree nodes [5]. These nodes, often in vertically very small grid cells, are usually adjacent to a cell of the quadtree that was descended but was not able to pick a node. Quadtree cells that are descended are the most likely candidates since the current mouse location is in the correct quadtree cell quadrant, but no tree edges are in that quadrant within the picking fuzz distance to the mouse location. However, the non-descended nodes in an adjacent quadtree cell could have been near enough to the mouse, but these nodes were already discounted by the quadtree picking algorithm. The quadtree picking algorithm lacks back-tracking capabilities to search other quadtree cell candidates.

An important design concept is that picking algorithms should be structurally similar to rendering algorithms. The similarity assists in providing intuitive picking with visible objects: visible objects can be picked and all pickable objects are visible. As shown in our picking algorithm in Figure 3.18, TJ2 uses the cell layout described

Picking Function
input: mouse screen position $M = (X, Y)$
root TreeNode $T = (kids, cell)$ where
$kids = \{T_0, T_1, \ldots, T_{n-1}\}$
$cell = (X_{min}, X_{max}, Y_{min}, Y_{max})$
output: picked TreeNode $T_{(X,Y)}$, a node close to (X, Y)

```
stack S ← ∅
S.push T
while S ≠ ∅
    N ← S.pop
    if (X, Y) over edge of N
        then return Nend if
    xMin ← N.cell.X_min
    if N.isLeaf() or N.cell.bounds(Y) or xMin > X
        continue        end if
    k ← BinarySearch( N.kids, Y )
    if k > 0
        S.push N_{k-1}    end if
    if k < n − 1
        S.push N_{k+1}    end if
    S.push N_k
end while
return ∅
```

Figure 3.18: *Picking* function that descends tree T from the topological root node of T until a tree edge close enough to mouse coordinates (X, Y) is found. A stack S is used for backtracking if a descent is unable to find a tree edge; at each step of the descent, the siblings to the immediate left and right of the next node to be checked are pushed onto S. We use binary search to select the next node for descent, if appropriate, using $N.kids$, the children of the current cell, and the mouse Y coordinate. For every function that we use for distance comparisons, including: BinarySearch; Y within the cell of N; and mouse over edge of N, we apply a picking fuzz, to satisfy Fitts' law.

in Section 3.1.1 to descend the cell structure until it finds the cell that contains the mouse pointer. However, as the algorithm descends in the tree hierarchy, it adds the immediate left and right siblings to a stack. If the algorithm is unable to find a node in the hierarchy after descending, it pops a node off the stack and continues descent searching with that node. Some subtleties of this algorithm are the stopping criteria, how child nodes are selected for descent, and how the picking fuzz is used to allow descent on siblings that are close enough to the mouse pointer.

The algorithm works by checking the bounds of the current grid cell with the mouse pointer coordinates. If the vertical range of the current cell contains the mouse pointer, then we know that a potential selected node is in one of three places: an ancestor of the current cell, the current cell itself, or a descendant of the current cell. Since we start descending from the root cell, we know that once we process a parent cell and determine that the horizontal mouse coordinate is spatially lower in the topological tree hierarchy, the selected cell is either the current cell or some descendant. Finally, if the current cell does not contain the horizontal mouse pointer, we know that a descent is necessary. Our stopping criteria for picking would therefore be that there are no pickable nodes in the cell that the mouse pointer is found in after a sufficient horizontal descent; the algorithm would then use the stack to continue searching.

TJ2 is able to deal with n-ary trees, so picking a child to descend is not trivially "left or right" as it would be in a binary tree. We know that the current node being examined has some descendant node that has a cell which contains the mouse pointer. To find the appropriate child to descend, we perform a binary search on the child nodes, using the mouse pointer location as our searching value. Once we select the child node for descending, we push its immediate siblings onto the back-tracking stack.

We use the picking fuzz to descend siblings that do not exactly bound the mouse coordinate with their vertical cell range. We know, when descending the

topology, that if we do not find an appropriate node for picking when we reach the end of our criteria, we need to search with the back-tracking stack. It is sufficient to place only one sibling in each direction for a descent since the adjacent cells are not empty and the adjacent edges are either within the picking fuzz or too far to pick. Finally, a node will only be pushed onto the stack at most once; back-tracking would follow a different path that could not possibly re-select nodes from previous descent attempts.

Our picking algorithm requires time linear in the height of the tree. This time complexity is not a problem for most tree types, and picking has been shown to be sufficiently fast on the deepest trees TJ2 is currently able to support, which are over 1000 nodes deep. One important note about picking in deep trees is that recursive picking methods quickly run out of stack-frame memory, which is why our methods use a Java-based stack that we can place on the heap. Other methods that use recursion on the height of the dataset topology, such as the tree parsing library, should also be written without recursion, and are currently the limitation to the depth of trees we would otherwise be capable of loading.

Chapter 4

Accordion Drawing

This chapter describes the advantages of using an Accordion Drawing (AD) infrastructure to develop new information visualization applications. AD applications have features such as guaranteed visibility, global Focus+Context, and progressive rendering, which all aid in the understanding and analysis of many different dataset types. We can easily develop new AD applications with these key information visualization features with a minimal amount of work in non-application specific functionality.

In this chapter, I focus on our improved motion algorithm for AD grids, which is numerically stable and correct over large amounts of grid movement. I also describe the split line hierarchy in detail, as well as how we use the hierarchy to efficiently perform generic operations used by applications such as TJ2. I then present the details for a single split line motion in our grid hierarchy, which is shown to be capable of several key features to ensure order, stability, and efficiency in our split line hierarchy. Finally, I present our algorithm that allows for concurrent motion of several split lines with minimal split line hierarchy updates. Our new algorithm is just as efficient as TJ1 motion, and ensures that TJ2 motions do not cause ordering inconsistencies in our split line hierarchy, which are present in TJ1 methods from lack of numerical precision.

In the remainder of this section, I details the AD mechanics from TJ1. Then

in Section 4.1, I describe our reshapable split line infrastructure. Section 4.2 describes our generic rendering infrastructure for pixel-based rendering, and Section 4.3 describes numerically stable AD navigation.

General accordion drawing mechanics

I begin by discussing the general mechanics of AD that persist between TJ1 and TJ2 implementations. Although both applications consider tree topologies for rendering, this section focuses on a more general approach of a reshapable grid.

We grow and shrink areas on rendered datasets using movable lines in a two dimensional plane, which has a growing effect for one region while shrinking the region on the other side of the moving line; the horizontal and vertical lines are independently movable boundaries of **interaction boxes**. Growing or shrinking is performed on the base grid of such lines, the set of all lines that form the grid, called **split lines**. When we grow or shrink an interaction box region between a pair of split lines, the AD infrastructure grows or shrinks the areas of cells between each pair of split lines in the region with equal ratios. This equal ratio can be seen in Figure 4.2, where a split line has vertically squished the region below a moving split line while vertically stretching the region of interest inside the interaction box. Interaction boxes themselves are a rectangular arrangement of a set of base grid cells, which are the smallest individual regions of space on the base grid bounded by four split lines.

Figure 4.1 shows a uniform split line grid of base grid cells, the typical initial state of an AD application, with an interaction box that I have selected. There are no restrictions on the initial properties or distribution of split lines for applications; developers of applications are responsible for defining their own split line arrangements if a uniform grid is not desired. After the base grid is created with application-specific dimensions, applications typically lay out and draw a canonical, uniformly scaled view of their datasets on the grid. Typically, the split lines them-

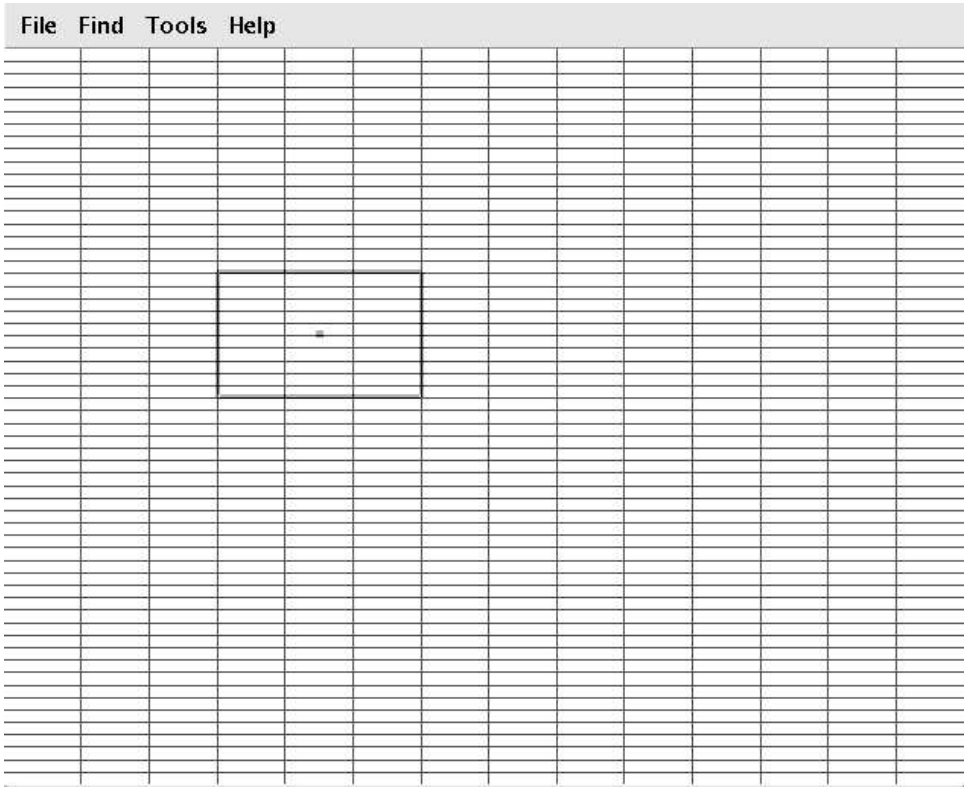

Figure 4.1: An initial uniform split line layout for AD applications, which appears as a grid of base grid cells separated by split lines. I have selected an interaction box, shown in grey in the upper left corner of the grid with a dot marking the center of the box, which is shown stretched in Figures 4.2 and 4.3.

selves are not drawn as shown in the figure, and application-specific nodes appear within regions of base grid cells.

Figures 4.2 and 4.3 show the interaction box from in Figure 4.1 stretched in the vertical direction and stretched in both directions, respectively. When we shrink an area of base grid cells, data in that area may be compressed to a size that is smaller a block, the smallest feature size for drawable elements. AD applications handle over-compression of drawable data with culling, or choosing a representation of the data for that compressed region. The AD framework conveys information to the application-specific drawing procedures about the position and size of base grid

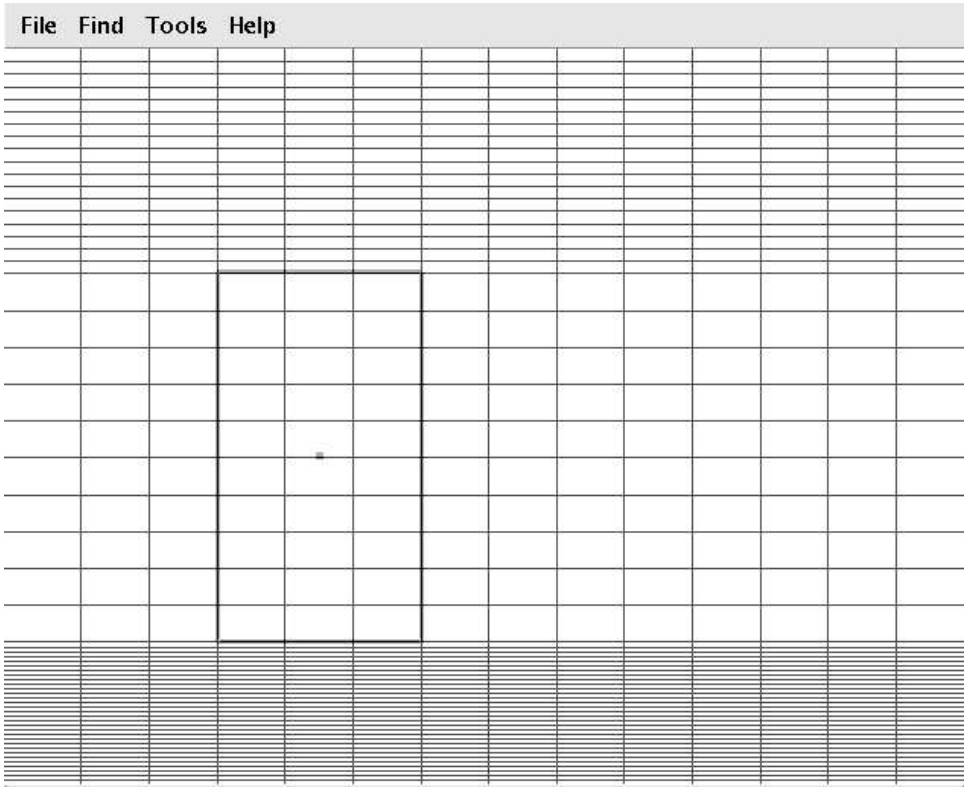

Figure 4.2: The interaction box in the grid of Figure 4.1 stretched vertically towards the bottom of the display. This stretching: does not affect above the interaction box; stretches between the top and bottom edges of the interaction box; and compresses below the interaction box. All stretching is uniform over each of the distorted regions.

cells, and the application determines how to draw in the current state of the base grid.

4.1 Split line infrastructure

In TJ1, a version of TreeJuxtaposer that featured the original implementation of AD, spatial subdivisions of the navigation space are created from a quadtree structure. Each TJ1 quadtree cell, a quad-cell, stores a pair of split values used to allocate

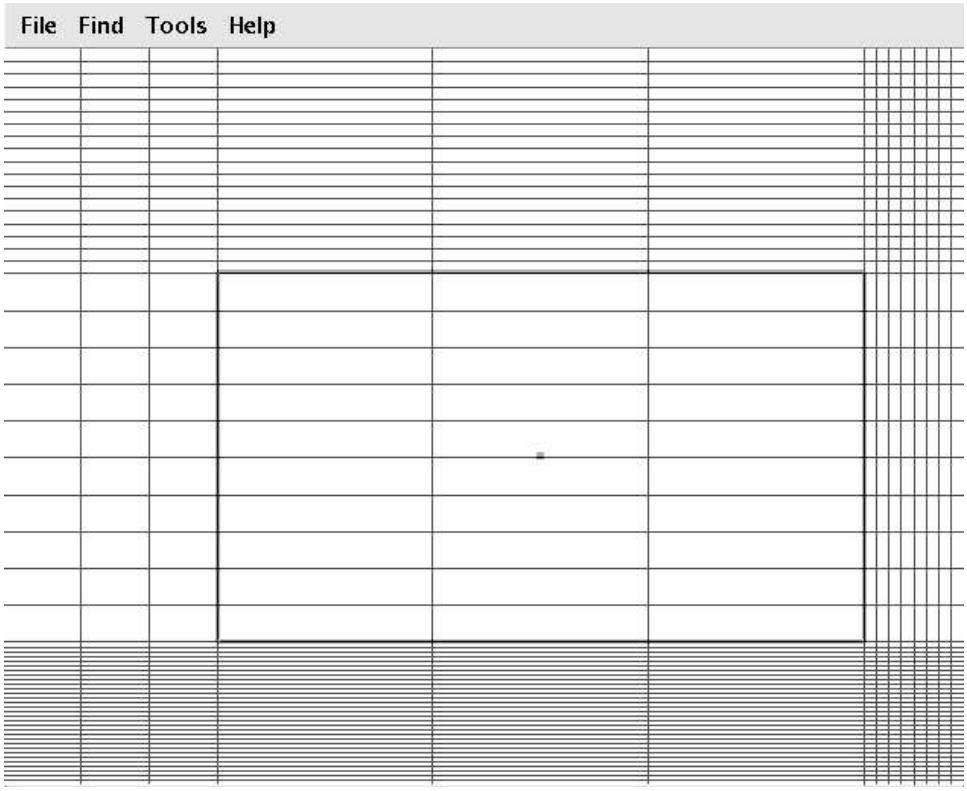

Figure 4.3: The interaction box from Figures 4.1 and 4.2 stretched both vertically and horizontally towards the bottom-right of the display. Notice the stretching does not affect the grid in the section above and to the left of the interaction box, but has been stretched and shrunk in other regions of the display adjacent to, and inside of, the interaction box.

space to child quad-cell nodes; one value is for a horizontal split line, the other for a vertical counterpart, as shown in Figure 4.4. The split values, between 0 and 1, are a relative offset with respect to the quad-cell boundaries. Split lines are global grid divisions as shown in Figure 4.1, and quad-cells reference the split lines for their boundaries and their movable split line. Several quad-cells reference the same split line since many parts of the quad-cell hierarchy descend into similar regions. For example, the quad-cells that divide the leaf nodes all reference the split line that defines the right edge common to all leaves. However, TJ1 only caches

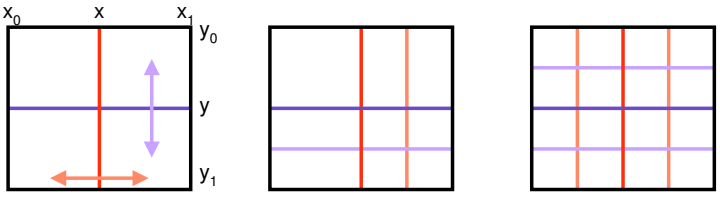

Figure 4.4: A quad-cell structure from implementation of AD in TJ1. **Left**: the split lines x and y, along with the boundaries of this cell, define the boundaries for the four child quad-cells of this cell. The split lines are movable and each quad-cell maintains the location of its two component split lines with a relative split value; the split value is a normalized ratio between zero and one. The split value for the horizontal subdivision in a quad-cell, for example, is the value of $(x - x_0)/(x_1 - x_0)$, which is the relative distance between the split line and the minimum boundary with respect to the size of the cell. The relative split values are used to compute the absolute location of split lines during rendering. **Middle**: the recursive structure of quad-cells means that the quadrants of any quad-cell can hold one quad-cell each. The bottom right cell has been subdivided, producing four new quad-cells out of one larger quad-cell. The new quad-cells use x, x_1, y, and y_1 as their boundaries. **Right**: all four quadrants subdivided. Notice how the bottom right and top right subdivisions share the position of a vertical subdivision. Quadtrees are less efficient for AD than a simple grid because of these redundancies.

split line positions for grid quadtree cells and does not cache the positions for split lines themselves; any split line that is referenced by more than one quadtree cell is updated with the same value for each reference.

Another quadtree inefficiency, described by Beermann et al. [5] for trees but applicable to any AD application, is that each quad-cell wastes substantial memory because they are all identical structures, with four pointers to child quad-cells. They point out that quad-cells are used unnecessarily at the leaf tree node level; leaves are the lowest quad-cells, do not have children, and are the majority of all tree nodes in a typical dataset. Creating a new type of quad-cell for those tree nodes saves memory in their implementation of quadtree-based TJC.

However, Beermann et al. also present a second approach that uses a simple grid layout for spatial subdivisions. Their grid method extracts the split lines from

the quad-cells and removes quadtrees entirely with grid-cell objects, tree nodes in TreeJuxtaposer, defined by their bounding split lines in the grid. The use of a regular grid layout for spatial subdivision, which is shown to load trees that are three times larger in an equal amount of memory, provides a convincing argument for not using quadtrees for AD in general.

We generalized the AD infrastructure from TJ1, and TJ2 is built using this generic API, as discussed in Chapter 3. Our infrastructure improvements are detailed in this section: abstracting the split lines from application specific topological layouts in Section 4.1.1; generalizing horizontal and vertical split line components providing a flexible API for new AD applications in Section 4.1.2; and storing the split lines in a tree hierarchy for efficient updates in Section 4.1.3.

4.1.1 Abstracting split lines

In order to use a grid-based generic navigation infrastructure, we need to ensure applications are capable of performing critical tasks such as layout, rendering, culling, and picking. All of these four tasks have application-specific components, but since each task depends on location of data in our grid structure, we must provide support each task with our AD infrastructure.

Dataset nodes, entities of datasets that provide the lowest-level dataset details, are assigned to a grid cell, which is a rectangle described by four split lines for the top, bottom, left, and right sides. Each node of a dataset is typically assigned to a grid cell when the dataset is initially loaded; this is not a requirement and the dynamic assignment of dataset nodes to grid cells is an interesting area of future work. In my thesis, we will restrict node layout in AD by only permitting layouts on grids with known dimensions; the parsing process for TreeJuxtaposer can determine how many horizontal and vertical split lines are necessary. When laying out data in AD, we place nodes in the grid where necessary, as described in Section 3.1.1, by partitioning the grid in a much more flexible manner than the methods in TJ1.

Other AD data mapping techniques used by PowerSetViewer, presented in [25], are capable of dynamic layouts of data, but this topic is beyond the scope of my thesis.

Rendering a dataset and culling data elements are related tasks since culling is a function of the rendering process; all rendering requires knowledge of the location of a specific item in a particular region of screen space. Our rendering approaches in TJ2 are topology-based, but the dataset-specific rendering functions collect information about screen position and node size using the cell layout. As mentioned in the TJ2 rendering section, Section 3.2, a node is rendered only if the cell in which it renders in is larger than a culling limit. The AD infrastructure assists the application-specific topology-based rendering and finds the culling limitations for ranges of nodes stored in ranges of split lines. For example, the infrastructure provides TJ2 with the desired segmentation-width partition of grid cells used to cull leaf ranges into single leaf renderings.

Picking is topology-based for TJ2, as shown in Section 3.4, but for application datasets that may lack an inherent topology, we want to use the infrastructure of split lines to pick dataset objects. Section 4.1.3 describes a hierarchy that we may use for generic picking when datasets are unstructured.

4.1.2 Separate horizontal and vertical split lines

Quadtree-based AD applications, such as TJ1, combine horizontal and vertical components in one data structure. Quadtree AD makes development of applications that either only require one-dimensional AD, or commonly have datasets with very mismatched quantities of horizontal and vertical split lines, difficult or inefficient to implement. Figure 4.5 shows how two one-dimensional arrays of split lines contribute to the two-dimensional grid structure of AD applications such as TJ2.

The quad-cell structures used by TJ1 may be modified to use one-dimensional accordions, but were optimized for two-dimensional, planar AD. Beermann et al. [5] show that there are several advantages to using one-dimensional data structures for

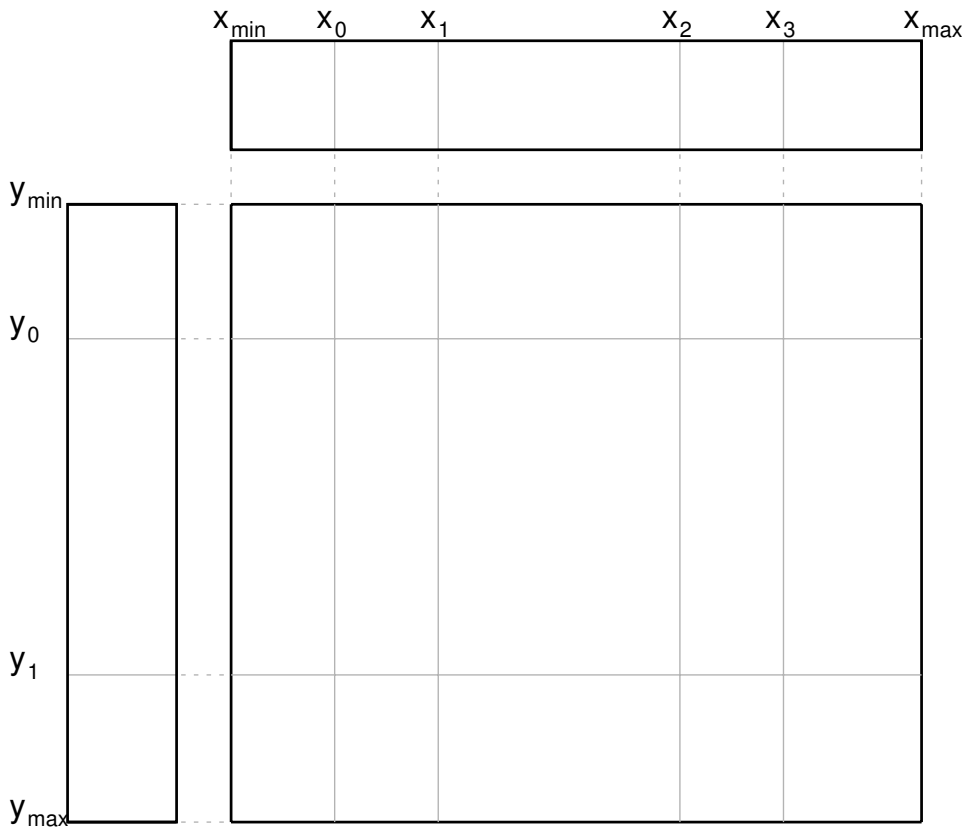

Figure 4.5: The combination of the horizontal x split line set with four movable split lines and the vertical y split line set with two movable split lines forms a grid of fifteen split line cells. The grid formed is the spatial subdivision used in TJ2; compare this grid with the subdivision method of quadtree cells in TJ1, in Figure 4.4.

split line storage with two TreeJuxtaposer reimplemented applications called TJC and TJC-Q. Their most substantial results in memory reduction were in TJC, which distinguished the horizontal and vertical split lines as separate entities. TJC is three times more memory efficient than TJC-Q, their version of TreeJuxtaposer that uses quadtree structures.

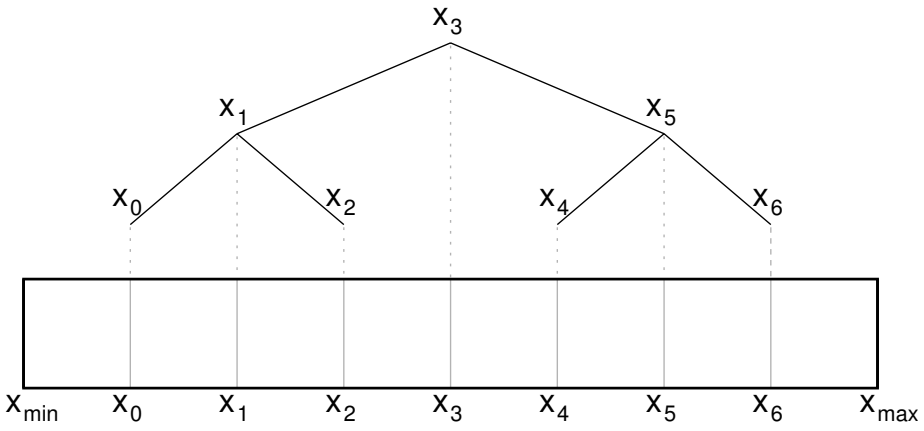

Figure 4.6: The split lines x_0 through x_6 are stored in a balanced binary tree hierarchy; the boundary split lines x_{min} and x_{max} are not stored in the hierarchy. This storage is analogous to the quadtree hierarchy in TJ1 where each cell of the quadtree stored a pair of relative split positions. The one-dimensional storage of split lines in TJ2 is more flexible than quadtree storage, allowing applications to be created that only require one-dimensional accordions.

4.1.3 Tree hierarchy for split lines

Split lines are stored in a balanced tree hierarchy. Upon determining the number of split lines necessary for a particular accordion direction, horizontal or vertical, we create a binary tree, as shown in Figure 4.6. The binary tree is organized with the center split line, the split line with half the number of total split lines on either side, as the root. Recursively, the tree represents progressively smaller regions to either side of a central split line.

The split line tree operates hierarchically much like the quadtree structure in TJ1. With this structure, split lines can be interpreted as either lines or regions, as shown in Figure 4.7. D is free to move inside the largest red box, since it is bounded to movements within the boundaries of the entire visualization. The two child split lines, B and E, split the regions left and right of D, and other split lines further split those regions. Movements of a B are bounded by its parent; it is free to move only in its brown box, which is always bounded on the right by D, and on

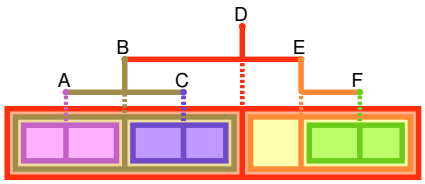

Figure 4.7: The split line boundaries for a simple seven split line example show how split lines can be represented by lines in a hierarchy or recursive-bounding regions. Each split line is color coded, bounded by a region of the same color, can move left and right in its box, and cannot leave its region. Moving a split line in the lowest levels of the hierarchy, A, C, or F, does not affect the absolute positions of any other split lines. Moving B, however, affects the absolute position of A and C, which share B as a boundary; when B moves, the relative position of A and C in their respective regions does not change, but the size of the regions change with B. Furthermore, moving D affects *all* split line absolute positions, even A and F, which do not have D as a boundary. The raw movements of split lines, as described, are transparent to applications, which only request legal split line movements in absolute $[0, 1]$ screen coordinates.

the left by the boundary split line. C, the right child of B, is bounded by both D and B, and so on. Neither B nor C may cross over each other, and neither may cross over D; split lines always remain ordered and never leave their boundaries.

As lines, split lines are ordered in the hierarchy between their adjacent neighbors and can be indexed as such; the ordering of split lines never changes after initialization. As regions, split line domains, halving for each layer in the hierarchy with the root representing the entire domain, represent their region of movement for both the split line contained within and, recursively, the domains of their descendants. The split line hierarchy is a critical structure for the contained lines and regions since the hierarchy is used to both calculate the absolute position of lines for cell positions and control the navigation.

