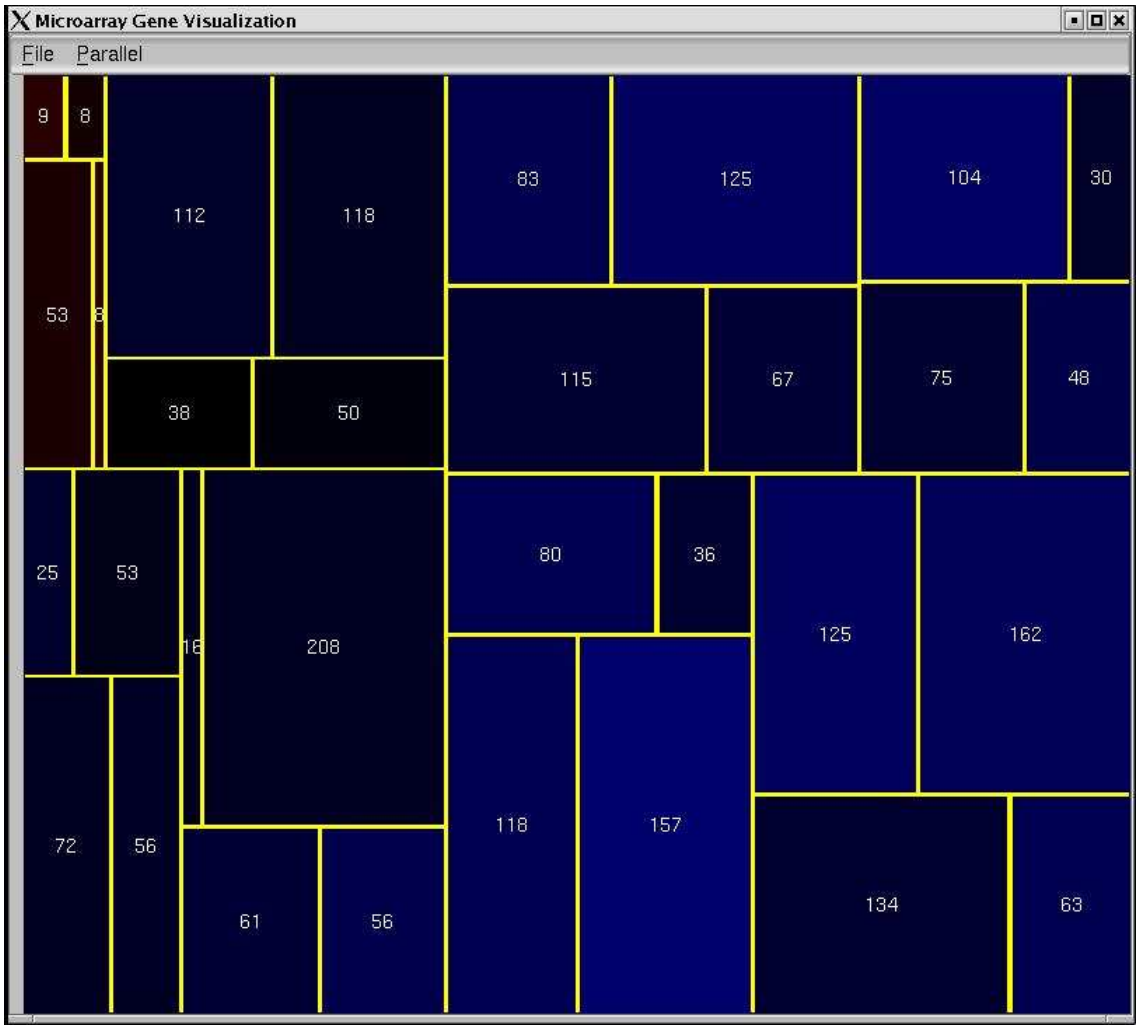
# Interactive Visualization of DNA Microarray Data

Matt T. Miller
Advisors: Dr. K.R. Subramanian
and Dr. Xintao Wu
December, 2003
University of North Carolina at Charlotte