Each split line stores a relative position between its boundaries, in its domain. To compute an absolute screen position, the relative positions of all ancestors of a split line are required; the absolute value is cached and only computed on demand

AbsolutePosition Function
input: split line S
output: screen position in [0,1]

```
pos ← S.getRelPos()
while S ≠ root
    P ← S.getParent()
    if P.isChildLeft(S)
        pos ← pos × P.getRelPos()
    else
        pos ← (pos × (1 − P.getRelPos())) + P.getRelPos()
    end if
    S ← P
end while
return pos
```

Figure 4.8: *AbsolutePosition* function that ascends the split line hierarchy from split line S to determine the position of S relative to the visualization boundaries. The function $getRelPos()$ returns the relative position of a split line and $P.isChildLeft(S)$ returns `true` if S is the left child of P in the split line hierarchy. In practice, this function is recursive and the absolute positions of all split lines are cached as they are computed.

when split lines have moved. The recursive calculation of the absolute location of a split line is shown in Figure 4.8.

4.2 Generic AD rendering infrastructure

Rendering in AD applications is a discretization process that maps the infinite-precision drawing of an object to block-level precision. The core rendering algorithm is linear in the number of blocks, unlike AD in TJ1, which was more dependent on the size and structure of the dataset. More specifically, generic AD methods perform TJ2 tree rendering with a time complexity of $O(b_v \times d)$, where b_v is the number of vertical blocks and d is the maximum depth of the tree topology.

AD rendering mechanics operate with a three-stage structure: **partitioning**, discussed in Section 4.2.1, where an application-specific set of split lines is divided

into renderable ranges; **seeding**, discussed in Section 4.2.2, where the partitioned split line ranges and marked ranges are arranged in an order appropriate for drawing; and **drawing**, discussed in Section 4.2.3, where a set of nodes is drawn for both the marked ranges and each partitioned, seeded split line range. Although seeding and drawing are more application-specific than partitioning, the general structure of all three stages follow a set of basic functional constraints for each AD application.

4.2.1 Partitioning stage

Each AD application that uses two independent sets of split lines to form a grid, like the horizontal and vertical split line sets used in TJ2, must decide which set to partition. For tree drawing applications, it is only possible to partition in the direction of the leaves since leaves may be followed to their set of ancestors, or ancestors to their descendants. The orthogonal direction in TJ2, in the direction of the topological height, has no linked structure analogous to topological associations. Other AD applications, that do not render a tree structure, such as SequenceJuxtaposer [35], would of course follow any imposed hierarchy or use knowledge of the application domain to determine which split line set to partition; determining the most appropriate set to partition is beyond the scope of my thesis.

Also associated with the application domain is the maximum partition size, also known as the segment size, often associated with the block size, which is the minimum feature size for application drawing. In the specific case of TJ2 tree rendering, for example, a segment size of one-quarter the block size is required for gapless rendering, as discussed in Section 3.2.2.

Once an application requests the partition of a split line with a segment size, AD begins a process that recursively descends the split line hierarchy until the split line domain width is smaller than the segment size; the first partitions smaller than the segment size are stored in a partition list. If a descent in the split line hierarchy reaches the leaves of the split line hierarchy without finding a split line domain

smaller than the segment size, the single split line leaf is enqueued in the partition list.

4.2.2 Seeding stage

Seeding the partitioned list of split line ranges into the rendering queue is the second stage of AD rendering. The seeding process is the key component of AD that provides progressive rendering support. Drawing important objects first, which is customizable for differing application domains, shows landmarks in the visualization and allows user-directed interaction on partial scene renderings. Applications that do not use progressive rendering techniques for interaction, or render the entire scene in a single frame, do not require an explicit drawing order but are seeded similarly. The seeding process is an $O(b_v + m)$ process where b_v is the number of blocks in the partition and m is the number of marked groups.

Prior to rendering a scene, the rendering queue is initially populated with the set of marked nodes. Next, the application adds each of the partitioned ranges in order. If a user interaction box is present, the application prioritizes the partitioned ranges corresponding to that region by placing them before all other ranges in the rendering queue. Although our naïve seeding method iterates through each of the partitioned ranges to add them to the rendering queue for this process, this does not add a large time overhead; non-progressive rendering should not perform the iteration and could gain performance increases, but non-progressive rendering optimizations are an area of future work and not analyzed here. This seeding order is expected by the next stage in AD rendering: the drawing process.

4.2.3 Drawing stage

Drawing is the final stage in our generic AD rendering infrastructure. Using the seeded queue from the previous stage, each marked range, described for TJ2 in Section 3.3, is rendered immediately. Since fast rendering techniques, such as marked

range skeletons for TJ2, or simple aggregations for contiguous marked ranges are used, AD attempts to draw each marked range in a single frame.

For a large number of marked ranges, the brute-force marked range rendering techniques may take too long to render, but adding progressive rendering time checks may impede this rendering too much. As shown in Section 5.4, marking a whole $190,265$ node tree while compared to a $198,623$ node tree in TJ2 with many differences adds about 200 milliseconds to the rendering time for both trees. Without adding progressive rendering to marked region drawing, we could seed fewer marked ranges, like TJ1, but this is another area of future work.

After marks are drawn according to the seeding queue, AD drawing draws the dataset nodes, one set of nodes per range of split lines, in the seeded order. Drawing from a one-dimensional split line range into a two-dimensional grid is another application-specific process. The node drawing for TJ2 is described in detail in Section 3.2, which describes how tree rendering starts from ranges of leaves and renders towards the root node.

Applications that do not render trees may use one split line range as an outer-loop and the second split line range as an inner-loop for an iterative rendering process over the base grid surface. By partitioning along both sets of split lines, such applications may aggregate their datasets into a coarse grid of blocks that can be rendered in $O(b_v \times b_h)$ time, for a horizontal number of blocks b_h and vertical number of blocks b_v. Identifying other interesting topological structures for drawing in new AD applications, such as applications like SequenceJuxtaposer [35], is left to future work.

4.3 AD navigation

This section describes the user-driven distortion-based navigation of AD. The generic base grid structure undergoes distortions similar to the methods used in TJ1 AD, but our methods are more numerically stable. In this section, I first describe how

a single split line can move in the split line hierarchy. The techniques for moving the line are discussed, with emphasis on how the movement transaction achieves numerical stability and its correct movement position. I then describe an extension that allows multiple simultaneous split line motion in AD, again with correctness and stability analysis.

4.3.1 Moving one split line

We accomplish navigation and zooming in AD applications by repositioning split lines in such a way that the cells on one side of the split line appear to stretch, while cells on the other side are squished. We perform the stretching and squishing actions according to our hierarchical split line tree, which provides an algorithm for motion that performs with time complexity of $O(log(n))$, where n is the total number of split lines. The case of moving one split line, the target, from an initial position to a final position within the range of the split line boundaries, is the basis for all navigation in AD applications.

Unlike TJ1, our algorithm descends the split line hierarchy tree towards the target split line instead of ascending towards the root, moving each split line encountered to its final position. The final relative local position of each of the $O(log(n))$ split lines in the path is calculated in $O(1)$ time with linear interpolation between the target and an ancestor split line, where n is the total number of split lines in the hierarchy. Once our algorithm reaches the target split line in the hierarchy, we move it to its final position and the recursion stops. The algorithm, *moveSingleSplitLine*, is shown in Figure 4.9.

In order to show generic movement in AD works, we show that the following four properties hold for the motion of a split line, when we move a single target split line to some final position:

1. a target split line can be moved anywhere in the bounds of the window;

2. all split lines remain ordered during a transition;

moveSingleSplitLine Function
input: split line S, at its initial position $S.i$
output: S moved to $S.f$, the final position of S

$L \leftarrow minBoundary$
$R \leftarrow maxBoundary$
$C \leftarrow getCenterSplit(L, R)$
while $S \neq C$
  if $C.isChildLeft(S)$
    $C.f \leftarrow \left(\frac{C.i - S.i}{R.i - S.i}\right) \times (R.f - S.f) + S.f$
    $R \leftarrow C$
  else
    $C.f \leftarrow \left(\frac{C.i - L.i}{S.i - L.i}\right) \times (S.f - L.f) + L.f$
    $L \leftarrow C$
  end if
  $C.moveFromTo(C.i, C.f)$
  $C \leftarrow getCenterSplit(L, R)$
end while
$S.moveFromTo(S.i, S.f)$

Figure 4.9: *moveSingleSplitLine* function that descends the split line hierarchy from the domain between the $minBoundary$ and $maxBoundary$ split lines, where the root split line is bounded, to the target split line S. At each step, the center C is found in its domain [L, R]. If C is the target S, we move S to its final position with $moveFromTo$. Otherwise, the final position for C, $C.f$, is calculated depending on the location of S relative to C. C is then moved from $C.i$ to $C.f$, a new boundary [L, R] is created with C, and the process continues until S is found. All positions i and f are global, relative to $minBoundary = 0$, and $maxBoundary = 1$.

3. each motion step positions half of the remaining split lines;

4. cells may become stretched when they should be squished during a transition, but they are in the correct final position when the algorithm is finished.

Property 1: Target split line can move anywhere in visualization

A user must be able to move any split line from any starting location to any ending location within the domain of the entire visualization. We must ensure that the split lines are movable enough, without breaking our ordering restriction. Suppose

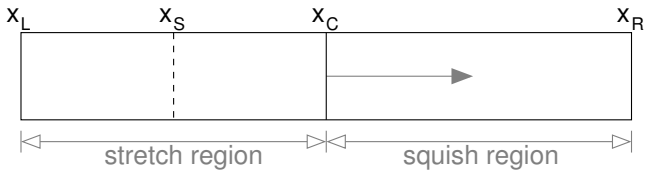

Figure 4.10: The absolute distances between split lines in a region being stretched grow with respect to the distance that split line moves away, but the relative distances between all split lines in $[x_L, x_C]$ do not change when split line x_C moves towards x_R. As x_C moves, x_S moves away from x_L since x_S is in a stretch region, but $\frac{x_S - x_L}{x_C - x_L}$ remains constant. Since the resizing of cells, caused by movement of x_S, is uniform on both sides, *moveSingleSplitLine* from Figure 4.9 can complete half of the split line movements for $[x_L, x_R]$ in each step of its outer loop.

a split line, x_S, moved from near x_L to near x_R, in Figure 4.10. We see that this is possible by moving the ancestors of x_S all towards x_R, which drags x_S towards x_R. This extreme action also conserves the relative distances among all nodes on either side of x_S, so if we drag x_S back to where it was initially, there are no distortions in the nodes between either x_S and x_R, or x_L and x_S.

Property 2: Split lines remain ordered

Observing Figure 4.7, we note that the red central split line D is capable of moving in its own domain, and every other split line under D in the hierarchy is either squished or stretched in its respective half of the domain of D. Since the *moveSingleSplitLine* algorithm in Figure 4.9 only moves either S or C in its domain, it is not possible for those movements to exit their boundaries or cause other split lines to become disordered. The recursive division step of using C to form the new boundary is further protection from moving split lines out of order.

Care must still be taken in practice, however, since numerical roundoff errors have been observed with deep recursion into infinitesimally small cells. We do not have stopping criterion for preventing numerical errors, but have eliminated errors

with our densest datasets by imposing a limit on squishing the cells. Currently, we prevent a user from squishing any region of the visualization beyond one percent of the width or height of the drawing canvas, so regions cannot be squished out of view, which provides a minimum rendering size and guaranteed visibility of all regions. One positive side effect of the minimum rendering size prevents numerical errors that might occur when regions are squished to infinitesimally small sizes. However, we realize that constraint does not stop numerical errors with sufficient squishing effort. Using a stopping criteria on recursion in *moveSingleSplitLine* to limit the smallest cell width to some precision may work in theory, but it has yet to be tested.

Property 3: Each motion step positions half of the remaining split lines

This property is more of a statement of efficiency than of correctness: we move a minimal number of split lines with simple calculations in each step of our motion algorithm. We achieve logarithmic performance because the absolute, rendered distances between all split lines on either side of a cental split line change with respect to the movement of that split line. Since moving the target split line conceptually resizes cells uniformly on either side, either by squishing or shrinking, the central split line is able to move to its final position, and in doing so, half of the cells are resized with each step in *moveSingleSplitLine* in Figure 4.9. Referring to Figure 4.10, we see that the region $[x_C,\ x_R]$, the half of $[x_L,\ x_R]$ without target x_S, can all be resized and ignored once x_C has been moved. Our calculation of $C.f$ in *moveSingleSplitLine* determines the final position of x_C with a simple rescaling with respect to initial and final positions of x_S in $[x_L,\ x_R]$.

Property 4: Cells may stretch before they shrink

Before the algorithm starts an iteration, we can look at the current state of the motion to see how some regions can be moved several times and still approach their intended final distortion from the original. Consider the state of the split lines in

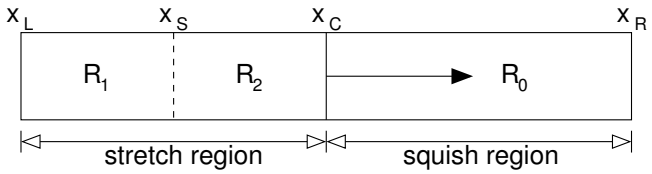

Figure 4.11: A user has moved split line x_S to the right, and the current state of running *moveSingleSplitLine* from Figure 4.9 has calculated that x_C should move to the right. Regions R_0, R_1, and R_2 are labeled sections of interest for this movement. R_0 is the region $[x_C, x_R]$ that is deformed, in this case squished, but not recursed through. R_1 is stretched when x_C moves toward x_R, but should be stretched more before the algorithm finishes. R_2 is also stretched when x_C moves toward x_R, but should be squished before algorithm finishes.

Figure 4.11, where x_S is the target split line we wish to move towards x_R, which is the right boundary for x_C, the current center split line.

The motion algorithm, *moveSingleSplitLine*, determines that x_C should move toward x_R. We know that of the regions R_0, R_1, and R_2, the movement of x_S should cause R_1 to grow and the other two regions to shrink uniformly. Our algorithm then shrinks all split lines in R_0 since x_S is left of x_C, and at this point, all split lines in $[x_C, x_R]$ are resized and in their correct final absolute positions. However, that rightward move of x_C stretches R_2 and does not stretch R_1 enough because x_S is not moved to its final correct place in $[x_L, x_C]$. Can we be certain that subsequent iterations are sufficient to correct this "mistake" and resize R_1 and R_2 properly?

We can look at R_1 and R_2 concurrently and see that although cells in both regions have been resized insufficiently, we are indeed approaching the intended outcome. The location of split lines in R_1 and R_2 are still in the region that has yet to be finalized, so we need to examine the effects of several iterations through *moveSingleSplitLine*, down the hierarchy. If running the algorithm is only able to resize regions uniformly between x_S and the immediately following iteration bound-

aries, then our algorithm performs properly with subsequent uniform movements using the center split lines.

We know R_0, the half of the hierarchy that does not get descended, is always correctly rescaled to its final size, through a uniform rescaling. Similarly, the side that our algorithm descends is uniformly scaled when x_C moves. Because previous resizings of R_1 and R_2 have been resized identically since they are in the same domain descended by our algorithm, the final split line movement of target x_S will ultimately correct the insufficient movements noticed in the intermediate stages of the recursion. This means that although the intermediate movements of R_1 and R_2 do not seem correct, the combined region is scaled uniformly and subsequently descended, which guarantees the next iterations will properly scale the areas on both sides of its center split line.

4.3.2 Moving several split lines

Unlike TJ1, TJ2 uses an algorithm capable of moving several split lines simultaneously. Motion in TJ1 consists of an algorithm that moves each split line in turn by performing operations similar to the TJ2 *moveSingleSplitLine* algorithm. The TJ1 algorithm starts at a split line, moves it a small fraction in its movement domain, then ascends to its parent to move it, and so on. This algorithm does not scale well: it moves split lines high in the quadtree hierarchy several times, once for every split line descendant being moved. These repeated adjustments high in the hierarchy lead to numerical instability.

For k moving split lines in n total split lines, although TJ1 only moves $O(k\ log(n))$ split lines, it moves the root of the hierarchy k times. In TJ2, we have developed an algorithm capable of moving with the same time complexity, but only moving the root and any other split line at most once, producing a numerically stable motion solution capable of moving more split lines accurately. Instead of ascending the hierarchy, TJ2 descends, moving each split line as it progresses, much

like the single split line movement *moveSingleSplitLine* from Figure 4.9.

Both TJ1 and TJ2 must compute the initial and final positions of the split lines being moved; these split lines are not the only split lines being moved but are the split lines specified by the resizing action. Assuming that we have a subset N of split lines, with initial positions $N.i$, that move in the set A of all split lines, an application-specific resizing function determines which regions between split lines in N either grow or shrink. AD functions provide assistance to the growing process by determining the new sizes for a set of regions, given a specified region growth rate; the shrinking function also uses growing functions, after inverting the set of regions. After computing the new sizes for each region in N, we determine the final locations of all split lines, $N.f$, by placing the regions in order starting from the $minBoundary$ until the last region is placed at the $maxBoundary$.

The reconstruction process for calculating $N.f$ also ensures minimum region sizes, for guaranteed visibility, are adhered to by not shrinking regions smaller than the minimum context size. For operations that wish to shrink regions smaller than the minimum context size, either the growing process does not proceed, or a limited amount of growing that does not violate the minimum size is allowed. The *moveSplitLineSet* algorithm in Figure 4.12 starts after calculating $N.f$.

Three interesting cases in *moveSplitLineSet* are shown in Figure 4.13. The termination case for recursion is shown as the left figure; there are no movable split lines from N in region $[A[start], A[end]]$, so recursion stops: the region in question has already been resized. The center figure shows the case where there are split lines in $[A[start], A[end]]$ and the center line C is in the set of movable split lines N. Similar to moving a single split line, we know that C is some split line in N and therefore, the final position $C.f$ has already been computed during the region preprocessing and is stored as $n.f$ for some $n \in N$. However, unlike the single split line case, which would terminate after this case, recursion continues on *both* sides of C. Finally, in the right image when $C \notin N$, we find $L \in N$ and $R \in N$, the two

moveSplitLineSet function
input: $N \subseteq A$ = list of split lines to move
where A is the set of all split lines
$A.i$ initial positions, $N.f$ final positions are known
$start$ = index into A, initially 0
end = index into A, initially $A.size$
output: A moved to final positions $A.f$

$C \leftarrow A[\lceil (start + end)/2 \rceil]$
if $N.nodesIn(start + 1, end - 1) = \emptyset$
return
else if $C \notin N$
$(L, R) \leftarrow N.neighbors(C)$
$C.f \leftarrow \left(\frac{C.i - L.i}{R.i - L.i}\right) \times (R.f - L.f) + L.f$
end if
$C.moveFromTo(C.i, C.f)$
$moveSplitLineSet(N, start, C.index)$
$moveSplitLineSet(N, C.index, end)$

Figure 4.12: *moveSplitLineSet* function that descends the split line hierarchy and recursively moves the set of all split lines A to their final absolute positions. *start* and *end* are two indices into A that allow descent into the binary hierarchy tree coded into A; the two split lines at these indices are initially the *minBoundary* (0) and *maxBoundary* ($A.size$) split lines, and are immovable for the current iteration of this function. If there are no split lines in N, the set of target split lines, that are between *start* and *end*, the recursion terminates. Otherwise, if the center split line C is not in N, the final position $C.f$ for C must be calculated, similar to the calculation in *moveSingleSplitLine*. L and R are neighbors of C in N, the closest nodes to C on both sides, in the range $[A[start], A[end]]$. If either L or R is not in that range, we use $L = A[start]$ or $R = A[end]$ as appropriate. Finally, after C has been moved, we recurse both directions in the split line hierarchy. Note that *moveSingleSplitLine* is a special case of this algorithm where $N = \{S\}$, where S is the single target split line.

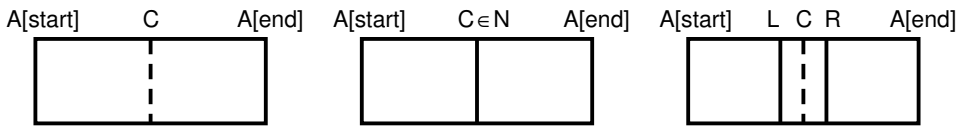

Figure 4.13: The three cases of function *moveSplitLineSet* are shown. **Left**: if N has no split lines to move between $[A[start], A[end]]$, the algorithm terminates since nodes in $[A[start], A[end]]$ have been resized. **Middle**: if $C \in N$, the algorithm moves C from $C.i$ to $C.f$ and recurses on the left and right sides of C. **Right**: if $C \notin N$, the algorithm finds the nodes $L, R \in N$ that are closest to C to compute $C.f$. After computing, the algorithm moves $C.i$ to $C.f$ and recurses.

closest split lines on the left and right sides of C, to calculate $C.f$, then continue the recursion on both sides of C. These movements resize regions to the left and right of the center split line uniformly, and arguments made for repeated recursions in our single split line movements also apply to moving a set of split lines.

Chapter 5

Evaluation and Discussion

In this chapter, I compare the relative performance of TJ2, with respect to the performance of similar TJ1 actions, discuss future work directions for AD applications, and conclude my thesis.

All performance tests were done on a machine with a 3.0 GHz Pentium IV processor, *Java* 1.4.2_04-*b*05 HotSpot runtime environment with a maximum of 1.8 gigabytes allocated for of heap, and nVidia Quadro FX 3000 video chipset, running twm in XFree86 version 4.3.99.902 with no additional processes running. The canvas resolution for TJ1 and TJ2 was set to 640 pixels wide by 480 pixels high and timing and node counts were output by TreeJuxtaposer, averaged from several manually prompted redrawings of each tested dataset. Since datasets that could be loaded by either version of TreeJuxtaposer are required for comparison, I chose to compare with two classes of trees and the largest contest trees from the InfoVis 2003 Contest dataset [28], described in Appendix A. I also compare larger datasets with TJ2 to get a better idea of non-synthetic dataset performance by using the directory tree structure from the Open Directory project [26], a large online browsable catalog of several billion websites. The directory categorization tree structures I use are from March and June 2004, with shortened names of 03/04 and 06/04, and are approximately 500 million nodes each with many structural differences.

The version of TJ1 that I use is from before the TJ1-contest, and I found

I was able to compare the two largest contest datasets, of $\mathsf{animalia}_\mathsf{a}$ and $\mathsf{animalia}_\mathsf{b}$. This pair of datasets do not load with TJ1-contest, and while I was evaluating TJ1 with this data, the memory consumption while doing this was near the maximum heap size, but no garbage collection occurred to skew the timing results. Also, several trees would load but not render in TJ1 due to a programming error. I found that by removing a single node, this error was eliminated.

The simple synthetic tree classes I chose to represent are the canonical balanced binary trees and star trees. Balanced binary trees were chosen since they have been used almost exclusively in TJC published results on rendering large trees in accordion drawing grids [5]; they are a good example of a well-known predictable structure. Star trees, simply one root node attached to many leaf nodes, were chosen based on my observations made of rendering traversal algorithms used in TJ1 and TJC; both render binary trees well, but may have problems with higher degree internal nodes that would be exacerbated by a node with an extreme number of children. A more in-depth tree classification system with many real tree datasets would be more complete, but I believe that the tree classes I test are simple to classify, show interesting progressive trends, and reveal a measure of efficacy of TJ2 versus TJ1. Furthermore, I show how the synthetic datasets lead to interesting curves through the space of all possible tree permutations, while real data validates my choice of synthetic datasets.

Each of the following comparisons between TJ1 and TJ2 are investigated: preprocessing, including layout time and the initial difference calculations, in Section 5.1; number of nodes rendered and how long to render a scene, in Section 5.2; memory consumption, in Section 5.3; and marking efficiency, in Section 5.4. When available, I also compare TJ2 with published results from TJC, investigating potential advantages of either method when rendering large trees of unknown topology. A summary of the results is given in Section 5.5.

5.1 Preprocessing

The time to load a single tree is the time needed to parse a dataset, construct the necessary split line grid, perform the gridding described in Section 3.1.1, and calculate the BCN values to perform associated difference marking. Loading several trees for comparison always involves the computation of differences between each pair of trees, which of course adds marking time for each of the differences on rendering as well; marking is covered in Section 5.4. Also, since parsing the dataset for a pair of trees is exactly the same as loading each tree sequentially, I will only investigate the parsing process for single trees, keeping each investigation as simple as possible; the more interesting preprocessing for multiple trees occurs after parsing.

Parsing

The dataset parsing time, shown in Figure 5.1, for TJ1 and TJ2 seems to be quite different, although they both use identical parsing libraries. An explanation for the differences in parsing time could be due to small changes in object constructors used in parsing; the change is simply the initialization of the BCN score of each node to 0 in TJ2. However, these differences are only a small constant factor to the linear time complexity for parsing and is not as much a factor as the dataset size increases. It is likely that small changes, magnified over millions of instances, would add as much to the parsing function in TJ2 as they would add to later preprocessing functions in TJ1.

TJ1 and TJ2 parsing functions are both linear with respect to the number of nodes read from the dataset file; there is no appreciable difference between parsing binary and star dataset classes in either TJ1 or TJ2. Published results from TJC [5], which uses a parser built from standard parsing libraries, indicate that parsing a two million node binary tree takes ten seconds, about one quarter the time of TJ2; parsing is still linear in TJC, albeit with a smaller constant factor than either TJ1 or TJ2.

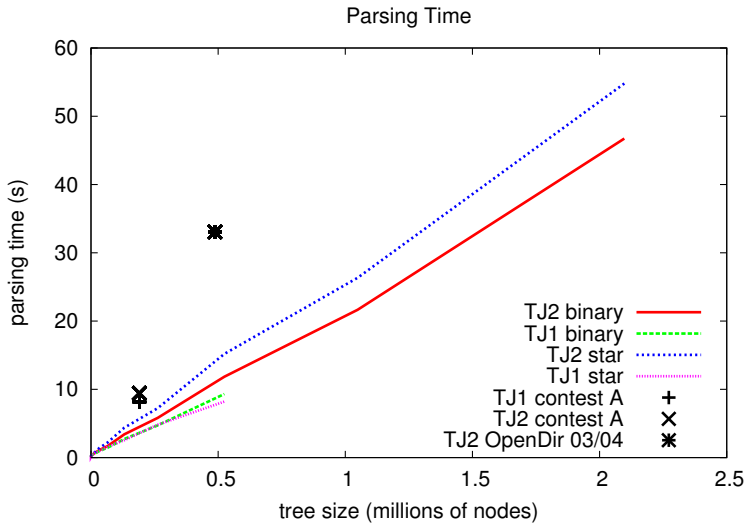

Figure 5.1: The parsing times for TJ1 and TJ2 with binary and star synthetic trees, and single contest and Open Directory real trees. All parsings are done with the same library, but TJ2 is slightly slower with a small change in its tree node constructor. The Open Directory tree takes much longer to load when compared to synthetic trees of the same size due to its structural complexity and non-synthetic node names.

Gridding

Preprocessing time, which is primarily gridding in TJ2 and the quadtree layout methods of TJ1, is substantially different between TJ1 and TJ2, even after accounting for the aforementioned differences in parsing. As shown in Figure 5.2, both TJ1 and TJ2 preprocessing are linear in time with respect to the number of nodes placed, but TJ2 is at least ten times faster than TJ1, due to construction of the quadtree structures in TJ1. The simplicity of TJ2, which has a low gridding constant and simple partitioning scheme as mentioned in Section 3.1.1, allows balanced binary trees with two million nodes to be loaded in under 15 seconds. Most of the extra time spent by TJ2 with the star trees, compared to binary trees, is maintaining the list of leaves, which is two times larger for star trees than for binary trees.

Finally, notice that the real-world contest and Open Directory datasets shown

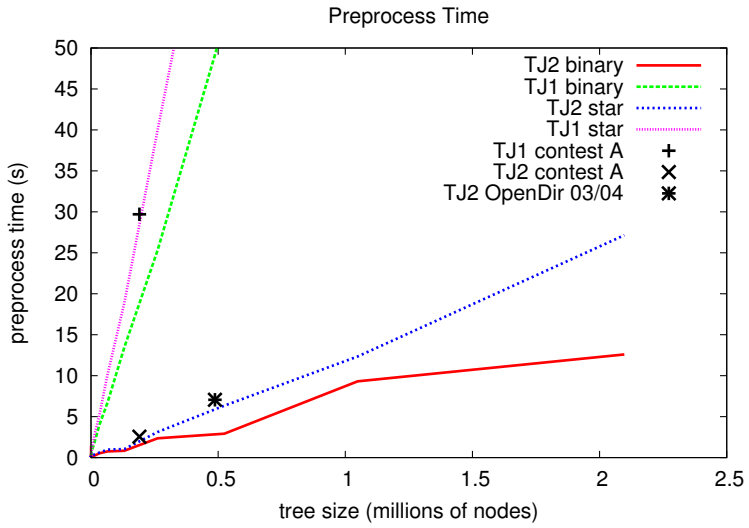

Figure 5.2: Preprocessing time for TJ1 and TJ2, which includes gridding and other initialization tasks. TJ2 is ten times faster than TJ1 for the largest trees loadable by TJ1. The contest and Open Directory datasets process in time similar to a star tree of the same size due to high branching factor, limited depth, and other node processing factors.

in Figure 5.2 for both TJ1 and TJ2 appear close to the respective synthetic star tree class dataset of similar size. A small amount of extra time is spent on sorting of real node names, but these datasets are particularly dense mostly at the leaf level, when considering preprocessing time. The ratio of leaves to internal nodes for the contest dataset is 154922 : 190265, or 81% leaves; there are many internal nodes that have many children, but it is difficult to determine a precise impact of the internal 19% of the tree. It is too difficult to determine how relatively efficient TJC would preprocess star trees from the published results [5]. The closeness of the contest dataset to the star tree of the same size does indicate that the family of star trees used in my evaluation is not entirely irrelevant.

Computing Differences

Adding another tree affects the preprocessing by including the difference computations after each subsequent tree is added. The time to process differences in TJ1 for the contest dataset is approximately 50 seconds, while TJ2 takes approximately 12.5 seconds. This time difference is not related to the difference computation, since the computation is unchanged between the applications, but the way marks are stored, which is investigated further in Section 5.4. The linked list of marked differences in TJ1 is implicitly sorted by the iteration process that adds each different node to the group of marked differences, but it contains a list of every marked node, and does not collect nodes with adjacent node key values into ranges. This means that for the first marking action of differences, and each subsequent change in the node marking, TJ1 must process the entire list of differences for each node range considered for drawing.