# INTERACTIVE VISUALIZATION OF DNA MICROARRAY DATA




Matt T. Miller

Advisors: Dr. K.R. Subramanian and Dr. Xintao Wu

December, 2003

Department of Computer Science

The University of North Carolina at Charlotte

Charlotte, NC 28223-0001

# TABLE OF CONTENTS

# TABLE OF ILLUSTRATIONS

# CHAPTER 1 -- INTRODUCTION

An increasing amount of bioinformatics research is in gene relationships. A new technology, microarray data, has emerged that allows for the storage of tens of thousands of genes on a single chip. This gives researchers the ability to look at entire genomes at once. Datasets this massive require a practical way to visualize them, an efficient, intuitive way to navigate and explore them, and a way to automate the grouping process. This paper documents the work I have done for my senior project at the University of North Carolina at Charlotte building an application that visualizes large microarray dataset hierarchies as treemaps, allows efficient and intuitive navigation of those hierarchies, and visualizes gene combinations and their correlation values on a parallel coordinate graph based on the research in [6].

The microarray data hierarchy navigation tool I developed merges the treemap visualization with the actual navigational graphical user interface. The treemap is a full-color and completely interactive visualization tool. Through the use of color, labels, partitioning, and tool tip text, the treemap can visualize four separate attributes of the gene clusters that comprise it. Currently, the treemap visualizes three attributes: correlation value (color), the number of individual genes that comprise each cluster (partitioning and label), and the gene's name (tool tip text). With an additional window, called the "sidebar", navigating the

entire hierarchy at varying levels of detail is simple, intuitive, and fast. The sidebar shows all of the clusters at a certain level of detail and above, show exactly where in the enormous hierarchy a cluster is located, and its correlation value. It also provides the interface for moving to different locations and levels of detail in the hierarchy. Together, these tools allows for the rapid absorption of the massive data visualized, which allows the user to more quickly separate areas of interest from the rest of the data.

Many methods exist to show relationships between two genes (where genes are only placed in one group), but few methods exist to show relationships between multiple genes. While finding two-way relationships between genes is beneficial, many genes influence multiple physiological pathways, and hence, would belong to more than one grouping. Finding these elusive k-way relationships will help the science community better understand exactly how genes affect organisms. A professor at UNC Charlotte, Dr. Xintao Wu, et al. have developed a method to automate gene grouping, and hence, separating interesting data from the rest, using a loglinear model. The loglinear model measures correlation rather than causality, and it allows genes to belong to multiple groupings, which enables the discovery of elusive k-way gene relationships. The other aspect of this project is to take the loglinear groupings (genes and discrete expression levels) and their correlation

values, and graph them in a variation of a parallel coordinate graph. This shall aid the user in finding those k-way relationships by visualizing the gene/expression level combinations and their correlation values. This should allow for the rapid identification of potential multidimensional relationships.

# CHAPTER 2 -- BACKGROUND

With the completion of the human DNA sequence as part of the Human Genome Project, [17,18], studies of gene-gene interactions will play an increasingly important role in the search for the causes of human diseases. While individual genes may be responsible for making proteins, those proteins usually interact in different physiological processes and pathways. Clues to the function of an unknown protein can be determined by investigating its interaction with other proteins whose functionality is known.

## 2.1 – Microarray Data

### 2.1.1 – What are Microarrays?

A Science Magazine article entitled "Technologies in DNA Chips and Microarrays: II" [9] states that "DNA chips (often called biochips) and microarrays represent a broad class of technologies rather than a single technique." Another article from Science Magazine [7] describes microarrays as "ordered sets of DNA molecules." These ordered sets are of individual DNA features (usually genes), which are placed in precise locations on a substrate. The features are extracted from a sample using either messenger RNA or oligonucleotide probes. Using messenger RNA produces the complimentary DNA (cDNA) of a sample, so the extracted sequence of features must be complimented again in order to store the original sequence in the microarray. Using oligonucleotides synthesizes

the sequence in one go. Microarrays are a refinement of their immediate predecessor, macroarrays. Microarrays differ from macroarrays only in storage density (the number of spots on the substrate). On microarrays, this density is so great (spots are usually less than 200 microns in diameter) that thousands (currently up to 40,000) of features can be fit onto a single chip (usually glass, nylon, or silicon). The spots appear as colored dots (figure 2.1), whose intensity and color represent information about a specific gene from the sample. The principal benefit of this technology is the ability to study thousands of genes simultaneously, which allows us to rapidly find relationships between genes.

There are multiple methods for encoding microarrays onto a chip, which is why it is said to represent a "broad class of technologies". Some low-end systems use radioactivity or chemiluminescence to mark the chip, while most other systems use various methods of attaching fluorophores to the substrate. Although each spot in a microarray represents an individual feature (usually a single gene) of a DNA strand, the manner of how that feature is encoded into a spot is entirely up to the designer of the microarray-creating machinery. Some methods are more accurate than others, making it easier for the detection machinery to do its jobs. One such method is applying the fluorophores in three dimensions on the chip, instead of a flat layer [7].

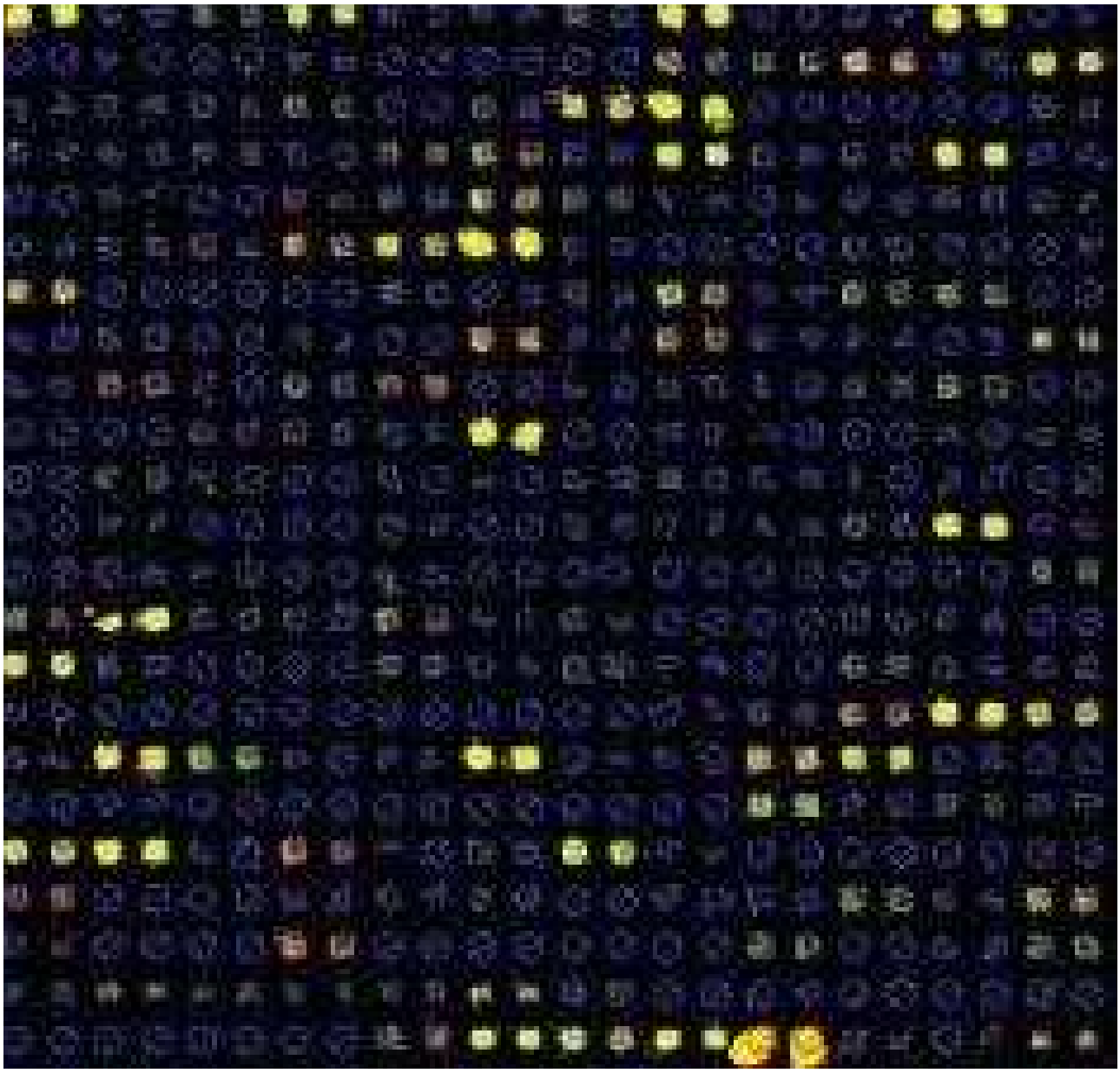


**Figure 2.1– An Example of a DNA Microarray Slide**

### 2.1.2 – Potentials of Microarray Data

Microarray data is still in its infancy, but it is already being used in research in many fields, such as genology, toxicology, drug discovery, and disease diagnosis. In fact, any area of research that concerns gene relations will benefit greatly from this technology. For example, the most immediate beneficiary is the pharmaceutical industry [10]. They seek to learn why medicines work better for some people than they do for others. Discovering the gene relationships which dictate certain behaviors in the body will allow for the development of drugs to treat genetic problems. Doctors will be able to use microarray data to diagnose genetic diseases and shortcomings in patients by comparing their gene sequences against those in various genome databases. The doctors could then prescribe the drugs the pharmaceutical industry will have developed.

### 2.1.3 – Problems with Microarray Data

One problem with microarray data is its sheer volume: the researcher is looking at tens of thousands of genes at once. Many clustering algorithms have been developed for finding gene relations amid this mass of data, such as CAST [11], MST [12], HCS [13], and CLICK [14], which group genes with similar expressions into clusters. These techniques force a gene into a single cluster, which does not take into account the fact a gene can affect multiple processes. In addition, these clustering methods do not consider relationships between genes inside a cluster