5.2 Scene rendering

There are two telling benchmarks involved in the complexity of rendering a scene: the time it takes to render a scene and the number of nodes rendered for a scene. Also, the time complexity per node is important to consider since there is no benefit to rendering fewer nodes if the time to process each node is substantially slower than alternative, more brute force, methods.

Scene rendering time

Assuming that a sufficient number of nodes are rendered to give an accurate representation of a dataset, a useful metric is the wall-clock rendering time for a dataset. Figure 5.3 shows the rendering time performance of TJ2 with the binary and star tree datasets, again with the contest dataset shown. As expected, the datasets are correlated to the number of nodes rendered, in Figure 5.4, but are not as smooth

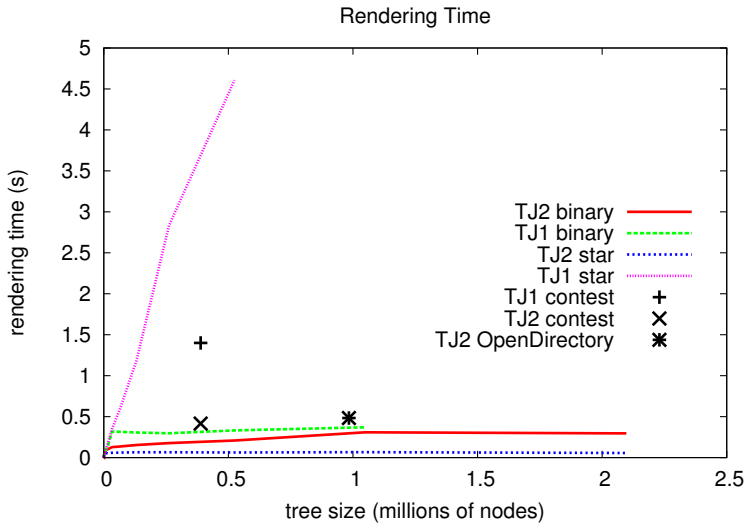

Figure 5.3: Rendering time for TJ2 is constant beyond datasets of a threshold for both binary and star trees. TJ1 renders binary trees slightly slower than TJ2, but star trees are much slower. Although TJ1 aimed for performance similar to TJ2, some classes of trees cause TJ1 to render slowly. The rendering time performances of TJ1 and TJ2 are closely related to the number of nodes rendered, as shown in Figure 5.4. Figure 5.5 shows another view of this relationship as the rendering time per node.

due to the timing accuracy. The correlations are similar for each of the datasets tested, so rather than analyze the rendering times for each pairing, I will compare the rendering speed per node rendered between each dataset.

Number of nodes drawn per scene

In Figure 5.4, we see that for TJ2, the number of nodes rendered for star trees is a constant after a certain number of nodes. This number represents the saturation of leaves under a subtree in TJ2, where once the maximum number of leaves per vertical height is reached, a subtree will not render any more; this is the result of our pixel-bounded rendering of leaves.

Similarly, but not as abrupt, is the series of binary trees. For each new layer

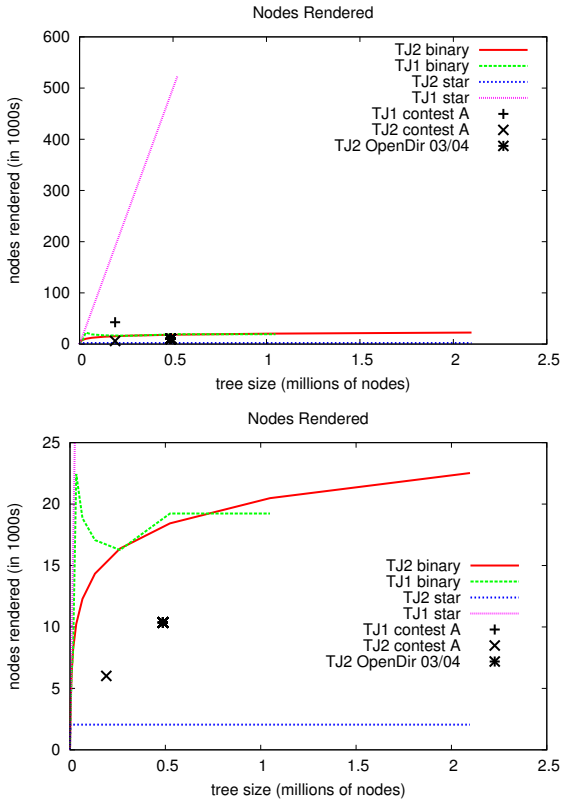

Figure 5.4: TJ1 is unable to cull nodes for my star tree synthetic datasets and draws every node, shown in the top graph; this performance is obviously not scalable. The bottom graph shows a more detailed view of TJ1 binary tree performance compared to TJ2 performance. In TJ2, star trees render a constant number of nodes and binary trees render an additional constant number of leaves for each doubling of nodes. The contest and Open Directory real-world datasets render between the star and binary tree examples since they have complicated internal node structures, but are not deep trees. TJ1 binary tree performance appears relatively close to TJ2 performance, but overculls dense regions since it does not properly render some datasets with its culling criteria.

of leaves added after a certain point, where a layer of leaves is the same size as the previous half-sized tree, there is a limit to the amount of rendering in that level, again dependant on the number of pixels. For trees larger than the first tree that maximizes the number of leaves rendered, the tree twice the size takes the same amount of time more to render. In the test case, where the screen is 480 pixels high, we render at most 2048 leaves and each progressively larger binary tree renders exactly 2048 more nodes than the previous. The limit in the binary tree case would be visible in the graph only if horizontal culling is used, but we do not cull in that direction.

For TJ2, the contest dataset renders fewer nodes than the binary tree of similar size simply because it is not as tall as that binary tree and has many more leaves. Of course the contest dataset cannot render as few nodes as the simple star tree of any size since the star tree has no interesting internal structure. Conversely for TJ1, also shown in Figure 5.4 the contest dataset renders fewer nodes than the star tree. The star tree in TJ1 is one of the worst case rendering examples since every node is rendered, and this leads to poor rendering for many trees with high branching factors. Binary trees are much more efficient than star trees for TJ1. Also shown in Figure 5.4, the binary trees that rendered properly with TJ1 show performance characteristics similar to TJ2. Unfortunately, TJ1 rendering quality for such large trees suffers from overculling effects and does not render some trees properly.

If we attempt to render a star tree with TJC, using its rendering algorithm as described in [5], it would have the same poor rendering count as TJ1, also rendering every node. This is reflected in the published TJC results for the contest dataset of $190,265$ nodes, where TJC renders $51,255$ nodes, compared to an $8,388,607$ node balanced binary tree, where TJC renders $50,356$ nodes. TJC rendering uses an algorithm that considers rendering a subtree as a single horizontal line if its extremal leaves subtend the same pixel, and does not partition the children of nodes

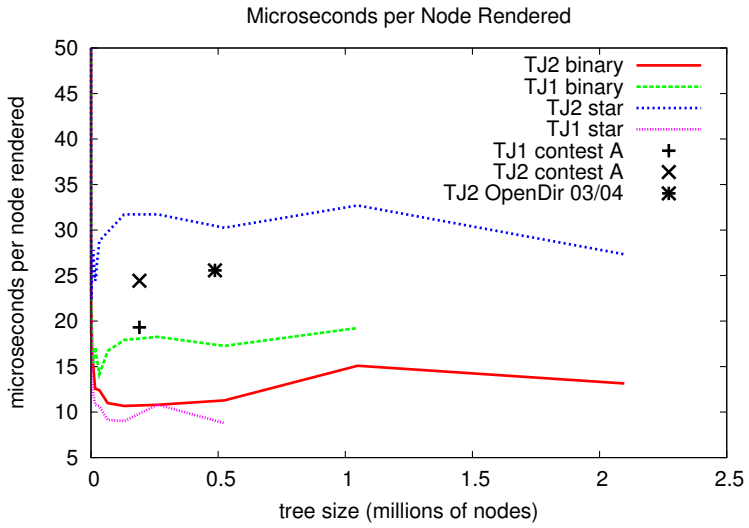

Figure 5.5: The relationship between the number of nodes rendered and the amount of processing required per node is important in understanding the tradeoffs of rendering fewer nodes for complex structures. This figure shows TJ2 rendering performance for star trees is three times slower than all other performance ratios, but the dataset renders scenes faster than TJ1 since TJ2 aggressively culls this dataset and TJ1 draws every node. Similarly for binary trees and the real-world contest datasets, TJ1 renders nodes faster on average, but TJ2 renders many times fewer nodes per scene, as shown in Figure 5.4. Note that for real-world datasets with very different sizes and structures, TJ2 renders nodes from the contest and Open Directory datasets with similar efficiency.

with high branching factors. This means that any node with leaves that subtend more than the pixel-based culling criteria of TJC will cause TJC to draw at least a path from each child node to a leaf. Therefore, although TJC has been shown to scale well with large balanced binary trees, algorithmic improvements that consider trees with higher branching factors would probably be necessary to scale its rendering performance with larger n-ary trees.

Average time to render a node

Figure 5.5 shows the per-node rendering performance of TJ1 and TJ2. As shown, the average time required to draw a node in a star tree for TJ2 is three times greater than for nodes in the same dataset for TJ1. But, even though TJ1 renders nodes each at a speed of about 10 microseconds per node and TJ2 needs 30 microseconds per node, TJ2 renders a much smaller number of nodes for that dataset class. In fact, TJ1 renders every node of the star tree datasets and TJ2 renders a constant number of nodes after a large enough dataset size, as shown in Figure 5.4. Interestingly, TJ2 draws binary tree nodes on average faster than TJ1, the opposite result of the star tree case, which means that although they render similar numbers of nodes, TJ2 renders binary trees faster than TJ1. Average node drawing performance for the contest dataset is close, with the TJ1 rendering cost approximately 25% lower than TJ2; the Open Directory dataset has similar per-node rendering performance. However, since TJ1 renders seven times the number of nodes that TJ2 renders for that dataset, the time is 0.3 seconds for TJ2 versus 1.4 seconds for TJ1, as shown in Figure 5.3.

5.3 Memory usage

The two classes of results to consider for memory usage include single trees and tree comparisons. I attempt to remove the minimal memory required to store names of nodes from each dataset by first subtracting the size of each dataset file from the raw memory results. This is simply to investigate the structural memory usage for TJ1 and TJ2, and should not influence the overall results, but does influence the raw memory usage ratios between each version of TreeJuxtaposer.

Using a simple grid structure to position dataset topology in place of the quadtree hierarchy has improved the memory performance, and therefore maximum sizes of datasets. Shown in Figure 5.6, memory usage in TJ2 is five times more

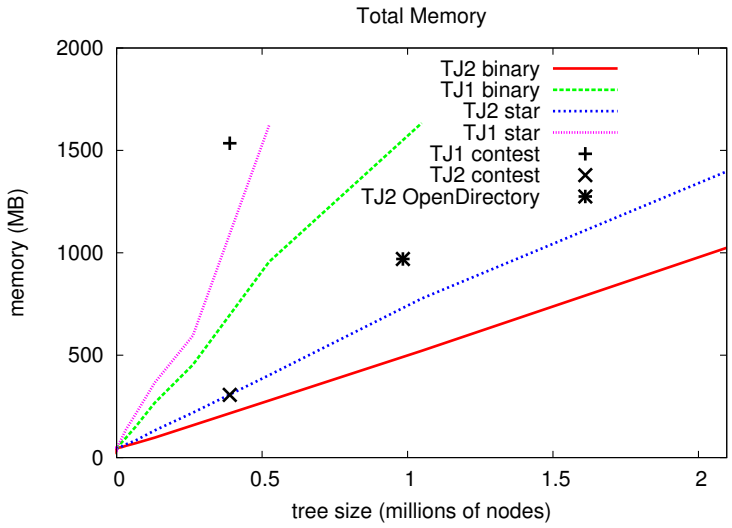

Figure 5.6: Memory performance for binary and star tree with TJ1 and TJ2. Memory usage in TJ2 is five times more efficient than TJ1 for identical datasets. Also shown are the contest dataset comparisons, which suggests that structural difference storage in TJ1 is poor compared to TJ2. The Open Directory dataset comparison uses more memory to store many structural differences and fully qualified node names, which are used to differentiate identical leaf names and provide a more accurate pair-wise node correspondence between the trees.

efficient than TJ1 since it either allows trees five times larger to be loaded, or uses one fifth the memory of TJ1. For smaller datasets, this ratio is slightly smaller, but still considerable.

Since the marked node storage has been improved in TJ2, much larger tree comparisons with full difference marking have been possible. Also shown in Figure 5.6 are the largest tree datasets from the InfoVis 2003 Contest, compared using TJ1 and TJ2. Clearly, the amount of memory to store the differences between the datasets impacts the TJ1 contest by more than a factor of five, since the comparison in TJ2 uses nearly the same amount of memory as the star tree dataset in TJ2, while the comparison in TJ1 is comparable to a star tree dataset with 50% more nodes using TJ1.

Application	TJ1	TJ2
First Scene Unmarked	115	0.3
Subsequent Scenes Unmarked	1.5	0.3
First Scene Marked	130	2.5
Subsequent Scenes Marked	1.5	0.5

Figure 5.7: This table shows the marking performance of TJ1 compared to TJ2, in seconds. When comparing two datasets of over $190,000$ nodes each, the first row shows the time to render the first scene, which includes the set of automatically marked differences. The second row shows time to render subsequent scenes; note that TJ1 caches results from the first scene, dramatically reducing its color lookup for each node while TJ2 performs the same efficient color lookup in both cases. The third and fourth rows show results of marking an entire dataset in both applications, which marks BCNs of marked nodes on the indirectly marked tree. The performance of TJ1 again is poor on the first scene, but aided by caching in subsequent scenes. TJ2 is slightly slower than an unmarked scene, and requires some time to compute the set of indirect marks immediately after the tree is marked.

5.4 Marking efficiency

Finally, I will show the tradeoffs in marking efficiency for TJ1 and TJ2. After marking an entire single tree in TJ1, each node caches the marking color for subsequent scene renderings, so marking takes time linear in the number of nodes in the tree once, then time similar to unmarked rendering for each additional scene. Similar marking in TJ2 traverses node structures in the indirectly marked tree, as described in Section 3.3, which incurs a worst-case marking penalty in TJ2 that is $O(n)$, where n is the size of the topological tree. Since marking single trees in TJ1 and TJ2 are very similar to their individual scene rendering performances, it is more interesting to consider marking an entire tree while two topologically different trees, such as the large contest trees, are loaded.

After marking one contest tree in TJ1, it takes approximately 130 seconds to render the first scene afterwards, which is approximately 15 seconds longer than the first scene rendered. The first scene takes 115 seconds simply because of the difference marks on both trees must be cached for every node. More time is required

after marking one tree because that single user-defined mark adds one more range that must be considered during the color caching process. Normally, these two trees render in under 1.5 seconds, and this time does not change considerably after marking.

After similar marking in TJ2, a delay of less than two seconds is required to traverse the directly marked subtree and compute the list of marked nodes in the indirectly marked tree, as described in Section 3.3. For TJ2 performance in scenes before marking, the average rendering time for both trees is 0.3 seconds, while after marking takes 0.5 seconds, because TJ2 does not cache the marking results per node and must lookup the color for every node rendered. The tradeoff of nearly doubling the TJ2 rendering time is mostly related to overdrawing marks in regions. Although marked region drawing is handled more efficiently in TJ2 than in TJ1, it would be an interesting area of future work to improve this drawing algorithm, perhaps by culling in areas of many marked regions.

5.5 Evaluation summary

TJ2, built on our new generic AD infrastructure, yields far better performance than TJ1 in every category that I measured: preprocessing time, rendering time, marking time, node drawing count, and memory consumption. Preprocessing times for TJ2 and TJ1 are both linear in the number of nodes, but TJ2 is typically ten times faster after the datasets have been loaded. TJ2 limits overdrawing and is able to render a constant number of nodes for my synthetic star tree examples, while rendering a fewer number of nodes than TJ1 for binary trees. For trees with complex structure that require TJ2 to spend more processing time per node than TJ1, TJ2 renders a much smaller number of nodes, and renders the entire scene up to five times faster than TJ1. The partitioned grid used by TJ2 for layout is also five times more efficient than TJ1, allowing much larger datasets to be loaded with less memory resources. Finally, my example of worst-case marking of the contest dataset comparison for

TJ2 is an order of magnitude faster than TJ1 for the first scene and three times faster for subsequent scenes. Moreover, TJ2 achieves its marking time performance without using up memory to cache marking results, allowing larger datasets to be loaded.

Chapter 6

Future Work and Conclusions

6.1 Future work

Mentioned throughout my thesis are several areas of interest that are either simple additions or more powerful features that would require modifications to the accordion drawing infrastructure.

Although we are interested in several different directions with our generic infrastructure, we will probably focus on several high-demand areas of TJ2, which include: editing a tree structure, saving changes to a tree, saving the state of a tree, replaying a transaction log recorded while navigation a tree, undoing navigations, and storing a meaningful representation of a transaction history in a human-readable format. Most of these modifications will make use of a sophisticated logging structure. The addition of these features would make TJ2 a much more powerful system, and much more appealing to biologists who need more than just the visualization system of the current TJ2.

Another area of future work involves adding more attribute capabilities to TJ2 for operations such as: data filtering, marking common features, and creating a true interface to TJ2 that another application can access. By creating an API for TJ2, we may use a second application to drive the performance of navigation, node selection, or editing. TJ2 may act as a navigation component to an application

that interfaces to a database of animal characteristics, for example when selecting animals that have wings, the animals with wings will be automatically highlighted when that application sends that set of interest to TJ2.

Progressive rendering offers two interesting areas of future work. First, we would like to minimize, or ideally eliminate, the overhead that our infrastructure incurs when progressive rendering is turned off, especially when progressive rendering is not necessary and the dataset can be rendered in a single frame. Second, we would like an automated way to decide whether progressive rendering should be on or off, rather than require manual intervention from the user.

Finally, we envision the juxtaposition of a phylogenetic tree with the sequence data used to build it by combining TJ2 with SequenceJuxtaposer (SJ) [35]. Since both TJ2 and SJ use the same AD infrastructure, it would be possible to have these applications share a set of split lines and for navigation to distort both grids concurrently. We would also like to investigate adding editing capabilities and more sophisticated navigation support where collapsing a subtree leads to the display of an aggregate sequence for the entire subtree.

6.2 Conclusions

I have presented our accordion drawing infrastructure, which provides rubber-sheet navigation and guaranteed visibility for information visualization applications that are capable of laying their dataset objects on a grid. Our rubber-sheet navigation is numerically stable and provides a scalable, malleable surface for exploration of large, complex datasets. Accordion drawing also provides an interface to generic partitioning, seeding, and rendering methods used to render datasets in time $O(p)$, where p is the number of pixels on-screen.

Furthermore, my implementation of TreeJuxtaposer on our AD infrastructure, TJ2, renders and navigates dense trees correctly with more time efficient rendering and layout techniques than its predecessor, TJ1. With compact represen-

tations of marked nodes, progressive rendering a skeleton of marks instead of an entire subtree, ascent rendering to guarantee a limit on the number of nodes rendered, five times more efficient memory performance, accurate picking, and limiting the number of nodes rendered for complex trees, TJ2 improvements give users a more responsive tree rendering with all of the advantages of TJ1.

Finally, I describe the improvements made for the InfoVis 2003 Contest on tree comparisons, where TJ1-contest, the improved TJ1 with incremental node searching and a helpful user interface, won first place overall. I present an in-depth analysis of how TJ1-contest supports many, but not all, of the common functions users of tree visualizations require. Our results show that TreeJuxtaposer is a capable, mature system that supports users in understanding the complex structures that exist in real-world datasets.

Glossary

Accordion Drawing (AD): an information visualization navigation paradigm that supports the stretching metaphor of manipulating data drawn on a malleable surface., p.2

Focus+Context: a technique in information visualization systems used to display areas of interest at focal points. The rest of the dataset, the context, is still displayed in less detail. The context provides additional structural semantics for the focus regions. Global Focus+Context systems, such as AD, show the entire dataset at all times., p.8

TJ1-contest: an improved implementation of **TJ1** with additional user interface tools, incremental node search capabilities, and other user tools to change the appearance of the tree visualization., p.112

TJ2: a redesigned implementation of the capabilities of **TJ1**, with improvements of **TJ1-contest**, which also uses our new **AD** infrastructure. Several improvements in rendering, marking, and correctness are described in detail in Chapter 3., p.4

TreeJuxtaposer (TJ1): an information visualization application used to navigate and compare several rectilinear trees, often phylogenetic trees, as shown in Figure 1.2. TJ1 is the original implementation of TreeJuxtaposer, as opposed to TJ1-contest, my InfoVis 2003 Contest submission version, and TJ2, my most recent TreeJuxtaposer implementation., p.2

ascent rendering: a rendering technique for trees that is topology-based and draws nodes along paths from leaves to the root. We can control the quantity of leaves and reduce the number of nodes drawn per scene for dense, complex tree topologies., p.38

ascent width: the width criteria of subtrees that we use as a stopping criteria for **ascent rendering**. Given as a value relative to block width, a larger ascent width means fewer ascents per leaf range, but we are limited to ascent plus **segment width** sums that are less than one-half block. With that restriction, when we find a subtree that is wider than the ascent width, we know that it cannot be drawn as a single horizontal line., p.40

base grid: the lowest quadtree level grid of TJ1, or the grid of split lines of TJ2 that are used to position topological tree nodes with the **gridding algorithm**., p.26

best corresponding node (BCN): when comparing two or more trees, nodes are paired up with the most appropriate matching node in every combination of pairs of trees. This relationship is not always bi-directional and some nodes do not have a BCN. A BCN value is calculated with the function $\frac{A \cap B}{A \cup B}$ for leaf sets A and B under two nodes; the BCN for node N, which is in tree T_1, in tree T_2 is the leaf set of the node with a maximal BCN value, in T_2, with the leaf set of N., p.52

block: the smallest size at which a geometric object is drawn, with the lower limit of a pixel. This is also known as the minimum feature size, which is equal to the line width of edges in TJ2. A block is always some integer multiple of pixels and is pixel-aligned., p.33

cell, (base grid): a region of the **base grid**, consisting of four lines on the base grid that form a rectangle, known as: top, bottom, left, and right. The grid cell

is used to position a topological tree node for rendering, culling, and picking in TreeJuxtaposer. Cells for a tree in TJ2 partition the entire base grid and do not overlap., p.26

directly marked: when comparing two or more trees, a node that a user has explicitly marked is called directly marked, while a node that is marked in another tree, as a consequence of node correspondences with directly marked nodes, is called **indirectly marked**., p.51

drawing (rendering stage): third stage of AD rendering associated with drawing the **seeded** marked ranges and split line ranges. The split line ranges are **partitioned** according to a previous stage., p.70

found nodes: nodes that match a searching criteria, such as substring matching in the `Found` panel, in the incremental search functionality of TreeJuxtaposer. These nodes are highlighted with the highlight color, which is modifiable through the `Group` panel., p.120

gridding (algorithm): the partitioning of a uniform grid used by TJ2 to assign a set of grid coordinates, which form a rectangle on the **base grid**, to topological tree nodes., p.28

guaranteed visibility: a property information visualization systems use to display important data at the expense of less important data; important data is always visible on the screen. TreeJuxtaposer and other AD applications use both **static** and **progressive** guaranteed visibility paradigms to give users navigational landmarks in their dataset visualizations., p.5

horizontal gaps: rendering gaps in **ascent rendering** that appear if we do not choose a subtree in a leaf range that horizontally covers all other subtrees., p.38

indirectly marked: a node that is not **directly marked** when comparing two or more trees with user defined marks., p.51

interaction box: a region defined by user interaction on an Accordion Drawing grid, which may be stretched or squished to reveal more details of datasets in regions of interest., p.60

marked data: data that is marked by a user. This data could be user defined with marking or computed using a user-specified function that performs the marking. For example, when comparing two trees in TJ2, the topological differences are marked data and the unmarked, similar nodes are **normal data**., p.5

marked ranges: regions of interest such as computed differences, search results, user marked groups, and even mouse-over highlighted nodes in TJ2. Ranges are used to compactly store subtrees and forests of subtrees for quick color referencing for nodes during rendering., p.46

node key: the enumeration value of a particular node in TJ2. We use keys to identify relationships between nodes by assigning keys in a pre-order, so the roots of subtrees are smaller than its descendants, and the entire subtree can be represented by a single range of integers., p.46

normal data: data that is not marked but is drawn to provide overall dataset structure and position of **marked data**., p.5

overlapping ranges: a pair of node ranges, often with both node ranges marked, in TJ2 that are either adjacent, non-unique, or partially overlapping. Two overlapping ranges can be combined into a single, unique range., p.50

partitioning: first stage of AD rendering associated with dividing a split line range into a set of **drawing** ranges. Partitioning precedes **seeding**., p.70

picking fuzz: a margin of error, which we set to five pixels, that allows us to pick nodes with the mouse without exact mouse positioning. If a desired node is in a region where it is not pickable, such as a very dense region, we expect a user to stretch its region with accordion drawing to disambiguate unwanted picking., p.55

progressive guaranteed visibility: a property of an information visualization system to use a drawing order that favors **marked data** over **normal data** when rendering animation frames. This provides landmarks during navigation for large visualizations that rely on **progressive rendering** approaches. Compare with **static guaranteed visibility**., p.6

progressive rendering: is an technique used in several graphics systems that allow for complex, or otherwise rendering intensive, scenes to be rendered in several stages. After each stage, the system allows for user interaction or continues to render the scene. Progressive rendering is necessary with AD in TreeJuxtaposer since partial rendering is fast enough but full scene rendering could take over a second. When an AD application renders, it displays marked nodes first to provide **progressive guaranteed visibility**., p.9

seeding algorithm: the process of enqueuing key tree nodes, or ranges of tree nodes, in a drawing priority-based, ordered list prior to rendering. Typically, the list contains enough information to render an entire scene, but rendering only part of the scene is also acceptable, especially in progressive rendering where rendering does not dequeue all nodes from the list., p.34

seeding: second stage of AD rendering associated with ordering a **partitioned** split line range and any marked ranges prior to **drawing**., p.70

segment width: the partitioning stopping criteria for leaf range seeding. Given as a value relative to **blocks**, the larger the segment width, the fewer leaf ranges

we must process during rendering. However, if the segment width is too large, we see gaps in dense regions since we only render one leaf per **segment**., p.37

segment: a component of a tree partitioning, at the leaf level, which is either the smallest rendering partition with more than one leaf, or any other rendering partition with exactly one leaf. We guarantee that only one leaf is drawn per segment., p.35

split lines: movable lines in an AD visualization application. These lines are stored in a balanced tree hierarchy, and affect regions in their domain by stretching and squishing their hierarchical children, to reveal areas of interest while squishing other regions together. The linear order of split lines cannot be changed and no part of the visualization is ever pushed out of view since split lines cannot be pushed beyond their domain boundaries. Split lines are essential in giving AD applications global **Focus+Context** properties., p.60

static guaranteed visibility: a property of an information visualization system to prioritize **marked data** over **normal data** in single images. This provides a sense of location for the marked data using the marks as visual landmarks. Compare with **progressive guaranteed visibility**., p.6

Bibliography

[1] Thomas Ball and Stephen Eick. Software visualization in the large. *Computer*, 29(4), April 1996.

[2] Lyn Bartram, Albert Ho, John Dill, and Frank Henigman. The continuous zoom: a constrained fisheye technique for viewing and navigating large information spaces. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 207–215. ACM Press, 1995.

[3] Luc Beaudoin, Marc-Antoine Parent, and Louis C. Vroomen. Cheops: A compact explorer for complex hierarchies. In *Proc. of IEEE Visualization '96*, pages 87–92, 1996.

[4] Benjamin B. Bederson and James D. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *Proceedings of UIST '94*, pages 17–26, 1994.

[5] Dale Beermann, Tamara Munzner, and Greg Humphreys. Scalable, robust visualization of very large trees. *EuroVis 2005, to appear*, 2005.

[6] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach. Image rendering by adaptive refinement. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 29–37. ACM Press, 1986.

[7] M. Sheelagh T. Carpendale, David J. Cowperthwaite, and F. David Fracchia. Three-dimensional pliable surfaces: For effective presentation of visual information. In *Proc. UIST*, pages 217–226, 1995.

[8] Savrina F. Carrizo. A colour-filling approach for visualising trait evolution with phylo genies. In Neville Churcher and Clare Churcher, editors, *Australasian Symposium on Information Visualisation (invis.au'04)*, volume 35 of *Conferences in Research and Practice in Information Technology*, pages 117–126, Christchurch, New Zealand, 2004. ACS.

[9] Coordinator & Editor David R. Maddison. Tree of life project. `http://tolweb.org/tree/phylogeny.html`.

[10] P.M. Fitts. The information capacity of the human motor system in controlling the amp litude of movement. *Journal of Experimental Psychology*, 47:381–391, 1954.

[11] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23. ACM Press, 1986.

[12] George W. Furnas and Benjamin B. Bederson. Space-scale diagrams: Understanding multiscale interfaces. In *Proc. SIGCHI '95*, 1995.

[13] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[14] David Hibbett, François Lutzoni, David McLaughlin, Joey Spatafora, and Rytas Vilgalys. Assembling the fungal tree of life. `http://ocid.nacse.org/research/aftol`.

[15] David Hibbett, RH Nilsson, M Snyder, M Fonseca, J Costanzo, and M Shonfeld. Automated phylogenetic taxonomy: An example in the homobasidiomycetes

(mushroom-forming fungi).
`http://mor.clarku.edu`.

[16] John P. Huelsenbeck and Fredrik Ronquist. MrBayes: Bayesian inference of phylogeny, 2001.

[17] Susanne Jul and George W. Furnas. Critical zones in desert fog: Aids to multiscale navigation. In *Proc. UIST '98*, pages 97–106, 1998.