and those across different clusters, causing these methods to lack the ability to find multidimensional relationships. Such relationships show if and how genes contribute to multiple physiological pathways (such as the p53 protein [6]). This problem brings forth the need of a tool that can assist a researcher in finding these elusive k-way relationships; one goal of this project is to use loglinear modeling to achieve this. The other goal of this project is to develop an interactive graphical environment for navigating and exploring microarray data that has been grouped into hierarchical clusters in a binary fashion. Grouping the clusters further into a binary-tree-hierarchy of clusters provides a framework for viewing the dataset at varying levels of detail [2].

### 2.2 – Loglinear Modeling

The goal of [6] is to use loglinear modeling [15] to rapidly separate useful information (from the glut of microarray data) and find the multidimensional (k-way) relationships between genes. Wu, et al. define loglinear modeling as "a methodology for approximating discrete multidimensional probability distributions", and it is based on correlation measure instead of causality measure. The multidimensionality aspect overcomes the main shortcoming of traditional clustering methods: a node (in this case, a gene) can not belong to more than one unrelated cluster in a cluster hierarchy [6]. Wu, et al. transform microarray data into a boolean matrix, apply the Apriori

method to find all of the large gene sets, and then iteratively build k-way relations between the genes. One aspect of this project is to develop a tool to visualize the data generated in the method described in [6] using a parallel coordinate graph. This graph will plot gene/expression-level combinations, as described in [6] on axes measuring interaction effect (a value measuring correlation between the gene/expression-level combinations). This visualization will help a researcher quickly recognize those relationships uncovered using the method of Wu, et al. The details of this aspect of the application are described in more detail later in the paper. These expression levels are found through the following equation.

$$\hat{l}_{i1i2\cdots in} = \log \hat{y}_{i1i2\cdots in} = \sum_{G \subseteq \{d_1, d_2, \cdots, d_n\}} \gamma^{G}_{(i_r | d_r \in G)}$$