[18] T. Alan Keahey and Edward L. Robertson. Nonlinear magnification fields. In *Proc. IEEE Symposium on Information Visualization*, pages 51–58, 1997.

[19] John Lamping, Ramana Rao, and Peter Pirolli. A Focus+Content technique based on hyperbolic geometry for viewing large hierarchies. In *Proc. SIGCHI*, pages 401–408, 1995.

[20] Wayne P. Maddison and David R. Maddison. *MacClade: Analysis of Phylogeny and Character Evolution. (User's manual).* Sinauer Associates, Sunderland, MA, 1992.

[21] W.P. Maddison and D.R. Maddison. Mesquite: A modular system for evolutionary analysis. version 1.0, 2002. Available from `http://mesquiteproject.org`.

[22] Tamara Munzner. H3: Laying out large directed graphs in 3D hyperbolic space. In *Proc. InfoVis 97*, pages 2–10, 1997.

[23] Tamara Munzner. Drawing large graphs with H3Viewer and Site Manager. In *Proc. Graph Drawing '98, Lecture Notes in Comp. Sci. 1547*, pages 384–393. Springer-Verlag, 1998.

[24] Tamara Munzner, François Guimbrètiere, Serdar Tasiran, Li Zhang, and Yunhong Zhou. TreeJuxtaposer: Scalable tree comparison using Focus+Context with guaranteed visibility. *SIGGRAPH*, pages 453–462, 2003.

[25] Tamara Munzner, Qiang Kong, Raymond T. Ng, Jordan Lee, Janek Klawe, Dragana Radulovic, and Carson K. Leung. Visual mining of power sets with large alphabets. submitted for publication, 2005.

[26] OpenDirectoryProject, 2005. `http://dmoz.org`.

[27] Ken Perlin and David Fox. Pad: an alternative approach to the computer interface. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 57–64. ACM Press, 1993.

[28] Catherine Plaisant and Jean-Daniel Fekete. Infovis 2003 contest, 2003. `http://www.cs.umd.edu/hcil/iv03contest/`.

[29] Catherine Plaisant, Jesse Grosjean, and Ben Bederson. SpaceTree: Design evolution of a node link tree browser. In *Proc. InfoVis 2002*, 2002.

[30] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3d interactive animation. *Communications of the ACM*, 36(4):57–71, 1993.

[31] George G. Robertson and Jock D. Mackinlay. The document lens. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 101–108. ACM Press, 1993.

[32] Ursula Rost and Erich Bornberg-Bauer. Treewiz: interactive exploration of huge trees. *Bioinformatics*, 18(1):109–114, 2002.

[33] M.J. Sanderson, A. Purvis, and C. Henze. Phylogenetic supertrees: Assembling the trees of life. *Trends in Ecology and Evolution*, 13:105–109, 1998.

[34] Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, and Steven P. Reiss. Stretching the rubber sheet: a metaphor for viewing large layouts on small screens. In *UIST '93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 81–91. ACM Press, 1993.

[35] James Slack, Kristian Hildebrand, Tamara Munzner, and Katherine St. John. SequenceJuxtaposer: Fluid navigation for large-scale sequence comparison in context. *German Conference on Bioinformatics*, 2004.

[36] James Slack, Tamara Munzner, and François Guimbrètiere. TreeJuxtaposer entry, InfoVis 2003 contest. `http://www.cs.ubc.ca/~tmm/papers/contest03`.

[37] Chris Stolte, Diane Tang, and Pat Hanrahan. Multiscale visualization using data cubes. In *Proc. InfoVis 2002*, 2002.

[38] David L. Swofford. *PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods). Version 4.* Sinauer Associates, Sunderland, Massachusetts, 2002.

[39] Li Zhang. On matching nodes between trees. Technical Report 2003-67, HP Labs, 2003.

Appendix A

TreeJuxtaposer Task Evaluation

The InfoVis 2003 Contest [28] was the inaugural IEEE Symposium on Information Visualization contest, and was composed of several tasks that TJ1 was well suited to solve, including its namesake: side-by-side comparison of trees. Example tasks from this contest included: detecting structural differences between trees, characterizing movements of tree structures, and searching for nodes with given attributes. The entire list of tasks proposed by this contest are found in the results section of this appendix, Section A.4.

Three of the major contributions made for TJ1, presented in this appendix, include: an analysis of the strengths and weaknesses of our TreeJuxtaposer and Accordion Drawing paradigm for the contest set of tasks; my addition of an incremental search capability for tree node labels; and my addition of an improved user interface, which greatly improves the usability of TreeJuxtaposer. We call the version of TJ1 that has search and user interface improvements **TJ1-contest** [36]; however, this version does not have the features discussed in Chapter 3. The results of the contest were very promising for the future of TreeJuxtaposer: our contest entry placed first overall and gave our work excellent exposure to the information visualization community.

In this appendix, Section A.1 describes the dataset provided for the contest, then Sections A.2 and A.3 describe the interface and search additions made to TJ1

for the contest, and finally the analysis for each task in the contest in Section A.4, as was presented in the contest entry. A summary follows in Section A.5.

A.1 Contest dataset

The contest dataset consisted of three different types of trees. The first dataset was a pair of small phylogenetic trees, the second dataset was a pair of large classification trees, and the third dataset was four file system trees.

Phylogenetic trees are constructed from sequence data and show possibilities in how each of the species represented in the leaves are related to each other; ideally the trees are binary trees but often are n-ary due to uncertainty in the construction methods or other biological phenomena. The phylogenetic trees supplied for the contest were a selection of bacteria classified with two different, unspecified methods.

Classification trees are created with a familiar Linnæan, hierarchical structure where subtrees represent groups of similarities in morphologies of species. The classification trees in the contest are of the kingdom *animalia* and are possibly from two different sources as they have several differences and inconsistencies. Each node has several attributes, which included Latin names and common, English names; I generated two classification trees from each classification dataset tree from Latin and common node names. Because not all nodes are supplied with common names, Latin names are used in the common trees for nodes that have a Latin name but no common name. When I use common names, such as mammal instead of mammalia, in the results, I am referring to common trees; in most cases Latin trees are used to provide solutions for the tasks.

File system trees represent the hierarchical structure of a simple file system. The four file system trees are snapshots from a university web-site over a three week period. Each internal node from this dataset corresponded to a file system directory and leaves corresponded to files in those directories.

The results from the contest in Section A.4 describe analyses of these three

datasets. When describing each dataset, $\mathsf{phylo}_\mathsf{A}$ and $\mathsf{phylo}_\mathsf{B}$ are the phylogeny trees, $\mathsf{animalia}_\mathsf{A}$ and $\mathsf{animalia}_\mathsf{A}$ are the classification trees, and logs_A, logs_B, logs_C and logs_D are the file system trees. However, since the classification and file system trees are too large to load simultaneously with interactive rates in TJ1-contest, subtrees are used in comparison tasks for those two datasets. The subtrees of classification trees are $\mathsf{mammalia}_\mathsf{A}$ and $\mathsf{mammalia}_\mathsf{B}$, rooted at class *mammalia* in the $\mathsf{animalia}$ trees, while the file system trees are hcil_A through hcil_D, the human-computer interaction laboratory web pages rooted at the *hcil* directory of the respective logs trees.

A.2 User interface

Figure A.1 shows the original user interface of TJ1 [24] with the slider, a node selection box, and buttons to add more trees, toggle differences and reset the tree navigation. All other functionality was accessed through a keyboard-based interface, which required users to remember keystrokes for most commands. Although some keyboard actions are essential in TJ1, having users remember too many keyboard commands is cumbersome. Some keystrokes involved advanced tasks that users not familiar with the system would not have understood and are only used for debugging; many mapped keys were not transformed into menu options.

Figure A.2 shows the contest user interface of TJ1-contest with a menu panel. The most interesting parts of the system included with this panel are `Find` and `Tools`. `Find` replaces the drop down box with the `Found` panel dialog, as seen in Figure A.6 and is described in Section A.3. `Tools` contains the two option panels `Groups` and `Settings`. The additions of the two `Tools` panels provides the original options, some new options, and state information that was previously hidden and made TJ1 hard to use.

In Figure A.3, `Groups` gives the user information about the currently resizing and marking groups in the top and bottom halves of the panel, respectively. The radio buttons between the color canvases and the labels collectively show the

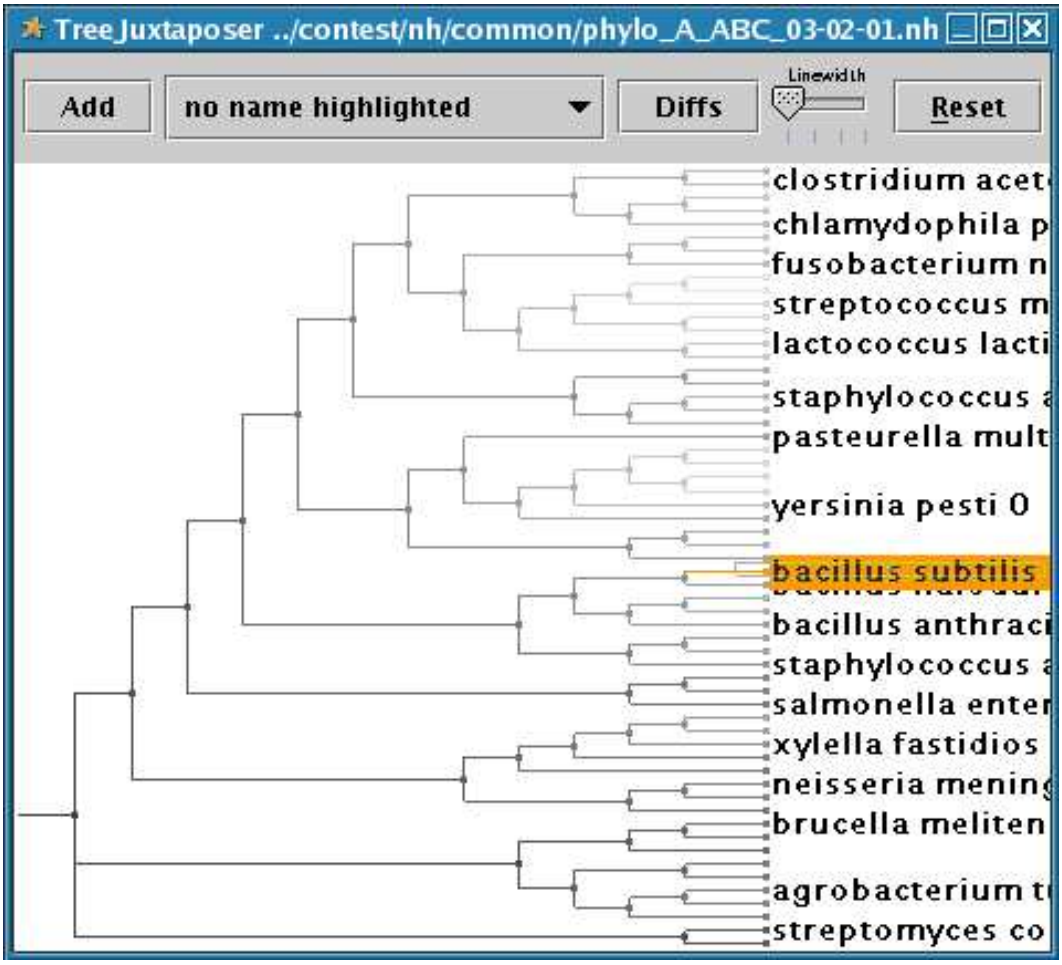

Figure A.1: TJ1, from [24], before modifications for the InfoVis 2003 Contest.

currently selected group for both actions; the canvases themselves can be clicked to change the marking color for that group. The buttons `Bigger` and `Smaller` allow the user to resize the resize group and `Reset` resets all tree views to their initial state. The radio buttons beside the options `Horizontal`, `Vertical`, and `Both` allow the user to choose how the resize group will act when growing bigger or smaller; `Both` resizes the group vertically as well as horizontally.

In Figure A.4 `Settings` offers some more options. The sliders on the panel `Line Width` and `Label Density` give control over the width of the edges in the

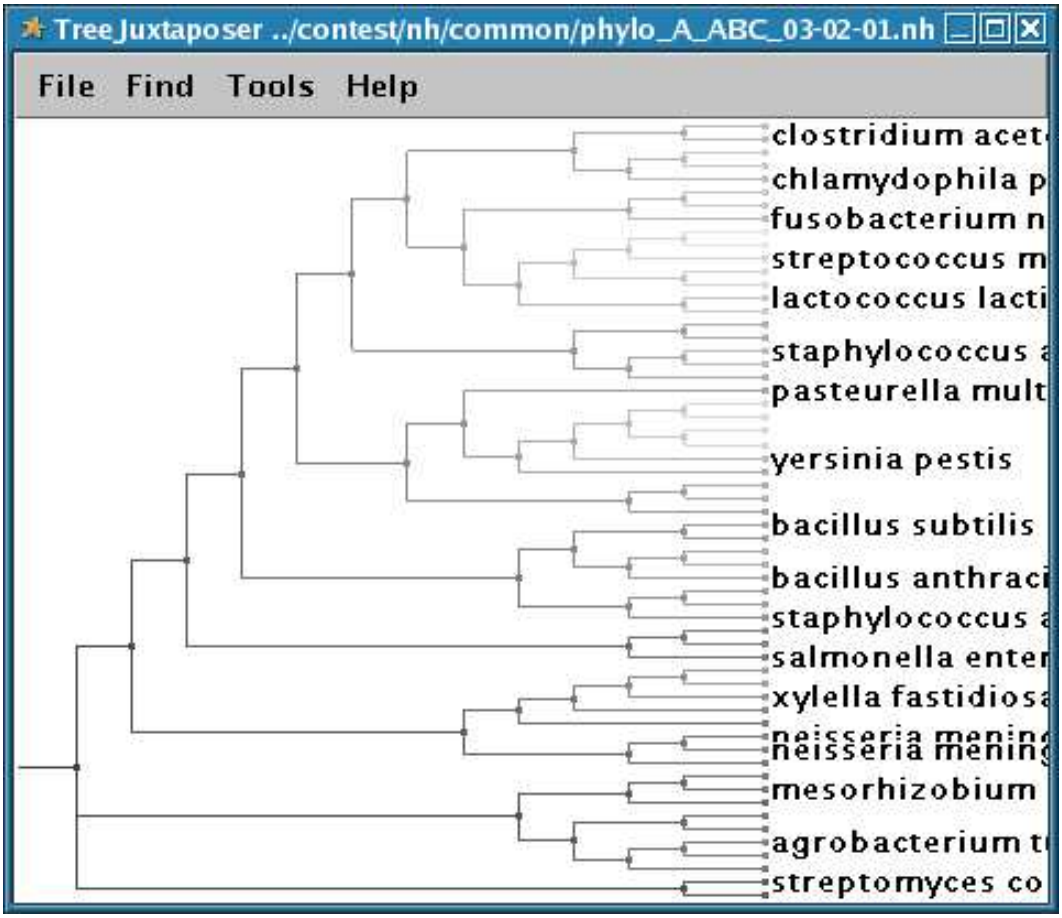

Figure A.2: TJ1-contest with title bar modifications for the InfoVis 2003 Contest.

tree, by default set to one pixel wide, and the density of the labels. At maximum density towards the left of `Label Density`, labels are squeezed together and adjusting the slider to the right increases the buffer space between labels, effectively decreasing the label density. Font sizes are also adjustable in this panel, and TJ1-contest uses the `Minimum` and `Maximum` values to draw labels as large as `LabelDensity` and other screen space factors allow. Other check boxes in the panel include: `Linked Navigation` for interactively resizing subtrees concurrently, `Show Differences` and `Show Labels` for selecting features to see and `Dimming` for marked and unmarked nodes.

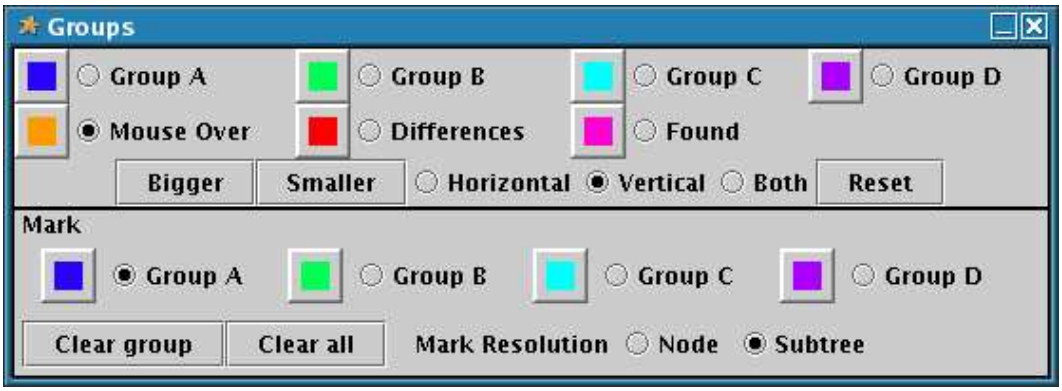

Figure A.3: The TJ1-contest `Groups` panel added for the InfoVis 2003 Contest.

The toggle buttons and check boxes in the `Groups` and `Settings` panels are also important for showing state of each of the properties they represent. Versions of TreeJuxtaposer prior to TJ1-contest, without indications of state, were quite difficult to use. State continues to be an issue in the development of new features and more recent versions of TreeJuxtaposer, including TJ2, use the `Debug` panel for displaying critical state information. Future improvements to the user interfaces are quite likely with the addition of features such as choosing tree orientations.

A.3 Incremental search

We determined that when analyzing the contest tasks that an improved searching tool was necessary for TJ1-contest. The searching tools are used to find a node with a particular label; a found node is marked with a highlight color and can be grown with a typed keyboard command: `b` for bigger and `s` for smaller.

TJ1 had only a drop-down selection box, as in Figure A.5, which was sorted alphabetically by node label. These early versions could only highlight a single node at a time. To find a node of interest, the entire list had to be scrolled through, a time consuming process; typing letters while the drop-down box was selected would also jump to the next node in the list starting with that letter but was still not

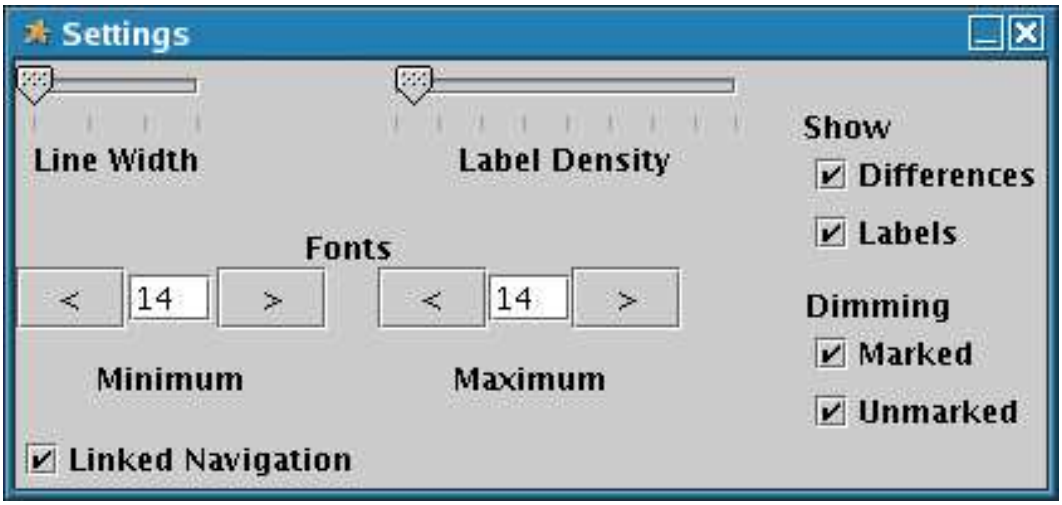

Figure A.4: The TJ1-contest `Settings` panel added for the InfoVis 2003 Contest.

efficient. Unfortunately, Java implementations of drop-down boxes do not scale well with several thousand nodes. With more than two thousand nodes in TJ1, the drop-down box is very slow and uses too much memory; the drop-down box itself is a major memory bottleneck in scalability.

The sorting and typing methods were also not at all useful for searching for known strings that did not occur at the start of the node names. There were also some problems with nodes that are not uniquely named. TJ1 makes the list of nodes in the drop-down box unique and although it attempts a renaming scheme, the scheme did not work properly: labels were also changed making navigation difficult whereas TJ1-contest distinguished between names, in searching, and labels, in displaying nodes.

We identified the need for an interface where a user could type in their query and have multiple nodes selected, preferably with performance that would lead to real-time analysis as users entered search strings. The searching approach in TJ1-contest is based on the kind of incremental search commonly seen in Emacs or Mozilla. When a user types in a search query in these dynamic searching sys-

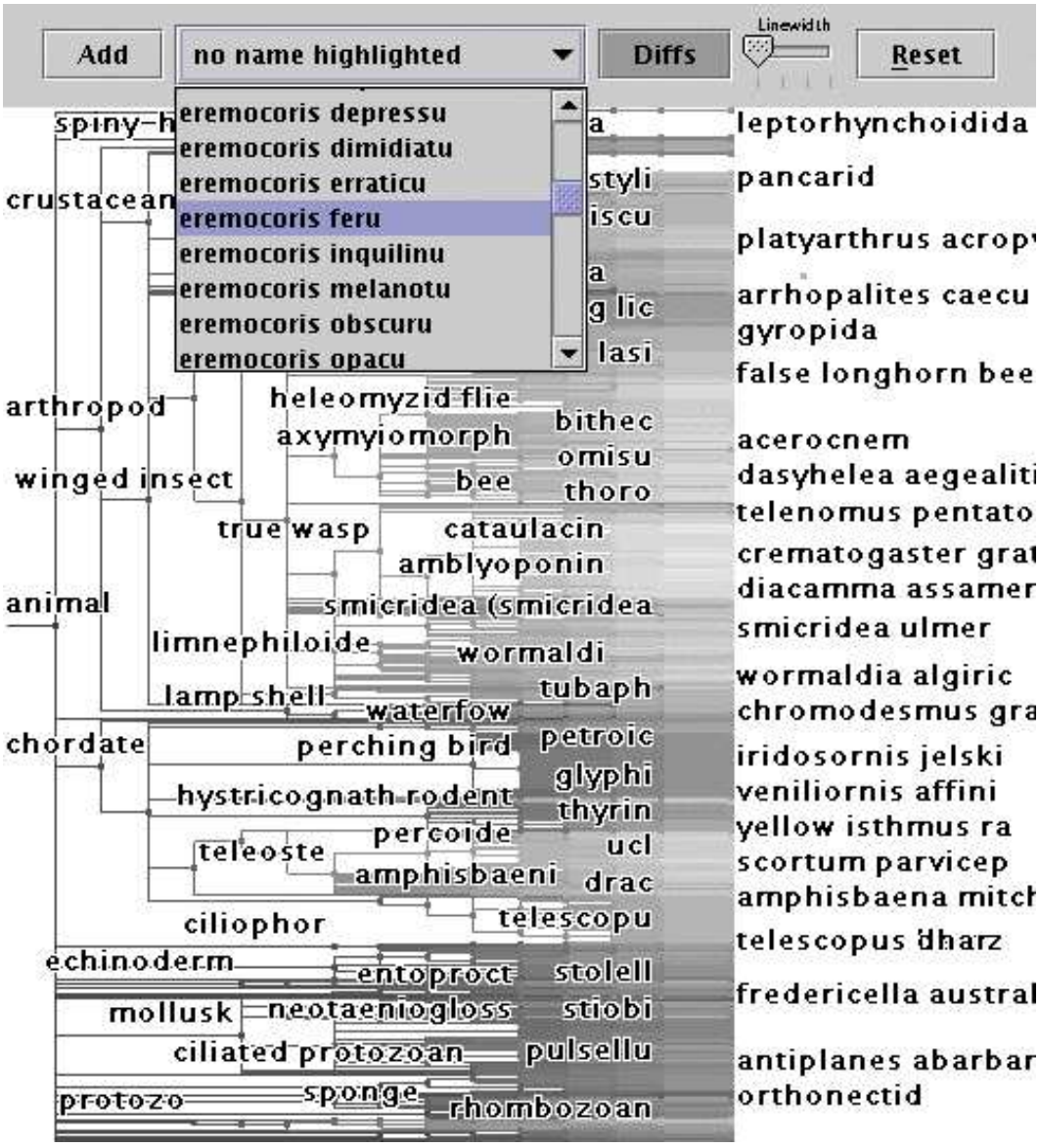

Figure A.5: The TJ1 selection box was a drop-down box. The selection box was difficult to use and did not scale well.

tems, partial search results appear highlighted; in Emacs all search results are highlighted, while Mozilla highlights only the first such match. Since the drop-down box approach was space limited, the `Found` panel, as shown in Figure A.6, was added to TJ1-contest to keep the layout around the canvas uncluttered. This detachment meant that the searching dialog could be extended to show only the multiple matching results and include a query entry box. Matching results appear as a refined list, sorted alphabetically by name, in the `Found` panel dialog and only nodes that contain the search string appear in the list. Items in the list are by default selected as the list changes with the entry of a query. The list selection can be changed by the user using the usual list selection techniques: a click selects a single item, a shift-click will select a range of items and a control-click will select multiple items in the list.

The matching selected results appear as highlighted **found nodes** if the total number of selected items is less than 200; too many matches are not visually useful in the tree layout. Searching incrementally would be slow if every letter for a query entry requires a new search on the total number of nodes. To make the query entry process more interactive, lists of partial search results are cached so searching can use the partial results instead of the entire set of nodes.

The caching stores all previous searches that partially match the query string starting with the first letter. New queries are found in the cache by finding previously cached results that match the first part of the string. If the cache contains results for the query string except for the letter at the end of the string, then the cached list is refined by the new query and stored in the hash table with a key of the new query string.

Initially, the cache starts empty. The first letter that a user enters caches all entries that contain this letter. For example, suppose I want to search for *dolphin*. If I enter `d`, nodes such as *dolphin*, *duck*, *dog*, *bird* and *armadillo* are selected. Next, when I enter `o`, making my search string `do`, the search will start with the cached

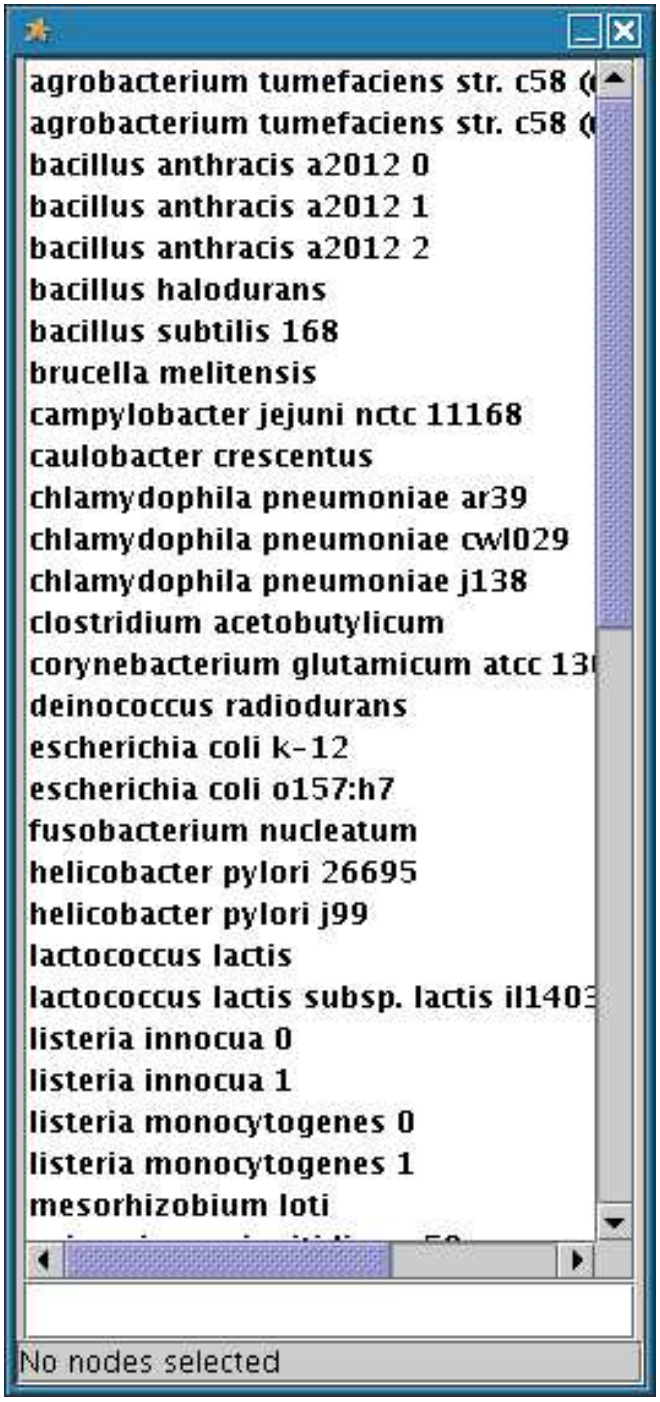

Figure A.6: The TJ1-contest `Found` panel added for the InfoVis 2003 Contest.

result for `d` and the nodes *duck*, *bird*, and *armadillo* will be excluded from the cache for `do`. The cache for `do` will contain nodes such as *dolphin* and *dog*. Further refinements will reduce the search result and use previous caches for a more efficient search time as the search string gets longer.

This approach is reasonable for progressive searches since the typical use of this tool is entering queries starting from the first letter with possible editing changes if the search fails. Searches that need to be entirely rebuilt, for example if *dolphin* is currently entered and the first letter is deleted to search for *olphin*, are reasonably fast as well, but the system is not optimized for those cases.