The coefficients corresponding to any group-by G are obtained by subtracting the average $\ddot{l}$ value at group-by G all the coefficients from higher-level group-by-s. For example, in a four-dimensional table with dimensions A, B, C, and D, (i,j,k,l,y[ijkl]) is used to denote a cell in four-dimensional cube-space, where I = 0…I-1, j = 0…J-1, k = 0…K-1, and l = 0…L-1 [21]. All of the possible factor effects of k, of k-1, all the way down to those of 1, and the mean $\gamma$ is shown in the following:

$$\begin{aligned} \log \hat{y}_{ijkl} \quad = \quad & \gamma + \gamma^{A}_{i} + \gamma^{B}_{j} + \gamma^{C}_{k} + \gamma^{D}_{l} \\ & + \gamma^{AB}_{ij} + \gamma^{AC}_{ik} + \gamma^{AD}_{il} + \gamma^{BC}_{jk} + \gamma^{BD}_{jl} + \gamma^{CD}_{kl} \\ & + \gamma^{ABC}_{ijk} + \gamma^{ABD}_{ijl} + \gamma^{ACD}_{ikl} + \gamma^{BCD}_{jkl} \\ & + \gamma^{ABCD}_{ijkl} \end{aligned}$$

## 2.3 – Information Visualization

Information visualization is about presenting abstract data in visual form to make it more understandable. It can be for the purpose of making data more understandable to others, making large, tedious data sets easier to study for a researcher, or making the data interactive.

There are countless methods for visualizing data, from pie charts and bar graphs to treemaps and parallel coordinate graphs. This project works with a binary tree hierarchy of clustered DNA microarray data, and gene combinations and their loglinear correlation values. Many visualization techniques have been developed to visualize clusters and hierarchies, which traditionally are based on a node-link representation. Although this is the most intuitive representation, it makes very inefficient use of display space; beyond a few-hundred nodes, the display becomes cluttered. 3D extensions to graph drawing algorithms have also been proposed, including cone trees and disc trees [19], where the children of each node are arranged on the base of a cone; in the case of disc trees, the lateral surface of the cone is dispensed with. These techniques, to some extent, provide efficient space usage, but occlusion and difficulty seeing the entire graph or hierarchy becomes an issue. To use space optimally, space-filling approaches can be used, especially for large hierarchies. Treemaps [16] use rectangles to represent clusters;

hierarchies are represented by recursive subdivision based on some attribute common among the clusters, such as the number of children.

In many domains, the incoming data has a large number of dimensions, and is said to be multi-variate; as most visualizations can accommodate at most 4 or 5 dimensions (3 spatial, time, and color). Most of these techniques only show a subset of the dimensions. Some techniques, such as parallel coordinates [20], are more scalable to higher dimensions. The multi dimensional data is mapped onto 2D plots, plotting n-dimensional points as polyline segments through N axes, all of which are parallel to each other.

### 2.3.1 – Treemaps

Developed by Ben Schneiderman, and described in [16], treemaps provide an efficient, two-dimensional space-filling approach to visualizing hierarchical structures. In treemaps, rectangles represent clusters in the hierarchy. The root cluster in the hierarchy owns all of the available space. This space is partitioned with vertical lines between that cluster's children. Each child is allotted space proportional to its size ("size" being the value of any attribute common to clusters, such as its number of children). Each child's space is subsequently partitioned in the same manner, but with a horizontal line. This process recurs down the tree, at

odd levels the partitioning lines are vertical (as with the root (level 1)), and at even levels, the partitioning lines are horizontal (as with level 2, where the root's children are partitioned). The formula for the partitioning is "x[n] = x[1] + (Size(child[n]) / (Size(root)) * (x[2] – x[1])", where x[n] is the x or y coordinate of the partition for the nth child of the root, x[1] is the minimal x or y coordinate of the root's rectangle, and x[2] is the maximum x or y coordinate of the root's rectangle. For odd levels (such as root), use x coordinates. For even levels, use y coordinates. Color coding of the rectangles provides both visual clarity and the opportunity to visualize an additional attribute of the clusters, but if the colors of adjacent rectangles are too similar, boundary lines must be used [16]. Figure 2.2 shows a colored treemap five levels down from the root. In this project we use interactive, colored treemaps to allow navigation and exploration of the clustered hierarchy of DNA microarray data.