However, the caching technique used for interactive searching does have the drawback of excessive memory consumption over time for the cached results. The reduction of memory is done with a least recently used (LRU) caching method and only the last 200 search results remain in the cache. Queries that do not match any nodes in the tree and hence would have an empty list of nodes in the `Found` panel are not cached and do not affect the LRU cache.

A.4 Contest results

The following gives an overview of the sets of tasks given for the contest and the results I obtained by using TJ1-contest to solve each of the tasks. The full details of my investigations, including detailed, higher resolution images for each task and videos for some of the tasks, are also available [36].

A.4.1 Tasks suited for TJ1-contest

In this section of the results, I present the details of contest tasks that I solved with TJ1-contest. I explain why each task is relevant in the context of how TJ1-contest is used to make analytical contributions. I present each task section and detail each task, stating the original task and some explanation as to the scope of the task for TJ1-contest related analyses. Some tasks include an ordered list of steps that are

used to produce interesting visualization output, where necessary.

- **Comparison of multiple trees for topological changes**

  In this section, I determine the suitability of TJ1-contest for comparison tasks for differences between several trees. These differences are topological changes: the topological similarities in internal ordering of subtrees, which nodes are added, which nodes are deleted, and which nodes move. We consider the small and large scale differences as equally important since detection of small changes has implications in many applications of tree comparison.

  - Where does the topology change?

    This question poses little difficulty for TJ1-contest since the application is built for this type of analysis. I was able to determine where the topology of each tree changed and I also investigated regions of change to determine the scale of each change relative to each dataset.

    * I expand the visible computed differences, which marks changes in the tree topologies, to view them in greater detail.
    * Mouse-over highlighting in TJ1-contest helps me analyze large scale topology changes as well as individual changes.
    * Subtrees with identical topology are not marked different, so I can focus more effort on the interesting parts of the trees.
    * $\mathsf{mammalia_B}$ differs from $\mathsf{mammalia_A}$ mainly by leaf additions. See Figure A.7 for the TJ1-contest representation of the datasets.
    * $\mathsf{hcil_A}$ through $\mathsf{hcil_D}$ do not change much topologically. See Figure A.8 for the four-way comparison in TJ1-contest.
    * $\mathsf{phylo_A}$ and $\mathsf{phylo_B}$ have identical leaves but are topologically different in several locations. The topological similarities can also be seen in Figure A.9 due to the relatively small size of this dataset.

  - Which nodes are added, deleted?

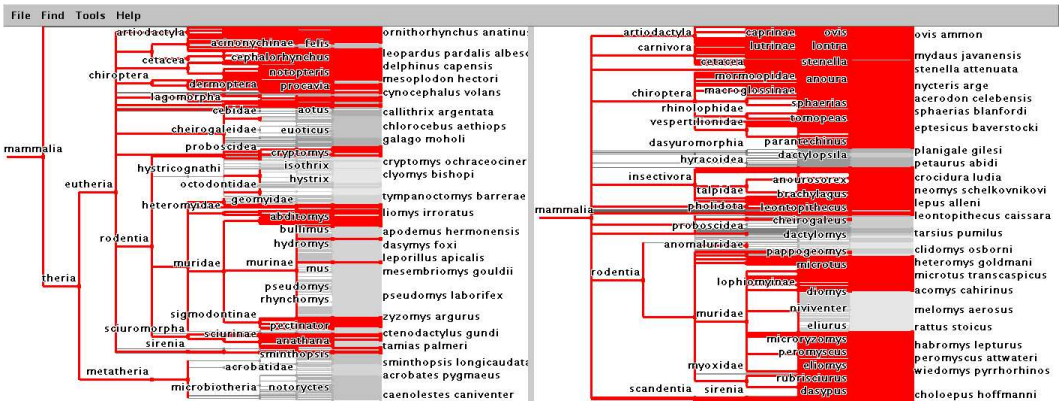

Figure A.7: Contest trees $\mathsf{mammalia}_\mathsf{A}$ and $\mathsf{mammalia}_\mathsf{B}$ compared using TJ1-contest. We mark the topological differences between these two trees in red. The differences, in red, are fewer than in Figure A.26, which uses common English instead of Latin names.

TJ1-contest also easily handles node additions and deletions. I am able to determine exactly where nodes are added or removed by examining the difference-marked regions.

* Additions to leaves are marked as differences in the second tree.
* Deletions to leaves are marked as differences in the first tree.
* Node differences propagate to the root if subtree leaves did not match.
* We do not mark the root nodes as different if the leaves are conserved, as in Figure A.9.
* $\mathsf{mammalia}_\mathsf{B}$ shows more leaves not in $\mathsf{mammalia}_\mathsf{A}$ than the converse.
* Figure A.10 shows the additions and deletions in two classification subtrees: genus *pteropus* and family *pitheciidae*.
* hcil tree leaves show mostly additions and some deletions
* The changes in hcil that are not shown are file modifications, which are attribute based, but would be more interesting since this action is probably more common than creating new files or deleting old files.

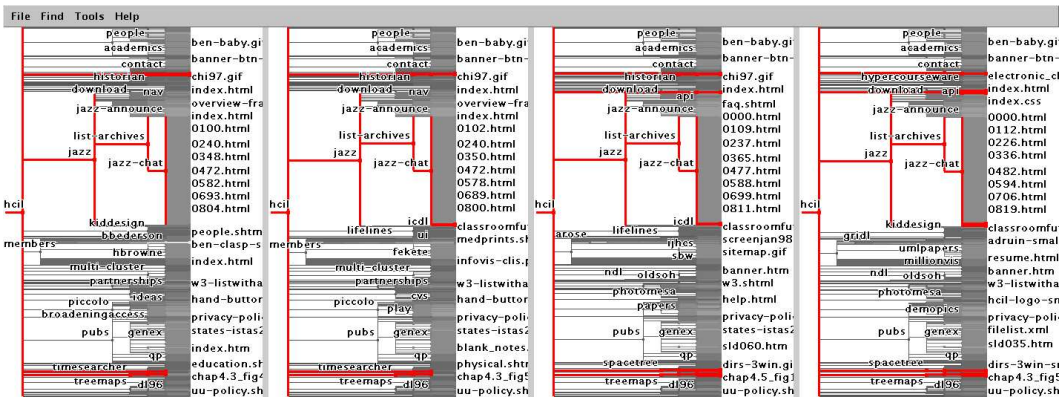

Figure A.8: Contest trees hcil_A, hcil_B, hcil_C, and hcil_D compared using TJ1-contest. We mark the topological differences between these four trees in red. There are few differences among all these trees, which allowed me to manually identify each of them using TJ1-contest.

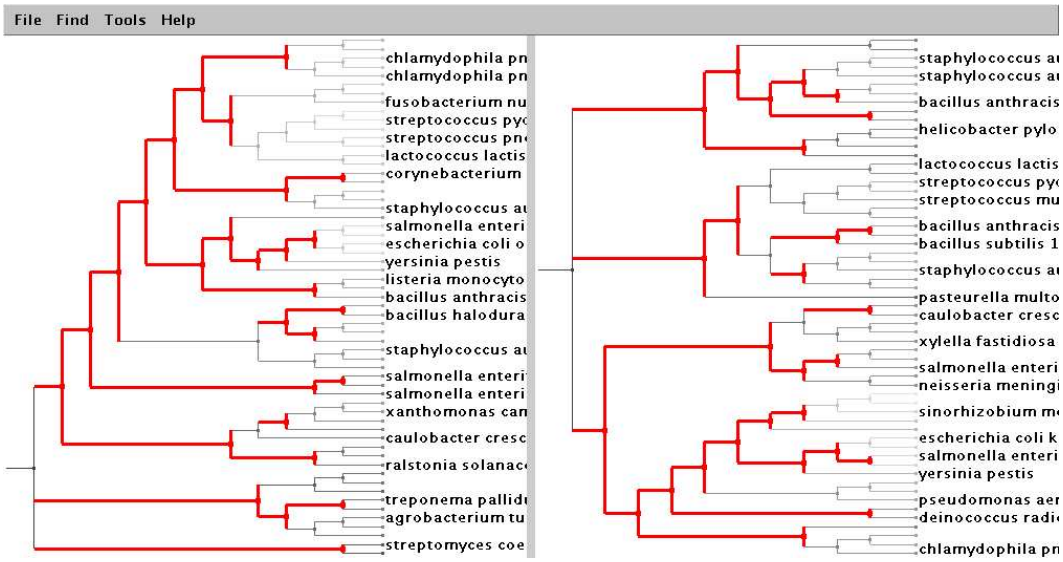

Figure A.9: Contest trees $\mathsf{phylo}_\mathsf{A}$ and $\mathsf{phylo}_\mathsf{B}$ compared using TJ1-contest. We mark the topological differences between these two trees in red, which made my investigation of this relatively small dataset simple. Notice how none of the leaves are different, which indicates no leaves are added or deleted from these trees; due to our comparison process, we do not mark the roots of subtrees when only leaves move topologically.

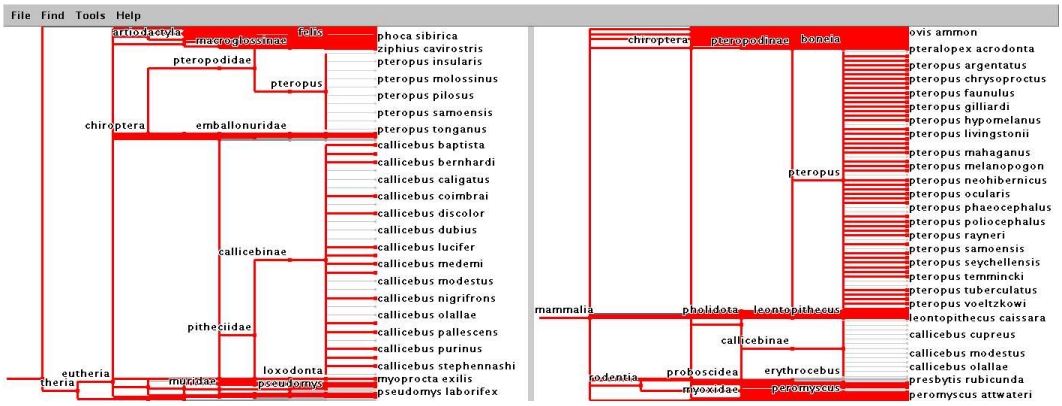

Figure A.10: Detailed differences of *pteropus* and *callicebinae* show additions of *pteropus* from $\mathsf{mammalia}_{\mathsf{A}}$ to $\mathsf{mammalia}_{\mathsf{B}}$, while *callicebinae* shows deletions in that direction.

* In Figure A.11, I show the additions and deletions in two file system directories: *counterpoint* and *iv03contest*.
* In Figure A.9, no additions or deletions are present in the leaves of the phylo trees, but the topological structure has changed. I can also claim that all leaves in these datasets are conserved since the roots of either tree are not marked, which is a property of the comparison functions used by TJ1-contest.

– Did any nodes or subtrees move? Can movements be characterized?

This section asks the hard question of general structural tree analysis and classification of those topological changes. Although I was not able to find and categorize every series of movements for significantly different, large datasets, I was able to characterize several movements in each set of dataset trees, especially for the small but complex phylo datasets.

* I discovered topological movements by examining marked differences, computed by TJ1-contest when the datasets are initially loaded.
* When a complete, topologically unchanged, subtree moves to a new parent, I am able to locate the difference at the level of the new and

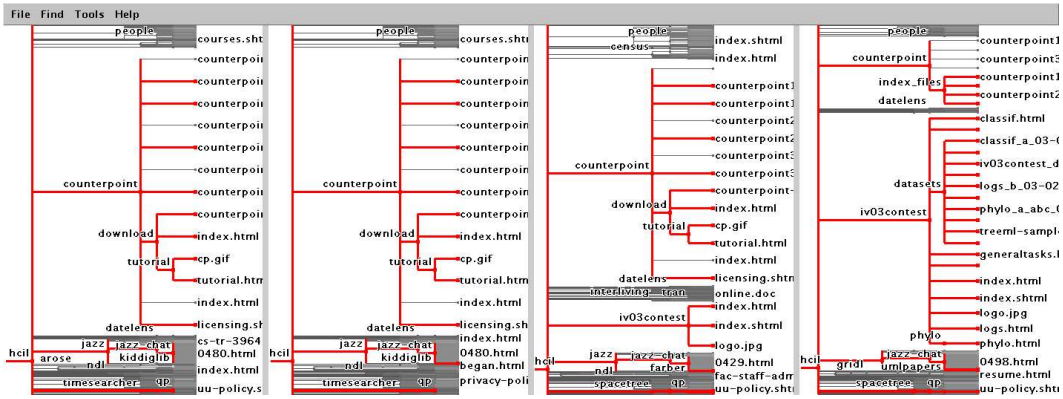

Figure A.11: Detailed differences of directories *counterpoint* and *iv03contest* show additions through the progression of the file system over time. The trees are hcil_A, hcil_B, hcil_C, and hcil_D, from left to right. In hcil_A and hcil_B, the *iv03contest* directory does not appear since it has yet to be created.

old parents in both trees as marked differences.

* There is no automatic characterization of movements in TJ1-contest, so I had to examine and characterize all movements found manually.
* I found a few subtrees that moved in the mammalia datasets.
* Figure A.12 shows one complex movement I found: *pitheciidae* splits into two subtrees and is reparented under *cebidae* from $\mathsf{mammalia}_\mathsf{A}$ to $\mathsf{mammalia}_\mathsf{B}$.
* I determined that no nodes move in the four hcil trees, and node additions and deletions are noticed at the leaf level, although some structural, file system subdirectory additions are also made above the leaf level.
* Most leaves move in phylo trees, only a few small subtrees remain topologically similar.
* In Figure A.13 I show the most topologically similar subtree, marked in the phylo trees, which has seven leaf nodes in an identical topology but since TJ1-contest does not reorder nodes, they appear as they

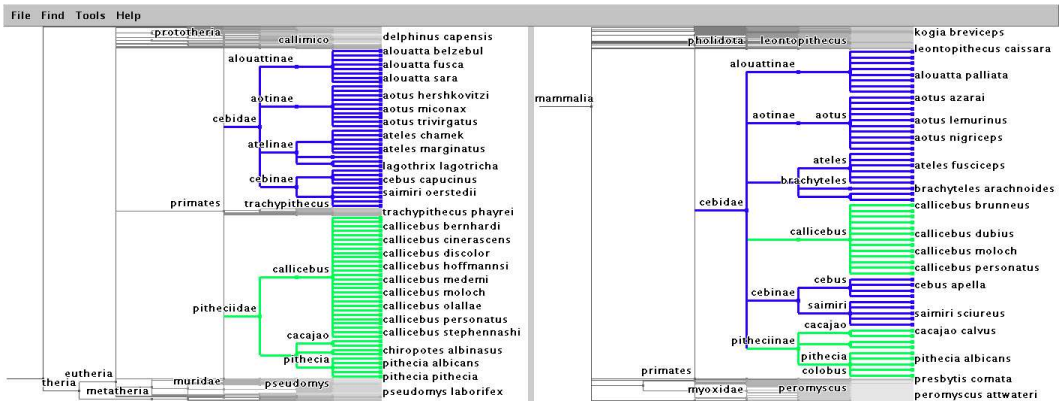

Figure A.12: Movements of *cebidae* and *pitheciidae* from $\mathsf{mammalia}_\mathsf{A}$ to $\mathsf{mammalia}_\mathsf{B}$. In $\mathsf{mammalia}_\mathsf{A}$, the two nodes are roots for two unique subtrees, but in $\mathsf{mammalia}_\mathsf{B}$, *pitheciidae* becomes two separate subtrees rooted under the *cebidae* subtree.

appear in the dataset file.

- **General visualization of tree topology**

  In this section, we focus on more general visualization solutions to understanding the tree topology. I can solve most tasks with single tree visualizations and require neither tree comparisons nor specific tree datasets.

  – How large is the tree? How many levels deep?

    We interpret the size of the tree as the set of numbers that characterize the tree such as fan-out, tree depth, maximum branching factor, and total number of nodes. We consider explorations of subtree properties as well as the entire tree dataset.

    * I determined the tree size, which is the total number of nodes in the dataset, by examining the density of leaf nodes; trees with a branching factor of at least two have more leaves than total nodes and each leaf node in TJ1-contest is initially assigned equal vertical screen space.

    * I estimated the density of leaves by scrolling through one leaf at a

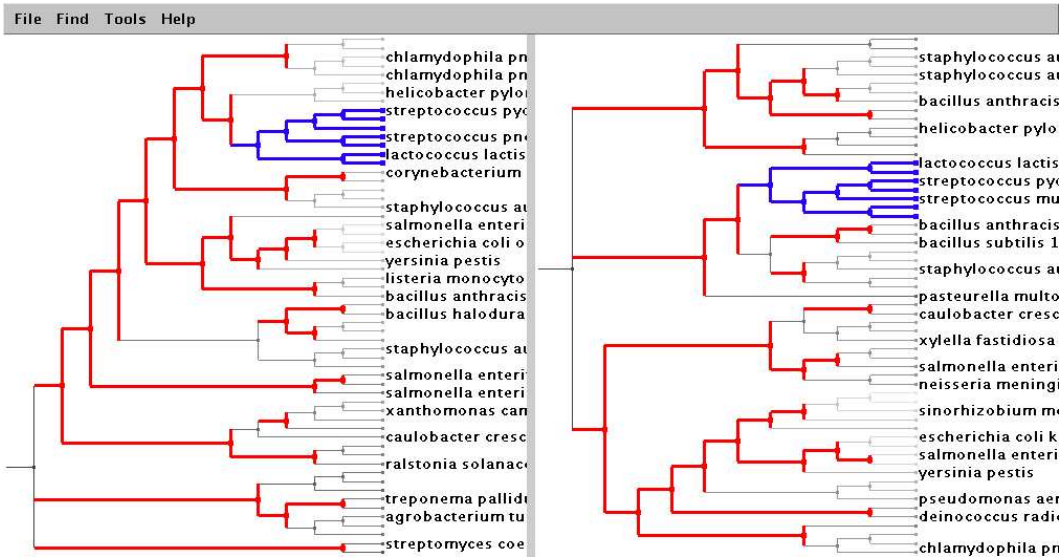

Figure A.13: I marked the largest subtree that remains structurally intact between phylogenetic trees $\mathsf{phylo}_\mathsf{A}$ and $\mathsf{phylo}_\mathsf{B}$. Although the subtrees are not exactly in order, I used TJ1-contest to determine that they are still topologically identical.

time, with up and down arrow keys. For large trees, the time to scroll through the leaves by holding an arrow key down is also effective.

  * The methods I used to determine tree depth in different regions of the trees were not exact. In TJ1-contest, nodes are dimmed relative to their depth; the root is black and nodes are more dim deeper in the tree. Regions of dim nodes correspond to many adjacent internal nodes.

  * Deep trees show a color gradient for depth and because of this, deeper subtrees pop out.

- What is the path of a given node?

  This section is also quite straightforward in TJ1-contest. Once I found a node with the `Found` panel, the path to the root node was followed with the left arrow key. I marked, or expanded, the path with the standard marking, or expansion, methods for individual nodes as the path was

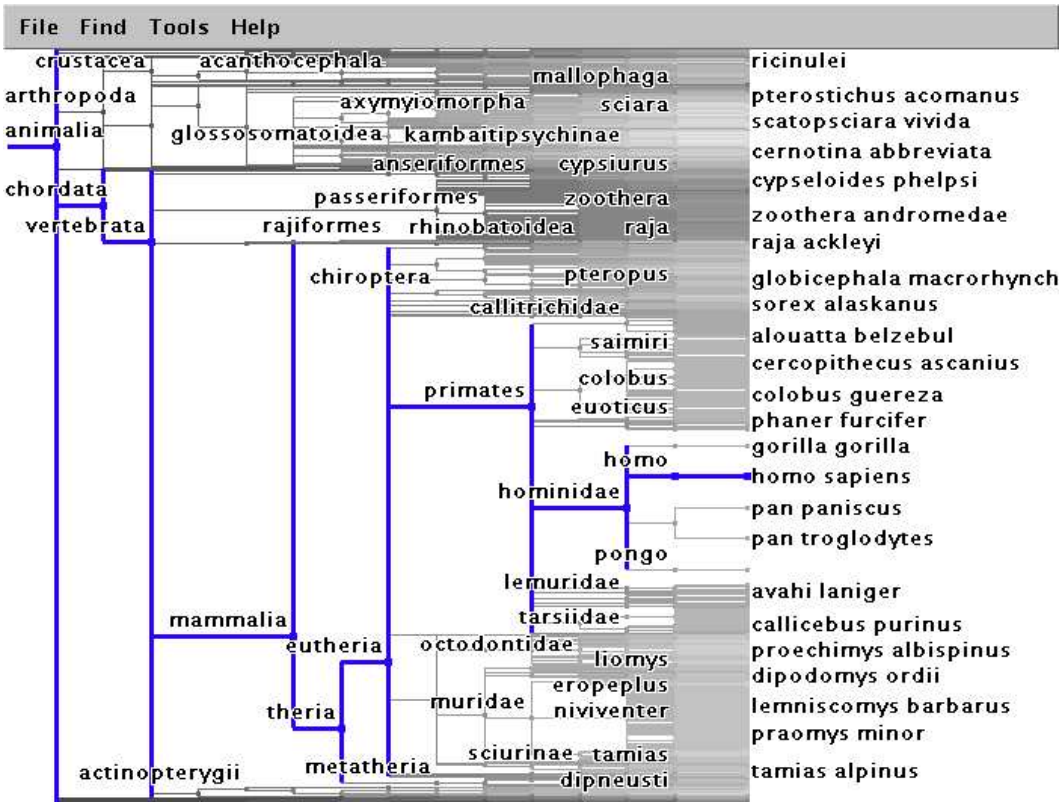

Figure A.14: The path from the root node *animalia* to the leaf node *homo sapiens* is shown in tree $\mathsf{animalia}_{\mathsf{A}}$. Although the path is shown expanded here, the path may be seen in the same tree with no expansion to understand the overall, unaltered location of any node path in a complicated, dense tree. The path is manually expanded by following the path towards the root with the left arrow key and vertically stretching each internal node.

followed.

* I found the ascent path starting from any node to the root interactively with the left arrow key.
* In Figure A.14, I marked and expanded the path from *homo sapiens* to *animalia* manually.

– Local relatives: what are the children/siblings/cousins of a node?

This section deals with more complicated structure than the previous

section. I found the related nodes using the arrow keys to navigate up, down, and through the breadth of even the densest regions of the tree datasets.

  * I found children by using the right arrow to get the first child and then the down arrow to scroll through siblings.
  * I found the siblings using the up and down arrows to scroll through each child of the common parent.
  * Cousins can be found using the up or down arrows as well. When I scroll beyond the first or last child, spatial cousins are highlighted; the actual meaning of cousins is not well defined for this task and spatially adjacent nodes seemed the most natural approach for cousins. We define cousins as the topologically highest node that is spatially, vertically adjacent to a specific node.

– Which branch has the largest number of nodes? Largest fan-out?

  The task of determining the largest number of nodes is also relatively simple with TJ1-contest. When the largest branch is much larger than all other branches, I can easily determine which branch is largest since TJ1-contest initially assigns each leaf node to an equal vertical space and allocates the vertical space for internal nodes from the outer-extreme leaf nodes.

  * The number of leaves determines how vertically large internal nodes are since TJ1-contest initially assigns leaf nodes equal vertical screen space to leaf nodes. Navigation breaks the equality when one subtree is stretched vertically.
  * Marked leaves of subtrees can be visually compared; I can mark subtrees to make estimates on subtree size relative to the entire tree.
  * In Figure A.15, I highlighted the *users* group to determine how large the group is relative to the entire logs_A dataset.

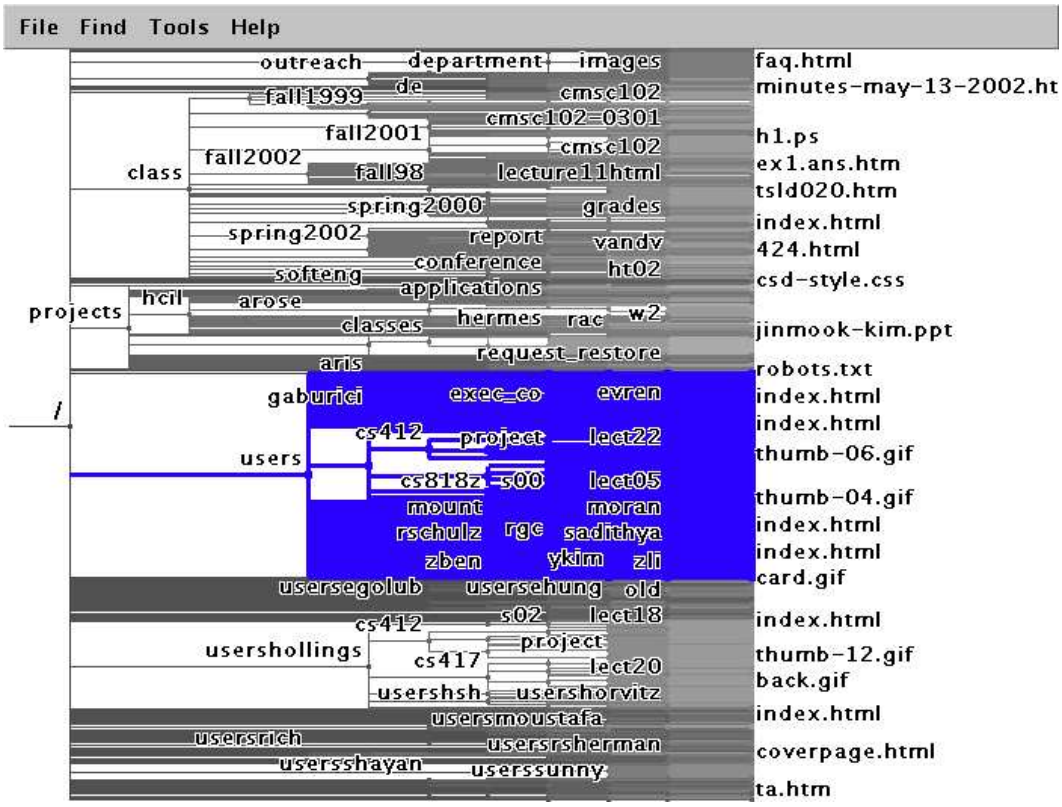

Figure A.15: The subtree *users*, marked blue in the tree logs_A, shows the relative size of the subtree compared to the overall tree size. Since TJ1-contest initially allocates identical vertical screen space for each leaf node, this method of comparing subtree fan-out can only be demonstrated on a tree that has not been stretched or shrunk.

  * For immediate feedback, I use the bounding box for a subtree, which is shown while hovering over the subtree root, to rapidly determine subtree size in a similar manner.

- **General visualization of tree attributes that can be aggregated**

  This section of results focuses on techniques of understanding tree attributes with general datasets. Since TJ1-contest could not solve many attribute related tasks, this section deals mostly with the aggregated analysis of tree structure, which means simple analysis of subtrees. Again, these tasks only

require single tree visualizations and neither tree comparisons nor specific tree datasets.

- What is the number of nodes in a subtree?

  TJ1-contest does not fully support this task, but I found an approach that is sufficient yet not immediately obvious.

  * TJ1-contest does not display the number of leaf nodes for each subtree so I can only determine relative quantities using the tree visualization.
  * I can determine the total number of named nodes in a subtree using the `Found` panel with a fully qualified naming structure, but this solution is not elegant.

- Comparison of branches of the tree: subtrees with most nodes

  This task focuses on determining properties of a subtree that I can use TJ1-contest to quickly analyze. I can examine several subtrees concurrently to estimate the relative number of nodes is each subtree, but exact numbers of nodes are beyond the capabilities of TJ1-contest.

  * I compare subtrees with the `Found` panel and fully qualified names. I use the naming structure from the dataset and the results from the `Found` panel to select structure in the dataset visualization.
  * I select nodes starting with ///*animal*/*mammal* to show the number of mammals.
  * By entering *mammal* into the `Found` panel for $\mathsf{animalia}_{\mathsf{B}}$, I found *mammal-nest beetles*, which are not mammals.
  * There are very few non-mammals with *mammal* in their name; I deselected the non-mammals in the `Found` panel to find the root of the *mammal* subtree. I could have arrived at the same result by searching for and selecting the ///*animal*/*mammal* tree directly from

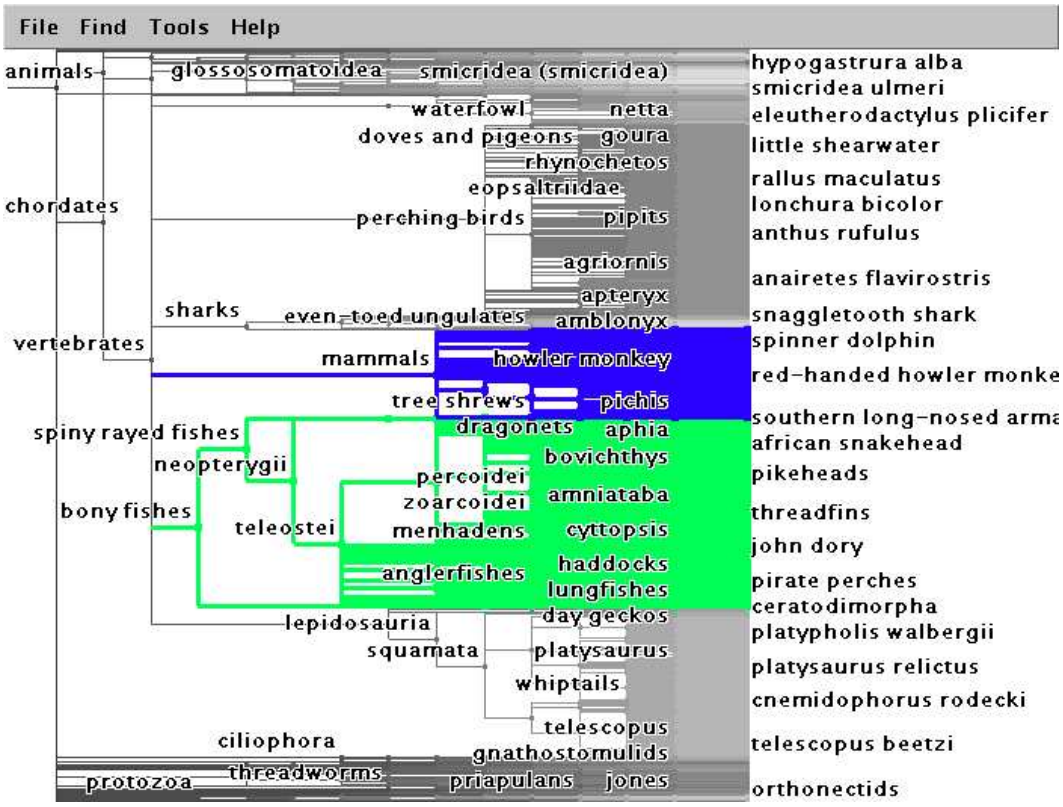

Figure A.16: *mammals* and *bony fishes* marked in animalia$_B$, which we can use to determine the relative sizes for these two subtrees. The *mammals* subtree is approximately half the size of the *bony fishes* subtree. The dataset has not been skewed by navigations so each leaf node is assigned equal vertical screen space.

the entire dataset, but that is slower than my method of pruning the tree.

- I could then grow the $///animal/mammal$ subtree or mark it for simultaneous visual comparisons with other interesting subtrees.
- By marking as described, I produce Figure A.16 with *mammals* and *bony fishes* marked in different colours.

- **General visualization of known items**

This section deals with visualization of known items in generic datasets. In

this context, nodes are considered to be known in the sense that searching for a particular node with knowledge of the name or path to the node from an ancestor node is general knowledge.

- Which nodes have a label containing the string *giraffe*?

  This task is straightforward with TJ1-contest using the capabilities of the `Found` panel.

  * I type *giraffe* into the `Found` panel and all *giraffe*s are highlighted with the colour of the Found group.
  * I resize the `Found` panel results for the Found group using the `Groups` panel.
  * Figure A.17 shows the result I achieved after searching for *giraffe* in $\mathsf{animalia}_{\mathsf{B}}$.

- Locate a node knowing its path.

  This task requires the navigation control of TJ1-contest. I browse the tree, directed by the hierarchical naming structure, using interaction boxes created with mouse controls.

  * I can either use the `Found` panel to find a node of interest if I know the label or browse through the tree structure from the topological root if I am interested in a node along a known path.
  * If the full path is known, browsing the tree with mouse-over highlighting may be faster than searching every leaf node.
  * I also reduce searching by starting closer to results using the `Found` panel to locate well-known internal nodes.
  * The `Found` panel method is especially helpful with bushy subtrees; browsing is difficult when viewing the children of a node with a branching factor larger than the number of vertical pixels on screen.

- Go back to a node you have visited before.

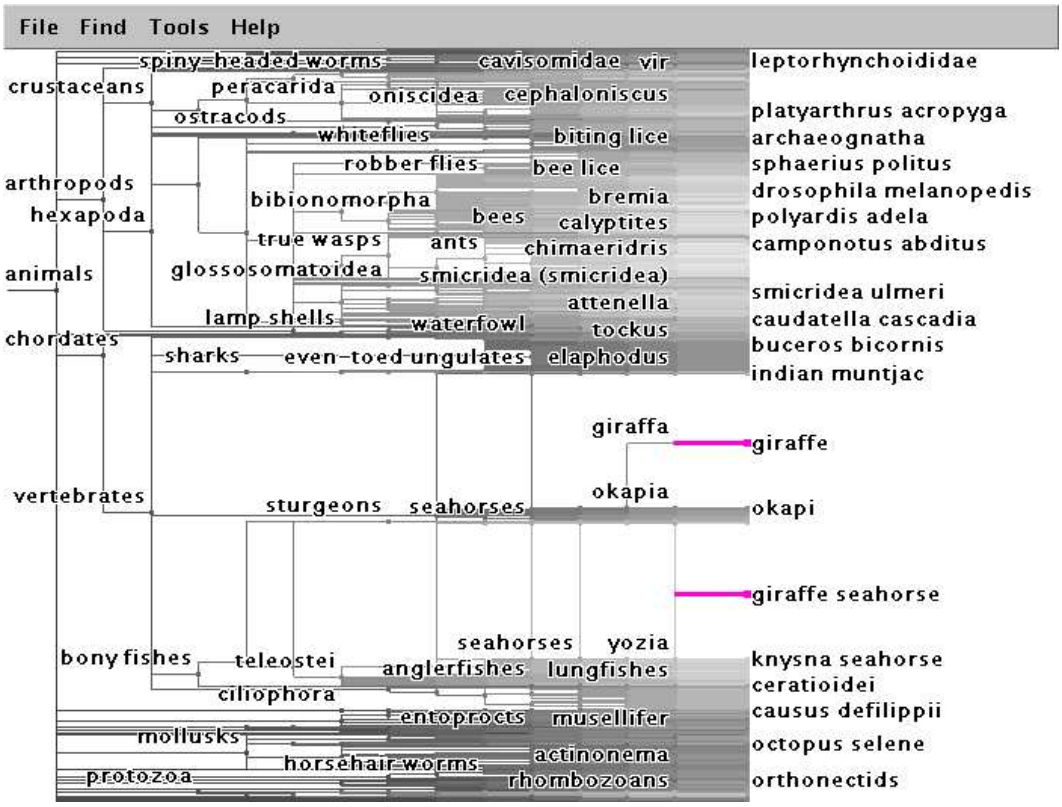

Figure A.17: Result of *giraffe* search in animalia$_B$, achieved by searching using the `Found` panel and growing the results for the Found group in the `Groups` panel.

This task would have made navigation in TJ1-contest much easier, but is left for future work in a more general approach to all Accordion Drawing applications.

* There is no explicit `undo` feature in TJ1-contest.
* I could return to a subtree after exploring other parts of the tree by marking interesting subtrees to either remember where I was or use the `Group` panel to grow the marked tree.

- **General visualization of labeled items**

This section relies on the labeling provided by TJ1-contest to give context of tree topology as well as readily available information for visible nodes with

enough screen space to display a label.

- Review all the labels in a subtree

  This task is not possible in the visualization of dense areas of a dataset, but TJ1-contest can extract labels with the `Found` panel if this is the case.

  * All labels in a subtree can be extracted through the `Found` panel. This technique is not necessary if a subtree is not too dense since labels that have enough space to draw are shown in the tree visualization.
  * TJ1-contest limits results in the `Found` panel to nodes matching the keyed-in entry. If I wanted to see only nodes in a particular subtree, such as more nodes than the `Found` panel displays with direct matching, I could find the subtree with the panel, grow the subtree root node and then review the labels of the subtree.
  * I could examine all labels on the subtree using mouse-over highlighting or by sufficiently decreasing the node density. We can also lower the font size or individually stretch sections of the subtree to see more labels.

- **General navigational visualization and browsing**

  This section focuses on the navigation abilities of TJ1-contest. Following a known path in the dataset with TJ1-contest is powerful when I want to browse the topology with no predefined species of interest. We are not limited to following a naming structure since it would also be useful to follow properties of a tree, should they exist, that are not related to the displayed labels. An example of this directed browsing would be manually exploring the computed differences in a tree.

  - Explore the tree by performing a series of up and downs in the tree: you

are looking for a cute animal. You look into mammals, then primates, then gorillas, and chimpanzees, but you realize that they are not that cute. You then go to felines, to tigers and cheetahs...

This task is straightforward in TJ1-contest with the mouse browsing and navigation tools. Mouse-over highlighting, mouse resizable trees and the keyboard interface are all useful tools depending on the user and the exploration task.

I performed the following to explore the animalia$_B$ dataset:

1. I grew *vertebrates*, *mammals* bigger using mouse-over highlighting to find the item, then typed `b` to make the tree larger, using the `Group` panel to first ensure I was making the mouse-over group larger.
2. I found *primates* in *mammals*, then resized it with the interaction box growing method, finding that *great apes*; *gorilla* and *chimpanzee* appear in the *great apes* subtree when grown using the same method.
3. I used the mouse to highlight *primates*, then repeatedly pressed the up arrow key until *carnivores* was highlighted since *mammals* was too dense in the region outside of the grown *primates*; dense regions are easy to step through with the keyboard arrows.
4. I then grew the *carnivores* selection, again with the keyboard, until it was large enough to see the *cats* subtree.
5. Using the mouse, I grew the *felinae* subtree enough to see *cheetah* and *tiger*...

– Following those steps, I produced Figure A.18.

- **General management of analysis**

This section deals with general techniques that TJ1-contest uses for analysis. Namely, this section only deals with marking nodes of interest since other analysis methods addressed by this section such as editing, saving settings

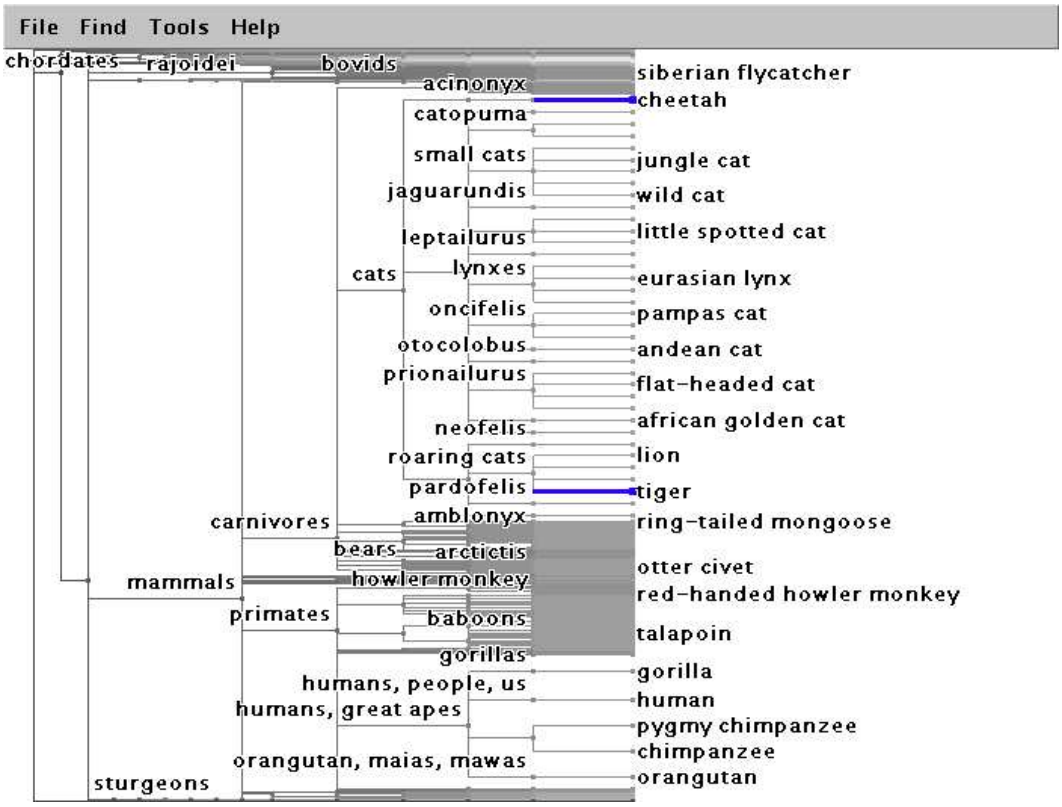

Figure A.18: Result of browsing for cute animals in $\text{animalia}_{\text{B}}$. We marked *tiger* and *cheetah* since they were the cutest animals. This figure also shows multiple areas of focus in TreeJuxtaposer, while still providing context with the squished regions.

and a history of analysis are tasks that we are unable to complete with TJ1-contest.

– Marking nodes of interest

TJ1-contest uses marking for many different functions such as resizing or a pseudo-history of navigation when I proactively mark regions that are visited. We consider marking to also include computed differences, search results and mouse-over highlighting groups.

* We can mark up to four `UserGroups` to highlight nodes of interest.
* Our granularity of marking is either node or subtree.

* We can mark multiple nodes or subtrees in each group.
* A node may belong to multiple groups simultaneously.
* The last group selected will be visible over other marks. The groups may be cycled through, with the graphical interface or using key g, to select the current marking group and change the priority of marks.
* We mark the best corresponding node on each tree if more than one tree is loaded and if that node has a correspondence to a user marked node.

- **Application specific tasks section with phylogenetic trees**

  This section deals with the tasks related to $\mathsf{phylo_A}$ and $\mathsf{phylo_B}$ datasets, constructed by evaluating genomic properties of two proteins.

  - Map the similarities between the two tree topologies, which would indicate co-evolution and possibly where two proteins were not co-evolving

    This task is one of the strongest abilities of TJ1-contest. Similarities are instantly visible in these small datasets as the nodes that are not marked as differences. The largest unmarked subtrees are indications of co-evolution. The following was noticed:

    * The leaves in $\mathsf{phylo_A}$ were all in $\mathsf{phylo_B}$ and vice versa. See Figure A.9.
    * Some leaf nodes have identical names in the same tree.
    * My analysis of TJ1-contest assumed all leaves have one-to-one relationships using a renaming scheme for identically named nodes. However, since the best corresponding node criteria is only onto, the renaming process to disambiguate similarly named nodes, namely appending a number to the node name but not the node label, might have affected the computed differences. The best naming scheme to produce the fewest differences or largest similar subtrees is not determined.

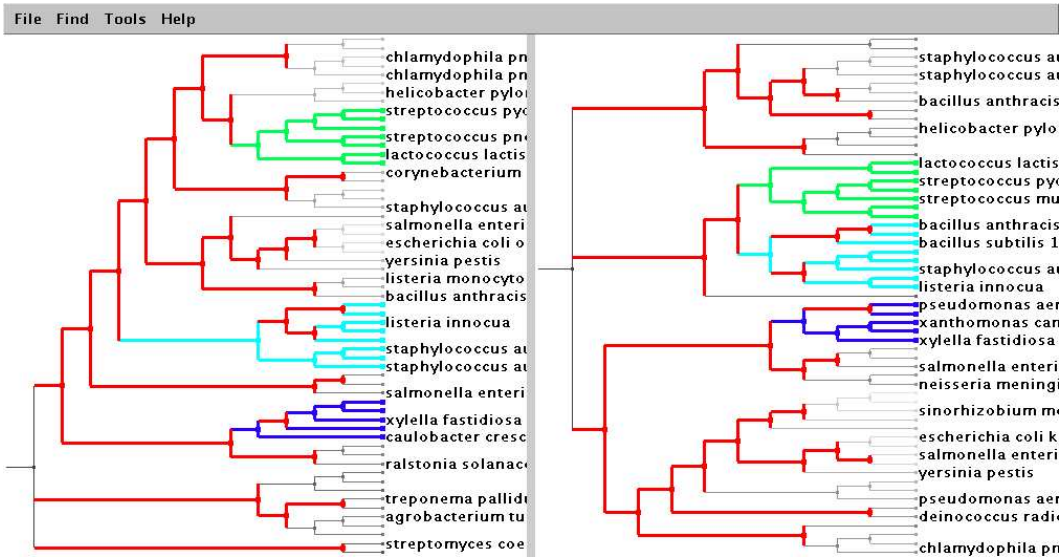

Figure A.19: The three most topologically similar subtrees marked in $\mathsf{phylo}_\mathsf{A}$ and $\mathsf{phylo}_\mathsf{B}$. The leaf assignment and renaming is automated and the leaf relationships cannot be edited for a better matching.

- TJ1-contest is only able to automatically assign best corresponding leaves; editing leaf relationships was not possible.
- Different leaf relationships produce different tree comparisons.
- A subtree of 7 leaf nodes matches topologically. See Figure A.13 for this subtree marked, in the phylo trees.
- Another subtree of 5 leaf nodes nearly matches with only one internal node topologically different.
- A third subtree of 8 nodes nearly matches with three internal nodes topologically different.
- See Figure A.19 which has the three most topologically similar subtrees marked in $\mathsf{phylo}_\mathsf{A}$ and $\mathsf{phylo}_\mathsf{B}$.

- **Application specific tasks section on classification trees**

  This section deals with the tasks related to comparisons of $\mathsf{mammalia}_\mathsf{A}$ and

mammalia$_B$ datasets as well as other visualization tasks with animalia$_A$ and animalia$_B$. Comparisons are not done with the animalia datasets since they are too large to evaluate with TJ1-contest.

- To what extent are the differences in the classifications due to differences in how animals are thought to be related?
- Are there other kinds of differences and can you explain them?

  These tasks intend to ask general questions on types of differences found during explorations of the mammalia$_A$ and mammalia$_B$ dataset comparisons. Although there were many differences found with TJ1-contest, I was able to classify each style of difference that I investigated as one of the following: an addition where nodes, typically leaf nodes but possibly including their ancestors as well, exist in mammalia$_B$ but are not in mammalia$_A$; deletions, which are opposite from additions; and subtree movements where whole subtrees are re-rooted.

  * I determined that the differences are mostly due to: additions to the tree, deletions from the tree, and slight modifications such as splitting where a leaf node in animalia$_A$ became a subtree with children in animalia$_B$. See Figure A.10, which shows leaf additions in the *pteropus* subtree and leaf deletions in the *callicebus* subtree.
  * I quantify additions and deletions on the leaves by examination of the rough quantity of marked differences in the leaf level since the leaves are equally spaced at that level; the amount and location of marked difference indicate the distribution of added nodes. See Figure A.7, which shows the relative number of additions for mammalia$_B$, on the right, as large regions of difference while mammalia$_A$, on the left, has had a few leaves deleted.
  * I highlighted the *rodentia* subtree, a high-level classification group, with blue in animalia$_A$, then the *rodentia* subtree in animalia$_B$ in

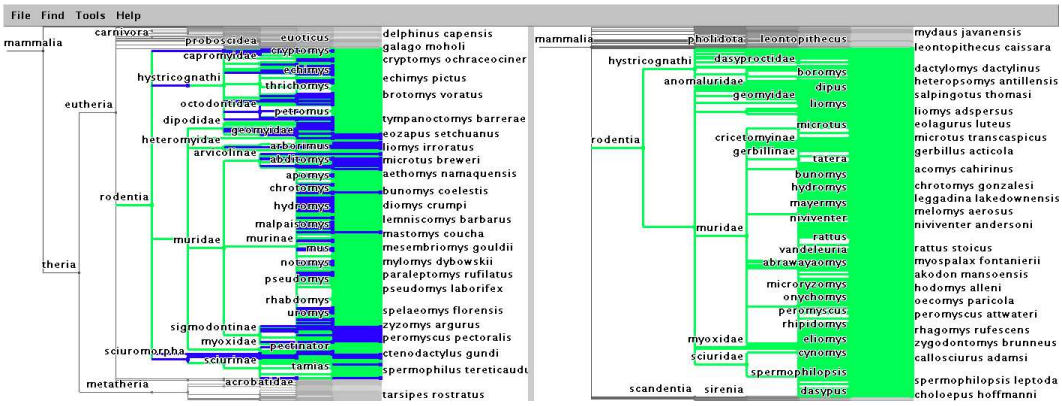

Figure A.20: *rodentia* subtree marked in mammalia_A and mammalia_B. The subtree is first marked in mammalia_A in blue and then marked in mammalia_B in green. The green marking in mammalia_B overwrites the blue in mammalia_A since the green is currently at a higher priority for all dataset views. No blue marks exist outside of the green marked subtree, which indicates that no *rodentia* from mammalia_A are misclassified as something other than *rodentia* in mammalia_B.

green. After these actions, there were no blue nodes visible in the mammalia_B green subtree. This allowed me to visually conclude that no species classified under *rodentia* in mammalia_A was classified under anything but *rodentia* in mammalia_B.

The green marks did not cover the blue marks in mammalia_A completely, which indicated that mammalia_A had more species classified in the *rodentia* subtree than mammalia_B; marking *rodentia* in animalia_B had the same results in animalia_A. Similar results with other large subtrees implied that, although not a complete investigation, the mammalia trees showed mostly differences in lower level nodes; see Figure A.20 for the result using TJ1-contest.

* Some differences such as the movement of *pitheciidae* from *primates* in animalia_A to *cebidae* in animalia_B were found through exploration; the subtree marking capability sped up the exploration process, as shown in Figure A.12.

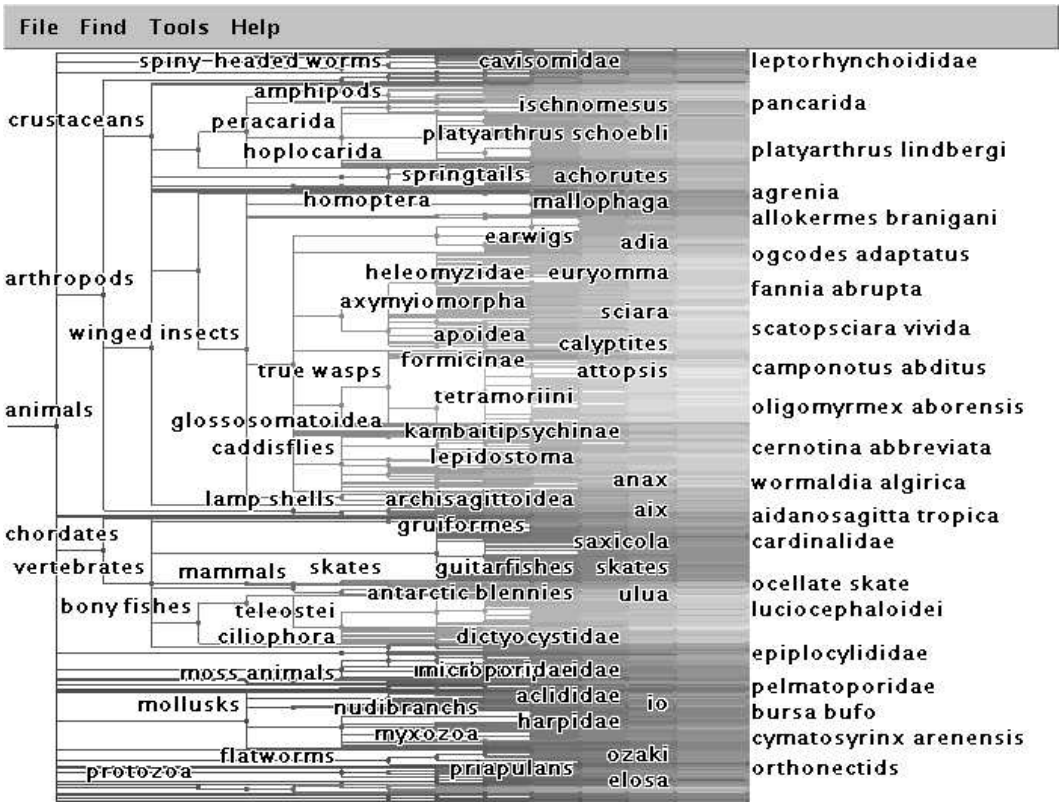

Figure A.21: animalia$_A$ displayed in its entirety with common names. This tree has 190,265 total nodes, 154,922 of those nodes are leaves, and has a height of 16.

– Can you say in how many different subtrees a particular common name, such as *dolphin* or *horse*, is used? How closely are these animals related? These questions are answered quite easily by TJ1-contest with the `Found` panel. Guaranteed visibility is able to show how closely the animals are related and multiple focus points allowed me to view all resulting matches concurrently.

  * With animalia$_A$, as in Figure A.21, using fully qualified names: `Found` panel returns 53 leaf and non-leaf *dolphin*s, see Figure A.22.

    · Of the 53 positive search results, only *myzomela adolphinae* was probably not named with respect to phylogeny or morphology of

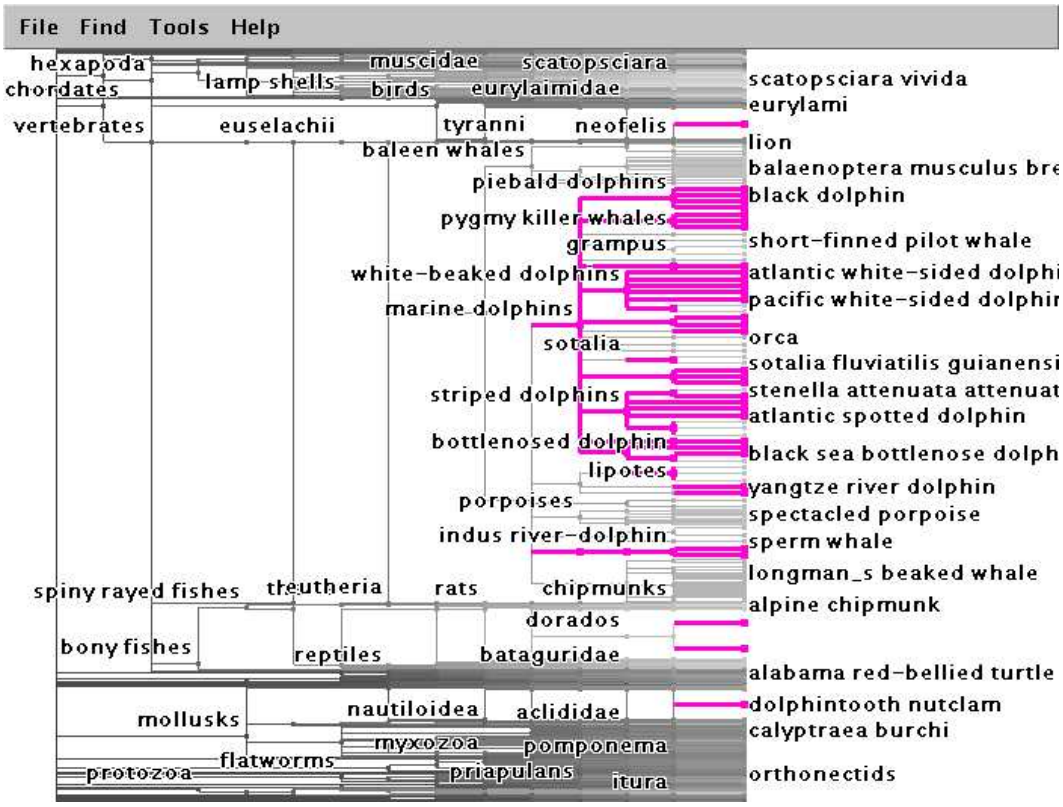

Figure A.22: Result of *dolphin* search in $\text{animalia}_{\text{A}}$ with common node names. 53 leaf and non-leaf *dolphin*s were found using the `Found` panel.

*dolphin*s, since *dolphin* only occurs as a substring.

- Many of the search result *dolphin*s were found in the *marine dolphins* hierarchy.

* Search for *horse*: `Found` panel returns 47 leaf and non-leaf *horse*s, see Figure A.23.
  - In addition to mammalian *horse*s, *horse* appears in many different subtrees across different parts of the classification tree: *arthropods*, *insects*, *seahorses*, and *snails*.
  - The animal species with *horse* in their names are not closely related at all.

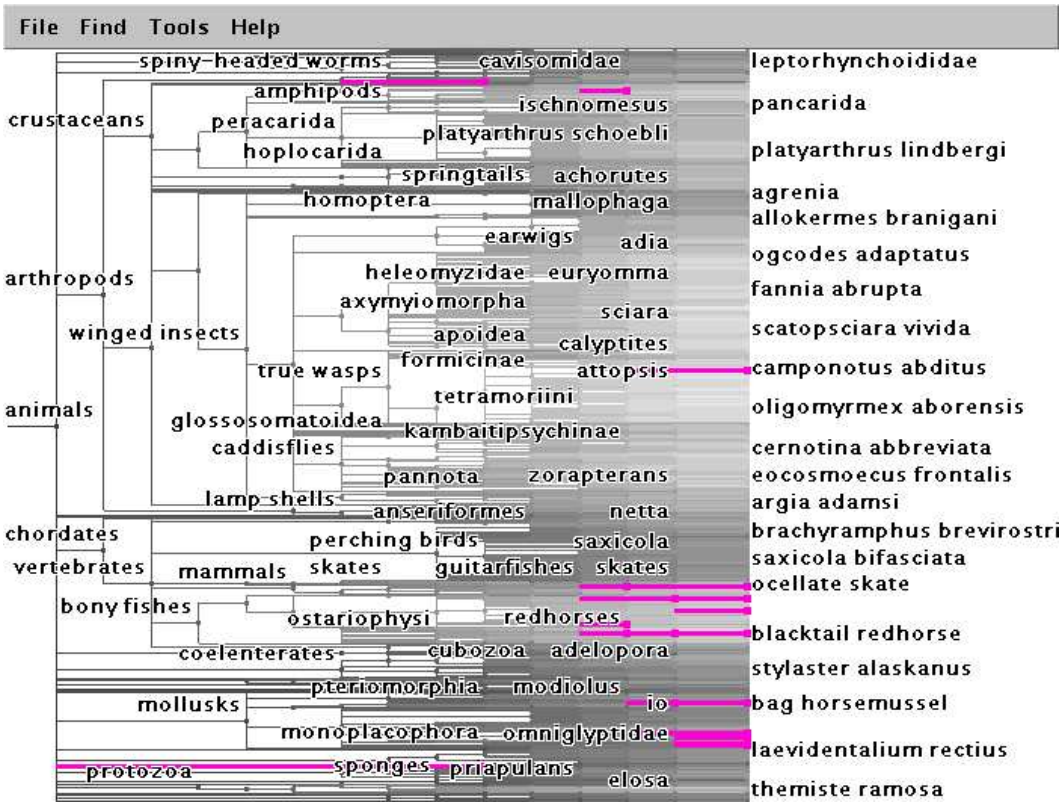

Figure A.23: Result of *horse* search in animalia$_A$ with common node names. 47 leaf and non-leaf *horse*s were found using the `Found` panel.

· Several *horse*-named roots of subtrees exist, such as *horsehair worms* and *horseshoe crabs*, which include only species that do not have *horse* in their names. These species are all lacking common names in mammalia$_A$ and are therefore labeled with the fallback: Latin names; perhaps their common names do/would include *horse*.

– Are common names a good guide to understanding relationships?

This task concludes the statements and findings from searches for *dolphin* and *horse*.

∗ Common names are not a good guide to understanding relationships.

Several common names were investigated and results would indicate that common names are used frequently to describe morphological features of yet unclassified and unnamed classes of species or of species themselves.

* Common names lack structure and do not have the same somewhat hierarchical classification structure as their Latin equivalents. The Linnæan system of categorizing species, into several layers commonly referred to as a classification tree, is also used to provide a standard for further classification.
* Common names may have, for example, historical or geographical influences and therefore are most of the time not helpful in understanding relationships in all cases.
* One classification may even look different from an identical classification tree if a naming convention is not adhered to: for example, I found that *marmota vancouverensis* is *vancouver island marmot* in $\mathsf{mammalia}_{\mathsf{A}}$ while $\mathsf{mammalia}_{\mathsf{B}}$ labeled the same species as *vancouver marmot*. See Figures A.24 and A.25, which are identical, expanded sections of common and Latin versions of the mammalia trees under the class *marmota*, but show the many naming differences in the common tree versus the Latin tree.
* Some common names may be simple and included in other common names: *horse* occurs in *seahorse*; the `Found` panel was able to focus in on sections of species with user defined selections in the search window.
* For species such as *dolphins* that are not expected to occur frequently across different species, it was interesting to see non-mammals occur: a mollusk, two bony fishes, a perching bird; they may either have *dolphin*-like morphological properties or *dolphin* in their name by

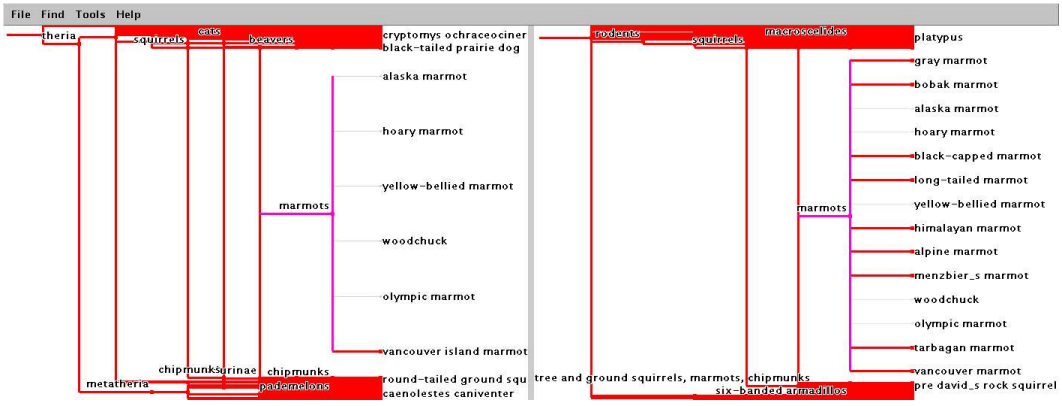

Figure A.24: The class *marmots* subtree expansion in $\mathsf{mammalia}_\mathsf{A}$ and $\mathsf{mammalia}_\mathsf{B}$ with common names. Note how in this tree *vancouver island marmot* and *vancouver marmot*, the same species, is marked as different since these names are not unique. Compare with Figure A.25 in which this species is called *marmota vancouverensis* in both cases and is therefore not marked as a difference. This is a case against using common names as a classification structure if there is no concensus on a unique species name.

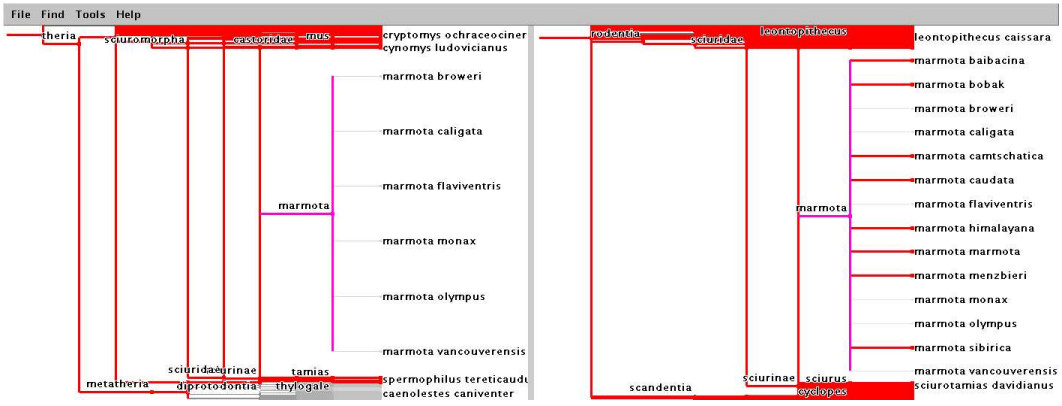

Figure A.25: The class *marmota* subtree expansion in $\mathsf{mammalia}_\mathsf{A}$ and $\mathsf{mammalia}_\mathsf{B}$ with Latin names. Note how in these subtrees the species called *marmota vancouverensis* is consistent and agreed upon by both datasets. Compare with Figure A.24 in which this species does not have an agreed upon common name and therefore marked as a difference. This is a case for using Latin names as a classification structure since they are more likely to be unique and agreed upon, at least in all examples I found in these datasets.

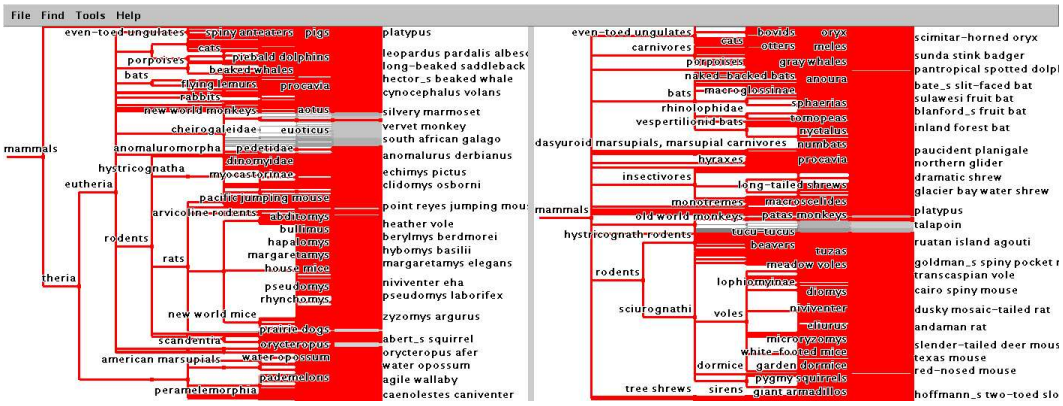

Figure A.26: Contest trees $\mathsf{mammalia}_\mathsf{A}$ and $\mathsf{mammalia}_\mathsf{B}$ compared using TJ1-contest. The topological differences between these two trees are marked in red. The differences, in red, are more plentiful than in Figure A.7, which uses Latin instead of common English names.

some other origin such as *adolphinae*, which I determined to possibly mean something mountain-related after referring to the common name found in a web search, from *myzomela adolphinae*.

* Common names were useful for providing recognizable names but they dramatically impede comparison. Figures A.26 and A.7 show the large naming problems as differences between the respective common and Latin versions of the same mammalia trees.

– How many species are named after biologists named *Townsend* in both the Latin and common name trees?

– Can you look at the pattern of names to deduce where in the world or on what kinds of animals *Townsend* might have done research?

This is a more general question of basically determining the usefulness of Latin trees, or common trees, to deduce certain reasons why a species has a particular name. In this task, the commonalities used to determine origin of species names are biologists named *Townsend*.

* Latin $\mathsf{animalia}_\mathsf{A}$, Figure A.27: 51 leaf and non-leaf *Townsend* nodes

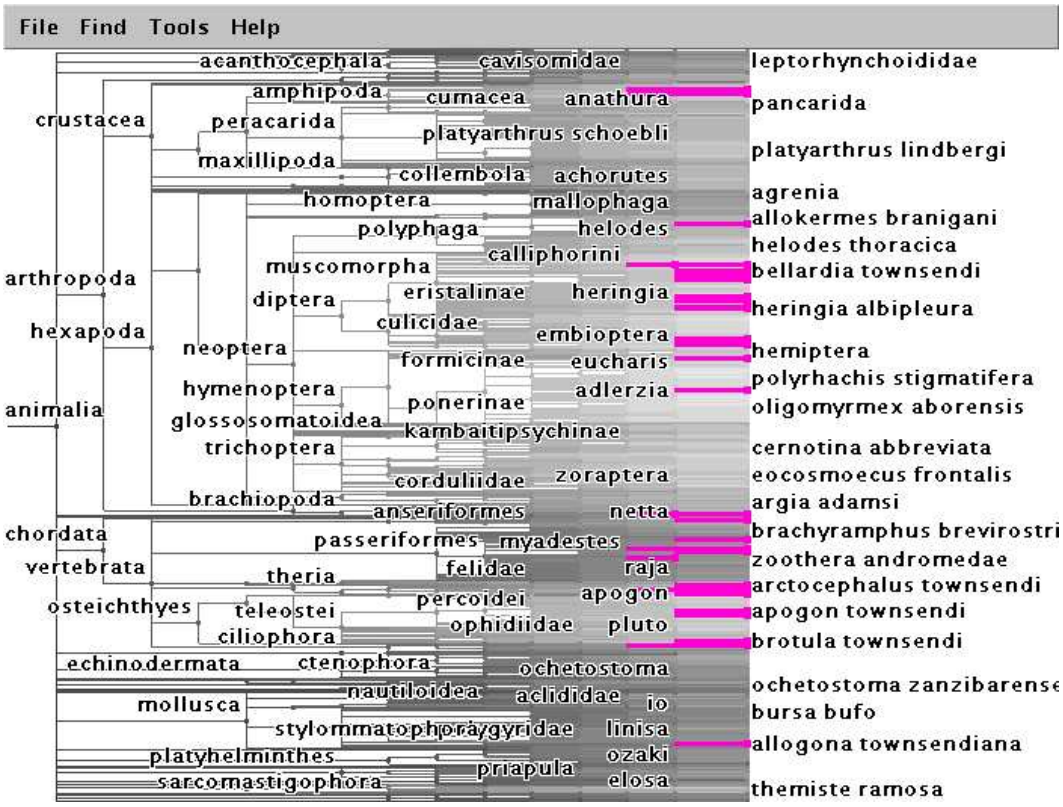

Figure A.27: Result of *Townsend* search in $\mathsf{animalia}_\mathsf{A}$, with Latin names. 51 leaf and non-leaf *Townsend* nodes were found. It is evident that the results are not of a particular class of animals, but spread out in the *animalia* kingdom.

were found.

* Common $\mathsf{animalia}_\mathsf{A}$, Figure A.28: 45 leaf and non-leaf *Townsend* nodes
* Some Latin names appear in common trees since nodes with no common name used the Latin name as a label. The only purely common names returned by the search that I recognized as common are: *townsend eualid*, *townsend snapping shrimp*, *townsend's chipmunk*, *townsend's dwarf gecko*, *townsend's ground squirrel*, *townsend's mole*, *townsend's pocket gopher* and *townsend's vole*. More of these are larger species, most in the *rodents* subtree, which might be indica-

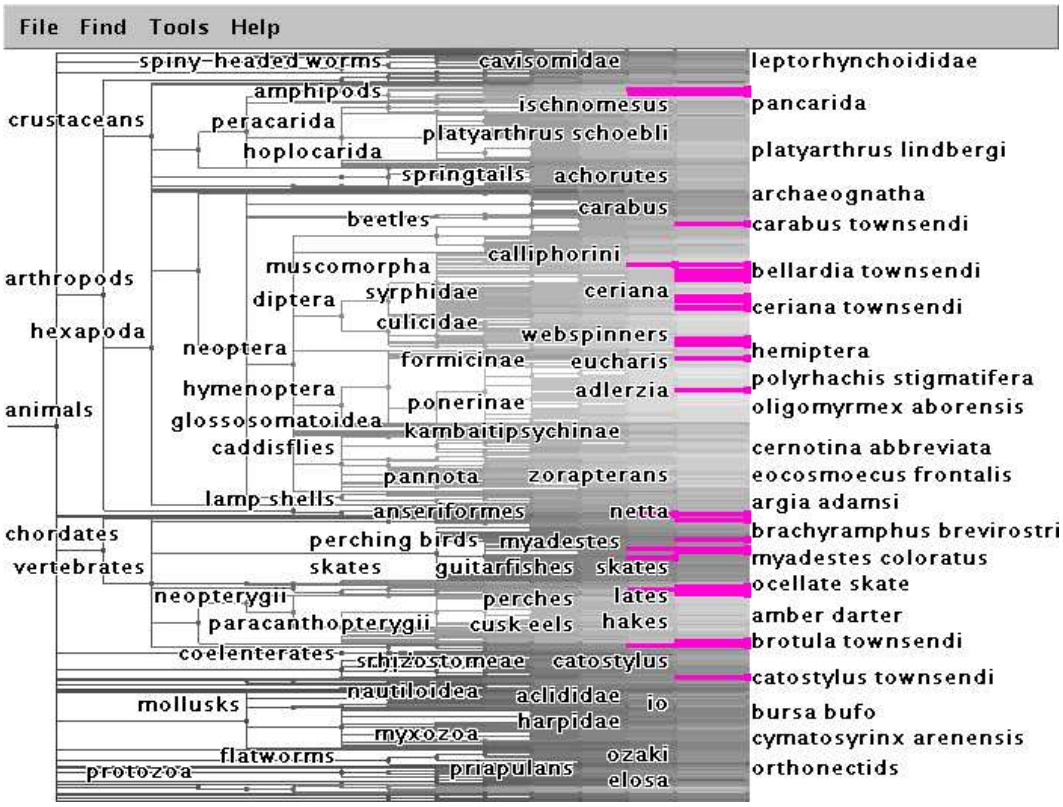

Figure A.28: Result of *Townsend* search in animalia$_A$, with common names. 45 leaf and non-leaf *Townsend* nodes were found. It is evident that the results are not of a particular class of animals, but spread out in the *animalia* kingdom.

tive that most large species are given common names for this dataset, but that is unconfirmed.

* Names returned in the search did not show a clear pattern that could be used to deduce where in the world biologists named *Townsend*, or geographical locations with *Townsend* in their name, might have done research. Furthermore, there is no indication that there was only a single biologist named *Townsend*.

* Common names give a range of possible geographic locations for types of *chipmunks*, *shrimp*, and *bats*. It is not possible that all *Townsend* animals were cohabitants of the same geographic location.

* The search returned quite a range in the classification tree and, therefore, the search highlights were distributed throughout the tree.

– *Spirulida* and *spirurida* are two nodes in two different subtrees. If a user types in the wrong one, what kind of feedback is used to alert the user quickly?

  Although the TJ1-contest application does not provide feedback for user errors, such as the search results not returning an expected node for typographic errors, I was able to quickly fix `Found` panel typing errors since the incremental search reacted with each character as I entered the string. The visual feedback of the tree was also interesting since an experienced user who knows where results should appear in a dataset may be surprised to find data in other regions, prompting further investigation. TJ1-contest has an easy to use interface which does not restrict input and promotes investigations with large datasets since it is scalable with no noticeable decrease in performance.

  For example, I performed the following steps:

  1. I loaded the Latin tree $\mathsf{animalia}_{\mathsf{A}}$.
  2. I intended to search for *spirurida* and I knew that *spirurida* is a type of *nemata* from my investigations as a novice roundworm researcher. I was interested in seeing the hierarchy around *spirurida*.
  3. Incorrectly, I entered *spirul**l**ida* in the `Found` box.
  4. I grew the results from the `Found` panel and noticed that the wrong section grew and no species of *nemata* appeared, as in Figure A.29.
  5. I read what was typed into the search box, realized the mistake, and corrected it

  * The unexpected results for found nodes did not grow the expected subtree. This might be the first indication that something was wrong

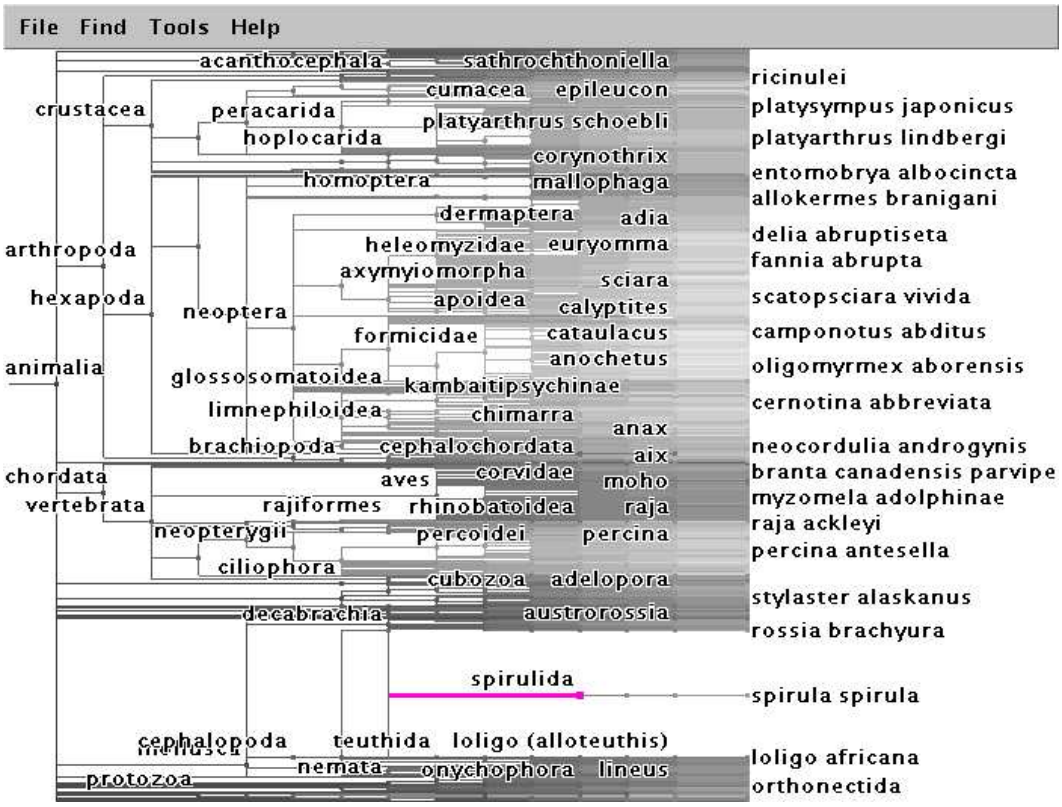

Figure A.29: After making a typographic error, the *spirulida* subtree expansion in animalia$_A$, with Latin names. Since the result was not a *nemata* classified species, this was sufficient feedback for me, a novice roundworm researcher, to conclude that I either made an input error or *spirurida* was not in the dataset.

if the search results were not looked at carefully; for such a minor difference, this might happen frequently.

* TJ1-contest did not store the rank as an attribute so determining if both names had the same rank was not possible and probably would not have helped with this task. Rank in the general case would be hard to address.

* The found node was not in the expected topology of the entire classification tree, which was an indication of user error or at least a warning to examine either the search results or the dataset contents.

- **Application specific tasks section on file system trees**

  This section deals with the tasks related to comparisons of two full logs_A and logs_B datasets as well as other comparison tasks with all four hcil_A, hcil_B, hcil_C and hcil_D datasets in the hcil subtree of the logs datasets. Four-way comparisons are not done with the logs datasets since they were too large to evaluate with TJ1-contest.

  - Where are the big directories?
  - Can you see different patterns in those files?

    These tasks are general visualization questions that TJ1-contest is well able to display. I had immediate feedback for locating the largest directories and was shown a general pattern of personal, project and course-based web pages.

    First, I loaded the dataset for tree logs_A. The root of the file system is called /// since all of the examples required fully qualified names: / is the name of the root directory and / is arbitrarily used as a separator.

    * Big directories were immediately visible from the layout since the vertical space consumed by directories indicated how many total leaves are in the subdirectory structure.
    * I found in logs_A, shown in Figure A.30, that *users* and *class* were the biggest directories linked to the root of the tree.
    * Finding the biggest directory in any subtree was done in this way, as long as no ancestor nodes of the subtree were previously grown or shrunk in navigations. If necessary, all marked nodes can be expanded at the same time to preserve marked ratios.
    * The leaves/files are right-aligned that means the leaves for interior nodes, which are the high-level directories containing subdirectories, are interspersed between the non-leaf children of the node. This made

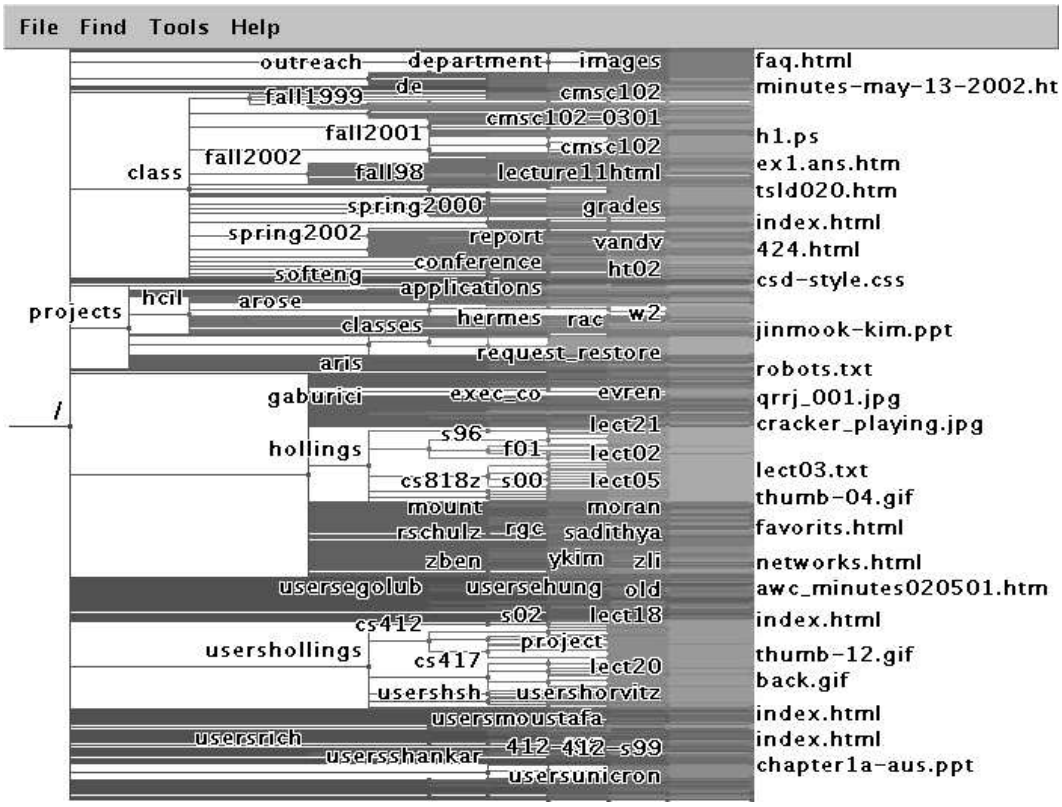

Figure A.30: The logs_A file system tree: *users* and *class* were the biggest directories linked to the root of the tree. The *users* directory is not labeled in this figure, but it is the node that is surrounded above and below by whitespace, indicating that it is large. The directory *usershollings* is the third largest. This method of finding large subtrees works only if there are a few large subtrees and no navigations were made.

accurate estimations of the number of immediate files in higher level directories impossible.

* I found personal pages in two locations: in the *users* subdirectory such as ///*users*/*hollings* and each user also had a personal subdirectory directly attached to the root such as ///*usershollings*. These directories might be symbolically linked to each other.

* The contents of the two personal directories were different. For ex-

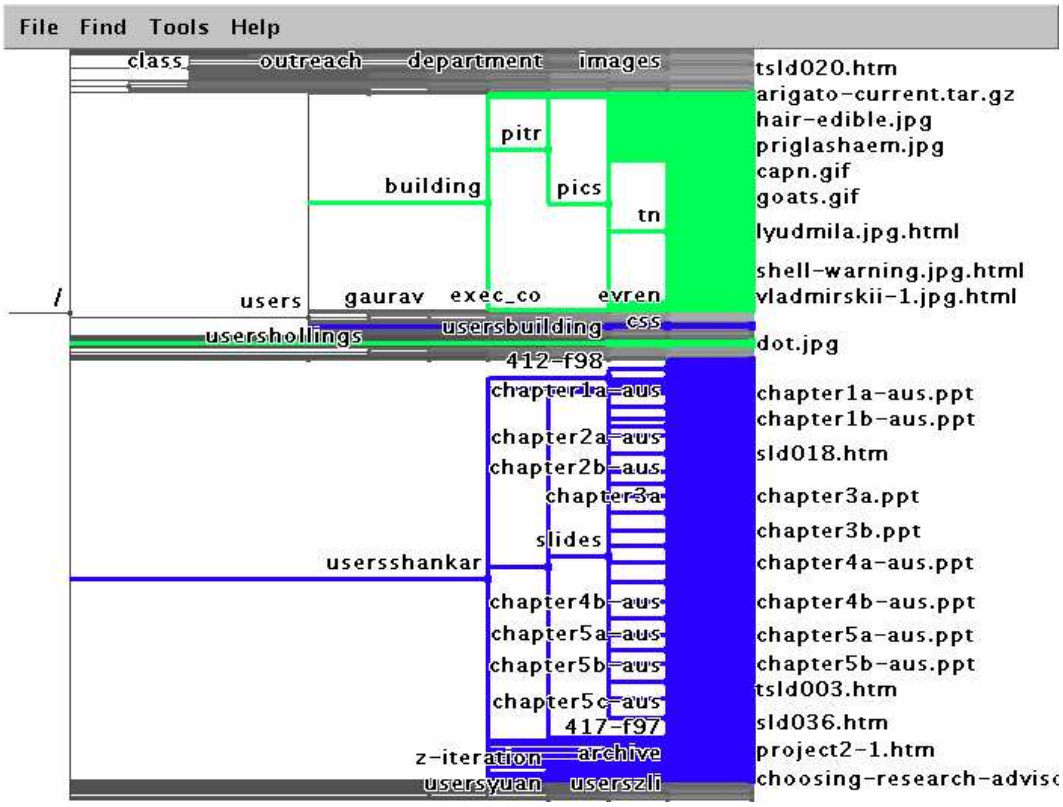

Figure A.31: The large differences in the number of files in each directory are shown, with ///*users*/*building* and ///*usersbuilding* marked in green, ///*users*/*shankar* and ///*usersshankar* marked in blue. Each directory is grown at a rate proportional to the leaves so the marked regions are still comparable to each other.

ample: ///*usersshankar* had more leaves than ///*users*/*shankar* but ///*users*/*building* had more leaves than ///*usersbuilding*; not much can be said about why the directory structure was set up this way without referring to attributes. Figure A.31 shows the large differences in the number of files in each directory, *building* marked in green, *shankar* marked in blue. Each directory is grown at a rate proportional to the leaves so the marked regions are still comparable to each other.

* The personal pages comprised of more than half of the total number

of leaf nodes in the system.

  * Of the 76547 nodes, personal pages made up 42877 nodes: 20480 of which were in the ///*users*/⟨*username*⟩ type personal pages and 22397 in the ///*users*⟨*username*⟩ type personal pages.
  * The size totals for the user directories are displayed by the `Found` panel but there were too many to display on the visualization to be useful.
  * Class pages were found in the *class* subtree which broke the years 1997–2003 into *fall*, *spring* and *summer* terms, such as *fall2002*, each of which contained *cmsc* course pages. Figure A.32 shows the *class* directory expanded to show the contents.
  * There were many fewer research pages, under ///*projects*, than there were personal or class pages. Figure A.33 shows *project* expanded to show the contents.
  * The largest directory in ///*projects* was *hcil*. This subtree will be examined later in the four-way comparisons.

– Are the newer directories bigger than the older projects?

– When was the page giving directions to the department last updated?

Although I did not use the attributes provided, the datasets were known to be weekly snapshots of a web-site, so I determined age characteristics using TJ1-contest comparisons to locate changes made to the file system. The datasets were too large to do four-way comparisons with the entire set, so these tasks were attempted with $\mathsf{logs_A}$ and $\mathsf{logs_B}$.

I loaded trees $\mathsf{logs_A}$ and $\mathsf{logs_B}$ to investigate differences in the *projects* directory as well as the other main directories with differences:

  * TJ1-contest was not able to determine the age of a directory unless the directory had been added between the times which data was

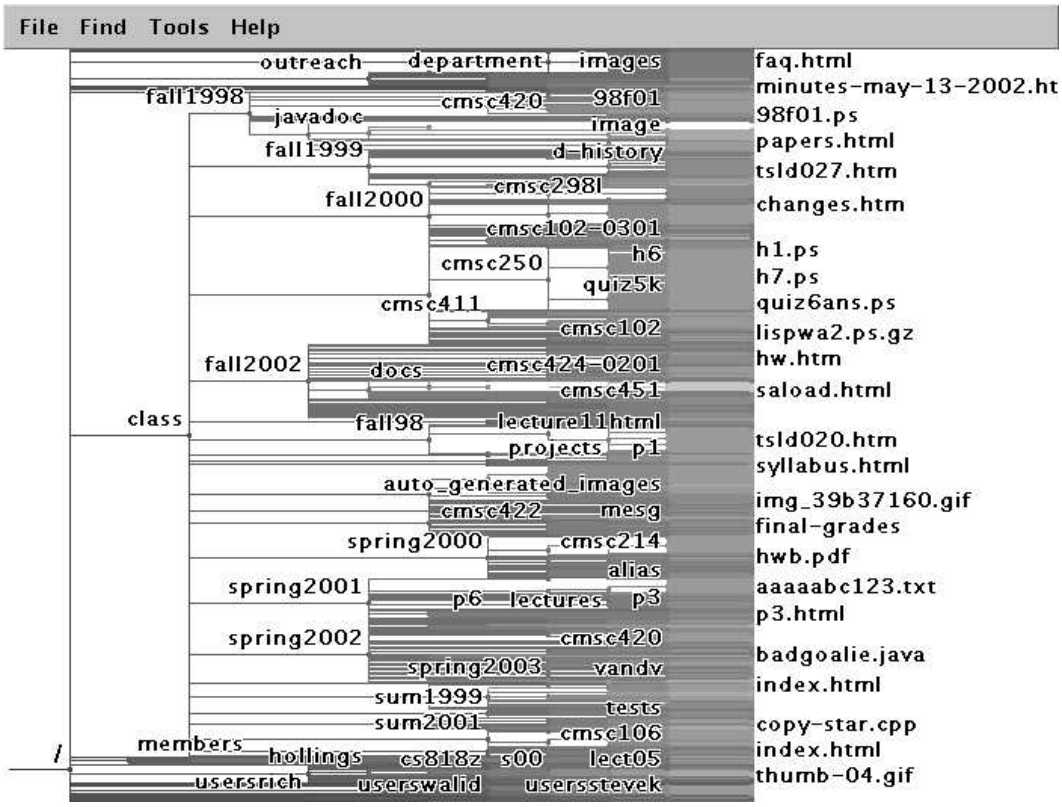

Figure A.32: *class* subtree expanded in logs_A to show details. This subtree contains a directory for each school term and the term directories contain course directories for each class.

collected. This is a restriction from the lack of attribute handling in TJ1-contest.

* The size, in total number of files, of the *projects* subtree was quite a bit smaller than the *users* directory; user *hollings* had about as many files as the entire *projects* directory. Using the `Found` panel, ///*users*/*hollings* had 7194 nodes, both leaves and internal, and ///*projects* had 8447 nodes.

* Finding the page giving directions to the department could not be done with TJ1-contest since this would have required an attribute describing the file contents. If the name of the file was provided,

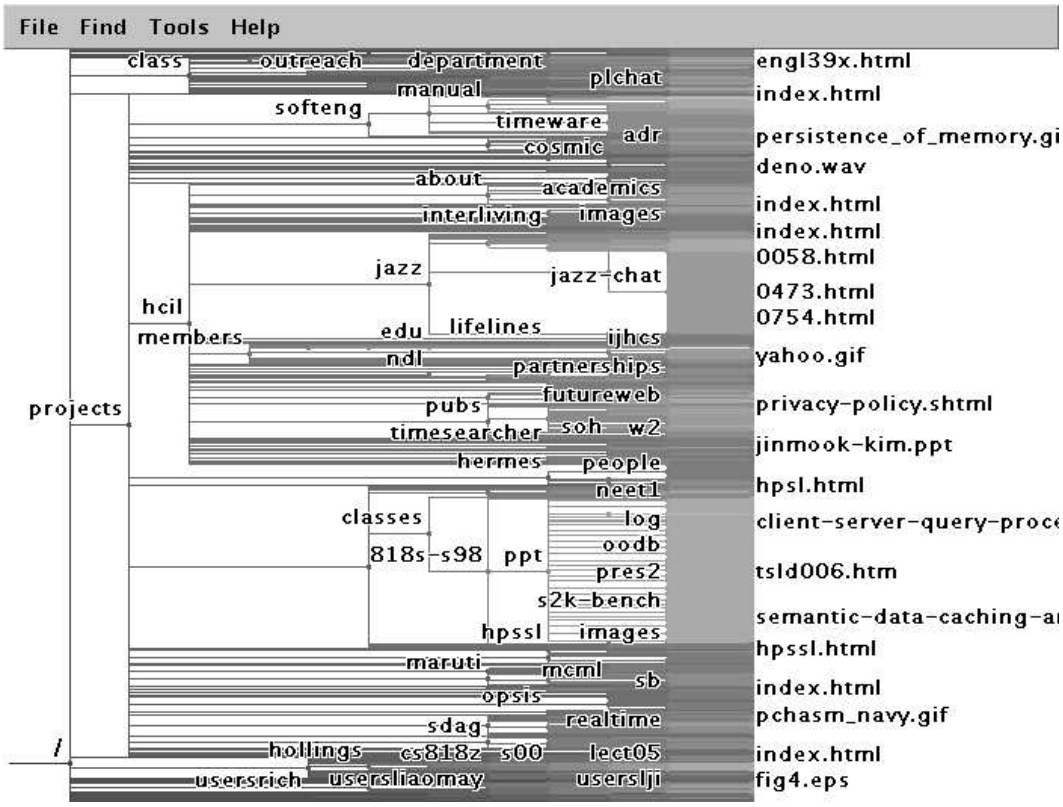

Figure A.33: *project* subtree expanded in $\mathsf{logs_A}$ to show the details. This subtree contains a directory for each project.

TJ1-contest would have been quite able to find the file.

* Personal pages showed the most diverse and sporadic differences. There appeared to be many people who added, deleted, or moved files in their personal directories, as shown in Figure A.34, an expanded view of the *users* directory.

* Class pages showed small amounts of difference, which was expected since these file system snapshots were made in summer months when most classes are not in session. Also, since classes are sorted into school terms in this file system, there are many dormant classes that are not modified several years after they have been completed. There-

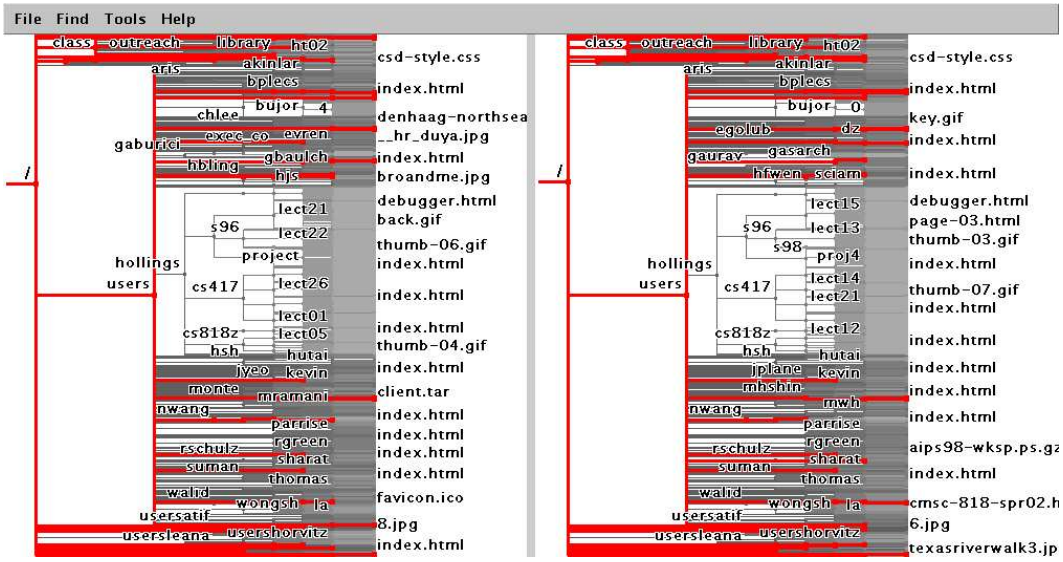

Figure A.34: *users* subtree expanded in comparison of logs_A and logs_B. There appear to be many people who added, deleted, or moved files in their personal directories in this one week time period. Differences are also seen in the context, indicating that other file system changes were also made in this time.

fore, the only differences in the class pages were between leaves in *fall2002* and *spring2003* subdirectories

* Closer examination of the *fall2002* differences showed that some files were deleted in the *projects* directory of *cmsc434-0101*, as shown by Figure A.35.
* Examination of the changes in *spring2003* showed that *cmsc838p* had changed. Those changes were: one delete, *design/openimpl.pdf*, and several additions in multiple subdirectories, shown in Figure A.36.
* *spring2003* had several additional subdirectories, possibly reflecting these courses beginning. Shown in Figure A.37, these courses included: *cmsc102*, *cmsc106*, *cmsc412-201*, *cmsc417*, *cmsc433* and *cmsc733*. The *cmsc434* directory had been further populated as well.

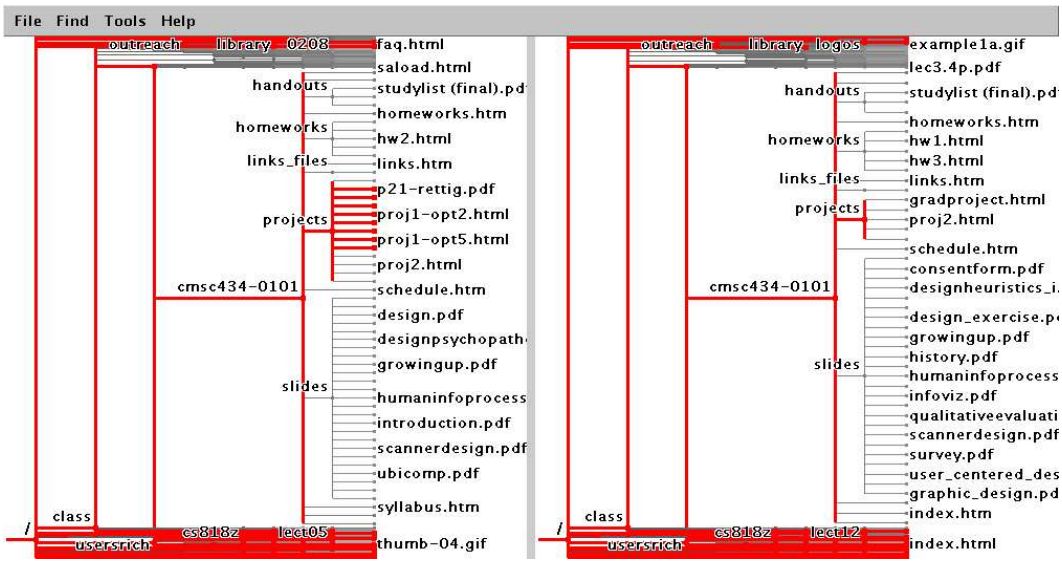

Figure A.35: Differences in *cmsc434-0101* between logs_A and logs_B show that some files were deleted in the *projects* directory. Judging by the names of these files, it seems like these are several project options that a professor might give his students but without context, I can not say for certain that this is the case.

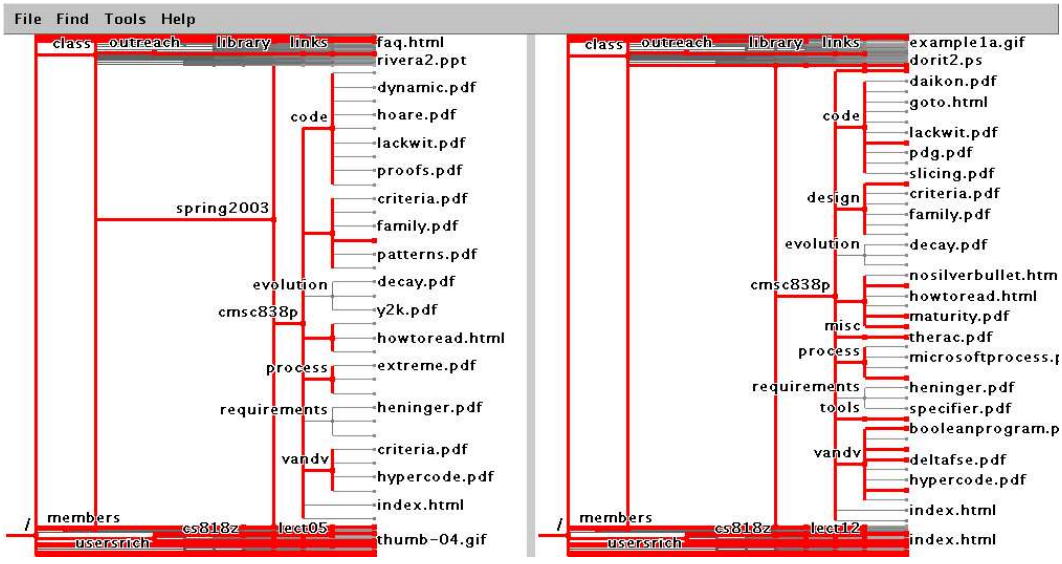

Figure A.36: Differences in *cmsc838p* between logs_A and logs_B. The only deletion was *design/openimpl.pdf*, not labeled but shown as the red marked expanded leaf on logs_A, and there were several additions; differences are shown as red, as usual.

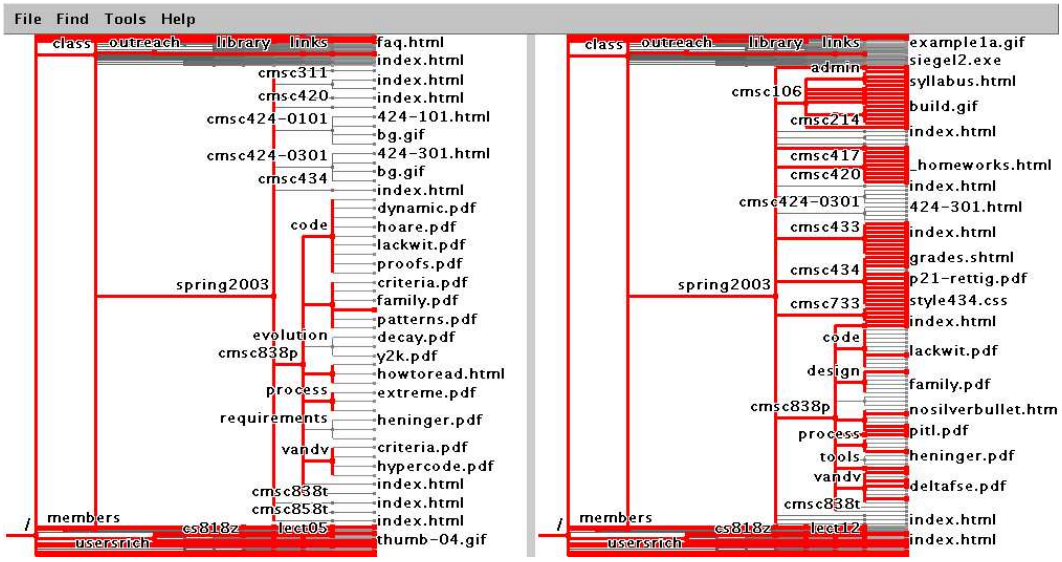

Figure A.37: Differences show new courses added between logs_A and logs_B. *spring2003* had several additional subdirectories, possibly reflecting these courses beginning: *cmsc102*, *cmsc106*, *cmsc412-201*, *cmsc417*, *cmsc433* and *cmsc733*. Also, the *cmsc434* directory had been further populated.

  - ∗ There were very few changes in the *projects* pages in this time period. The only leaf modifications were in the *jazz-chat* directory, where some files had been added that look like log files, shown in Figure A.38. These changes rippled up the tree to the root; the ripples did not reflect the entire structure changing but were useful in locating the path from the root to the differences.

– Additionally, examination of the hcil subtree was done with all four logs loaded: a four-way tree comparison. In this comparison, each node is assigned a best corresponding node with a node on every tree.

  This task was not a part of the contest tasks, but was interesting enough to mention since it shows that TJ1-contest is not limited to pairwise comparison tasks.

  I loaded trees hcil_A, hcil_B, hcil_C and hcil_D:

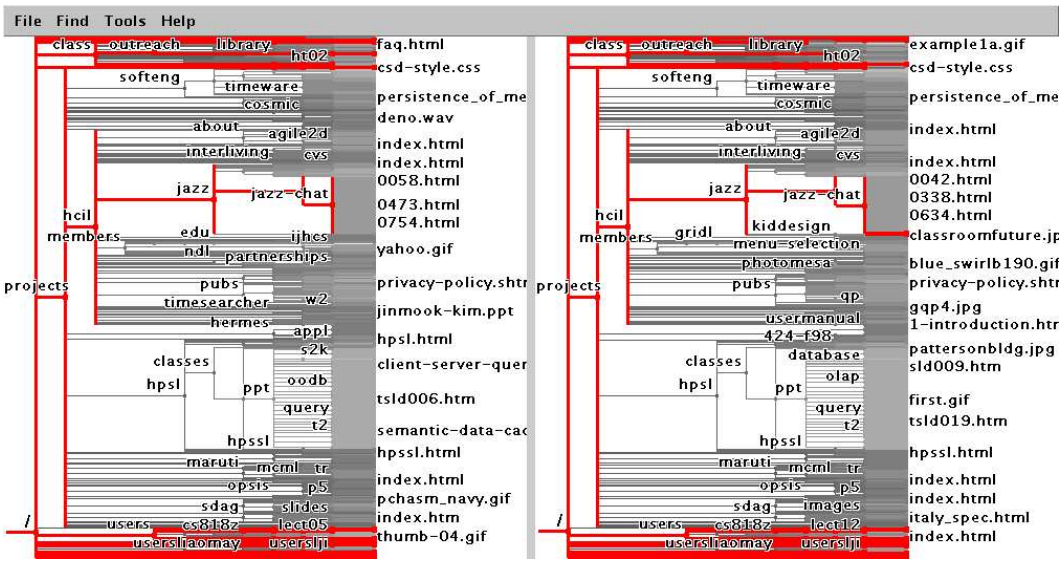

Figure A.38: Differences in *jazz-chat* directory between logs_A and logs_B. These changes rippled up the tree to the root; the ripples did not reflect the entire structure changing but were useful in locating the path from the root to the differences.

* In Figure A.39, growing the *counterpoint* directory, it was clear that the directory changes only between hcil_C and hcil_D.
* The *iv03contest* directory shown expanded in Figure A.40 was added between hcil_B and hcil_C; between hcil_C and hcil_D, the directory was further populated with contest information and all of the contest datasets, except hcil_D, of course.
* In Figure A.41, *spacetree* and *timesearcher* also showed some additions between hcil_B and hcil_C.

A.4.2 Tasks not suited for TJ1-contest

In this section of the results, I present the details of contest tasks that I did not solve with TJ1-contest. Most of these tasks were not possible since TJ1-contest did not handle attributes. Notice how this section is quite a bit smaller than the previous section; no tasks are hidden, TJ1-contest was able to solve most problems in the

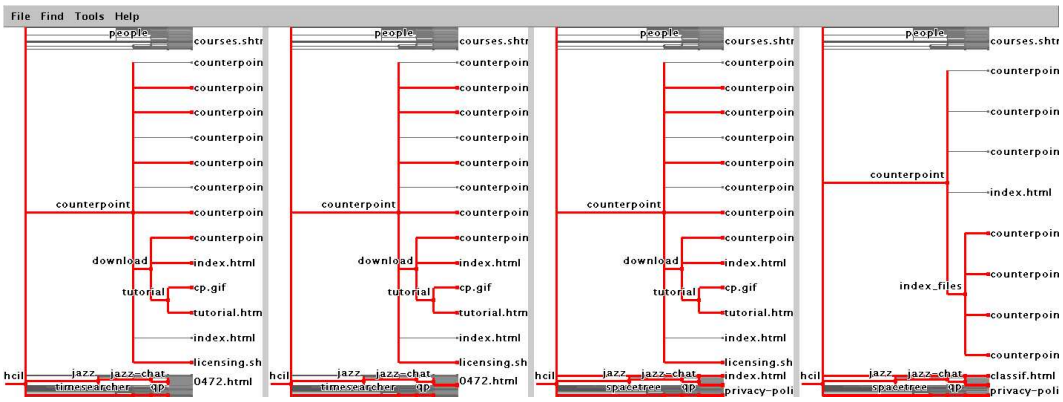

Figure A.39: Differences in *counterpoint* among trees hcil_A, hcil_B, hcil_C and hcil_D. It was clear that the directory changes only between hcil_C and hcil_D.

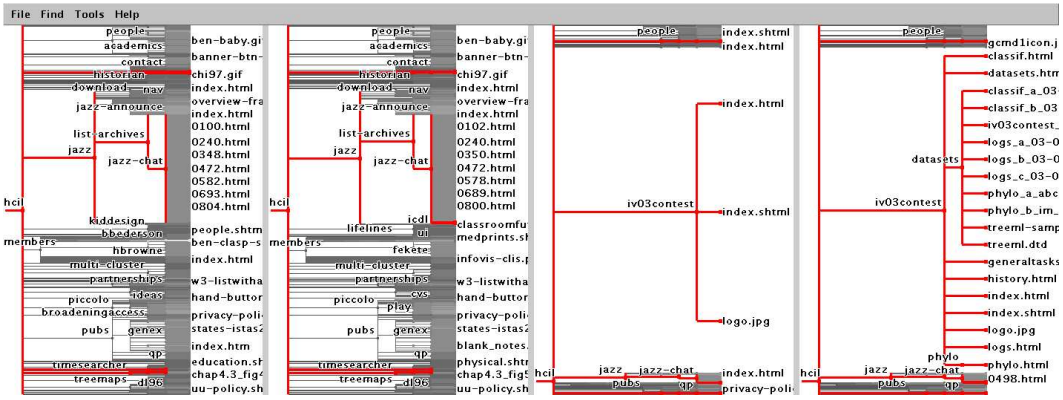

Figure A.40: Differences in *iv03contest* among trees hcil_A, hcil_B, hcil_C and hcil_D. The directory was added between hcil_B and hcil_C. Between hcil_C and hcil_D, the directory was further populated with contest information and all of the contest datasets, except hcil_D.

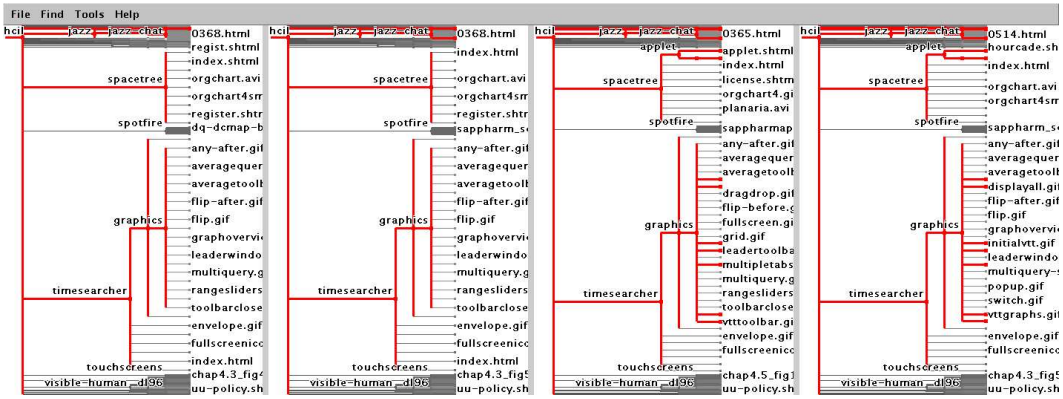

Figure A.41: Differences in *spacetree* and *timesearcher* among all hcil trees. *spacetree* and *timesearcher* show some additions between hcil_B and hcil_C.

contest using our information visualization approaches.

- **Comparison of trees for attribute value changes**

  This section deals with visualization of attributes of datasets. TJ1-contest was not able to use the attributes except for the name of each node, so this section was mentioned but dismissed as a weakness for our application.

  - Global impression: did attributes change a lot or not?
  - What nodes or subtrees changed the most?
  - Did the value of attribute XYZ for this node increase or decrease? In absolute terms, or relatively to other siblings or other nodes
    - TJ1-contest was not able to handle attributes for the contest. Additional work on parsing and handling extra attributes is interesting and may be part of future work for TreeJuxtaposer beyond TJ2.

- **General visualization of tree topology**

  This section relies on the visualization of datasets to determine certain properties of trees. There are several features that TJ2 does not support, since they are outside our domain of interest in visualization, but the features are

also possible additions to tools outside the scope of TJ2 that would use a TreeJuxtaposer API to communicate with the visualization components.

- What is the deepest branch? Does depth between subtrees vary?

  The deepest branch task is quite simple: it is a single number that TJ1-contest calculates but does not display. Since we are only interested in visualization and navigation of tree datasets, I focus on that aspect of analysis.

  * Since the tree was right aligned, the deepest path and depth between subtrees was not possible to determine visually. Determining the depth difference between subtrees is possible using the dimming values of trees—the deeper the branch the dimmer the node—but this is not an accurate metric since there may be many gradients of dimmed nodes. Determining the difference between two dimmed values is impossible, but large scale estimations are handy, so dimming does tell me where regions of deep nodes are in the dataset.

- Filtering by level: show only top n levels or remove bottom n levels

  This task is also unsupported by TJ1-contest and although filtering is a part of many visualization packages, we did not implement filtering. This task would be simple to implement with sliders to control the depth of filtering, but that complication is better left to future work.

  * No such filtering is available in TJ1-contest, but it may become part of TreeJuxtaposer in the future.

- **General visualization of tree attributes that can be aggregated**

This section of results is focused on techniques of understanding tree attributes with general datasets. Since TJ1-contest could not answer many attribute related questions, this section was also mostly dismissed as possible future work. These questions could be solved with single tree visualizations and

require neither tree comparisons nor specific tree datasets, but were simply not of high enough importance in our visualization research at the time.

  - Find high values of a numerical attribute
  - Find a given value of a numerical attribute
  - Find nodes with a certain categorical attribute value
  - Find values of a categorical attribute that occurs more often
  - Find nodes with certain values of two or more attributes

    These tasks are mostly quantitative and could be done with more sliders, similar to the filtering approaches in the previous task. Again, this would add complications and is better left to future work to keep TJ1-contest simple.

    * TJ1-contest was unable to assess attributes of nodes in this way. Additional search features could be added to assist in performing these tasks but they are not a priority to implement for our current interests.

- **General management of analysis**

  This section deals with general techniques that TJ1-contest uses for analysis. This section focuses on editing the dataset, saving views and supporting history functions. All of these techniques are considered to be future work beyond TJ2.

  - Removing special anomalies
  - Saving visualization settings for future reference
  - Keeping the history of analyses: reviewing, replaying with different parameters

These tasks are quite powerful yet they are not implemented in TJ1-contest due to the code complexity and time constraints. These tasks are mentioned in the future work section, Section 6.1.

  * TJ1-contest could not modify the tree, and did not support saving or history. TJ1 introduced mostly an information visualization technique, accordion drawing, that relied on static structures and editing the structure would be difficult with the layout mechanisms in that system. I consider TJ2 to be slightly more adaptable for these tasks but more work is required.

- **Application specific tasks section with phylogenetic trees**

  This section deals with the tasks related to $\mathsf{phylo_A}$ and $\mathsf{phylo_B}$ datasets, constructed by evaluating genomic properties of two proteins.

  – Low level tasks: interacting with the tree matching process to solve inconsistencies that can arise, displaying the trees, showing the relationships and differences from a computed or interactively constructed mapping, and providing ways to permute links and nodes to verify hypotheses interactively

    This task is highly related to the lack of editing functionality in TJ1-contest. Modifying the dataset is not possible in TJ1-contest and these interactive editing tasks are also considered future work.

    * The difference marking was provided by the automatic best corresponding node algorithm and relies on the input dataset. The best corresponding node relationships are only calculated when a tree is loaded for the first time, which is another benefit to using only static datasets.

    * Navigating through with mouse-over highlighting and marking subtrees with user marking groups allows me to recognize further simi-

larities in the tree, but no modifications of the input technology are possible to correct the automated matching process.

- **Application specific tasks section on classification trees**

  This section deals with the tasks related to comparisons of $\mathsf{mammalia}_\mathsf{A}$ and $\mathsf{mammalia}_\mathsf{B}$ datasets as well as other visualization tasks with $\mathsf{animalia}_\mathsf{A}$ and $\mathsf{animalia}_\mathsf{B}$. Comparisons are not done with the animalia datasets since they were too large to evaluate with TJ1-contest.

  - For the top five subtrees with the most nodes are they likely to have a parent of a particular rank? Or does this happen in many ranks? Can you comment on how useful rank is?

    This question is rank specific but TJ1-contest only quantifies rank with dimming. This task is not easy to answer without filtering as well, since a subset of the data is required. Besides, the task is not well thought out: if a subtree has a large number of nodes, the parent node will contain more nodes, so the top five subtrees, where one large subtree does not contain any other large subtree, when considering node quantity are all rooted at the root of the dataset. Hence, the answer is that all five of the largest subtrees are rooted at *animalia*.

    * I am unable to comment on rank since rank is an attribute that the TJ1-contest system does not handle.

- **Application specific tasks section on file system trees**

  This section deals with investigating the attributes of individual file system trees. Attributes that are considered in this section relate to the usage of the file system, as web page hit counts, which would be interesting to visualize in a tree structure over a period of several weeks, but TJ1-contest does not handle attributes.

- Which are the popular web pages?
- Are there some labs more popular than others?
- Which areas are getting more popular? less popular?
- Are new pages more popular that old pages?
- Which old page are popular?
- What proportion of the pages are never used? seldom used?

  The file system specific tasks that I could not answer with TJ1-contest are attribute based. Again, since TJ1-contest does not handle node attributes, other than a name, these tasks are not possible with our application.

  * I can not comment on file usage since attributes, which include file usage, are not handled in TJ1-contest.

A.5 Contest conclusions

Although TJ1-contest was not able to perform all tasks suggested by the contest organizers, I was able to show that the application performed well on tasks that it was designed to solve. TJ1-contest was judged to be the best entry overall at InfoVis 2003 and the entry itself was an excellent motivation to produce many forms of publicity such as: a descriptive video, a web-page, an introductory paper, a poster and a presentation. This contest motivated many interesting additions to TJ1, such as incremental search and a more advanced user interface, which made TJ1-contest a much more powerful tool. Some of the tasks that we did not solve directly were solvable with workarounds such as marking groups to return to later instead of undo functionality. The large contest datasets were too large to load completely and still be interactive for comparisons, but modifications since then have produced much more scalable versions of TreeJuxtaposer, such as TJ2, presented in Chapter 3.