### 2.3.2 – Parallel Coordinates

Parallel Coordinates, developed by Alfred Inselberg, visualize multi-variate/multidimensional data without losing information [3]. Parallel coordinate graphs allow plotting n dimensional points two-dimensionally on N parallel axes. Each axis represents a single dimension, and a polyline connecting the axes represents a single n dimensional point. Plotting several n dimensional points on the same set of axes allows the


**Figure 2.2 – A Colored Treemap**

viewer to quickly recognize similarities and differences between the points, thus making parallel coordinates good for finding relations in multi-variate data, converting that problem into a 2-D pattern recognition problem [3]. Similar n-dimensional shapes will have a similar parallel graph, for example, two 8-dimensional spheres will have a similar graph, which will look different than an 8-dimensional pyramid's graph. In this project we use a modification of parallel coordinates to search for k-way relationships between genes. The graph used in this project differs from Inselberg's parallel coordinates in a few key ways. This graph is, in fact, in two parts. The first part is a graph as described above. It contains m parallel axes, where m is the number of genes. Each axis has n possible places a polyline can intersect, where n is the number of expression levels a gene can take. Thus, the first part of the graph plots gene/expression level combinations. The second part differs considerably from Inselberg's. In this part of this graph, each axis represents a combination of genes instead of a single dimension. In this sense, single genes can be thought of as single dimensions. Each gene can take n expression levels (such as low, normal, and high, for n = 3). Each combination of genes and their expression levels (such as Gene A[low] + Gene B[high] + Gene C[normal]) is graphed as a single point on the axis that represents that combination of genes (for example, combination "Gene A[low] + Gene B[high] + Gene C[normal]" would be a point on the axis that graphs "Gene A[all] + Gene B[all] + Gene C[all]".

Another difference is that since each point is plotted in a self-contained point on a single axis, there are no polylines connecting multiple points. When a point is selected, its combination is plotted on the first part of the graph (which is like Inselberg's).

## 2.4 – Why Microarray Visualization?

In [6], microarray data is described as providing "a powerful basis for analysis of gene expression." Studies on gene-gene interactions will play an increasingly important role in the search for the causes of human disease, however, microarray data can bombard a researcher with tens of thousands of genes simultaneously. An application that presents an intuitive, interactive, and graphical navigation tool for such microarrays can aid a researcher in rapidly finding areas of interest within the massive dataset. The application developed as part of this project contains such a navigation tool using colored treemaps as part of a point-and-click interface. The loglinear modeling aspect of the application is designed to aid the researcher in finding multidimensional relationships between genes undetectable with other methods.

# CHAPTER 3 – EXPERIMENTATION AND RESULTS

## 3.1 – Treemap Visualization

The user interface has undergone grueling testing and has proven to be an intuitive, efficient, and effective way to navigate and explore even a large hierarchy (over 2000 clusters). The visualization aspect of the interface successfully allows the user to rapidly determine the correlation values of the clusters displayed, find out information about those clusters (such as which genes comprise them and the cluster's exact location in the hierarchy), and take a closer look at those clusters (by exploring the subhierarchies underneath those clusters). This navigation tool works consistently well under different visualization demands, from very small subhierarchies (figure 3.1 shows the smallest of all subhierarchies), to very large subhierarchies (figure 3.2 shows 8 levels down from the root node). This part of the visualization was tested using two different data files. The first one, "test.gtr", is very short (15 clusters and 16 genes), while the second one, "demo.gtr", is very long (2466 clusters and 2465 genes). The small file was used to debug and perfect the algorithms (figure 3.3 shows a treemap visualization using the small data file), and the second file was used to test these algorithms with a more realistically-sized data set. The second file provided the data for figures 2.2, 3.1, 3.2, and others of the treemap and sidebar, other than figure 3.3. The application, after debugging and perfecting, works well, and is stable with the large data set as well as the small.

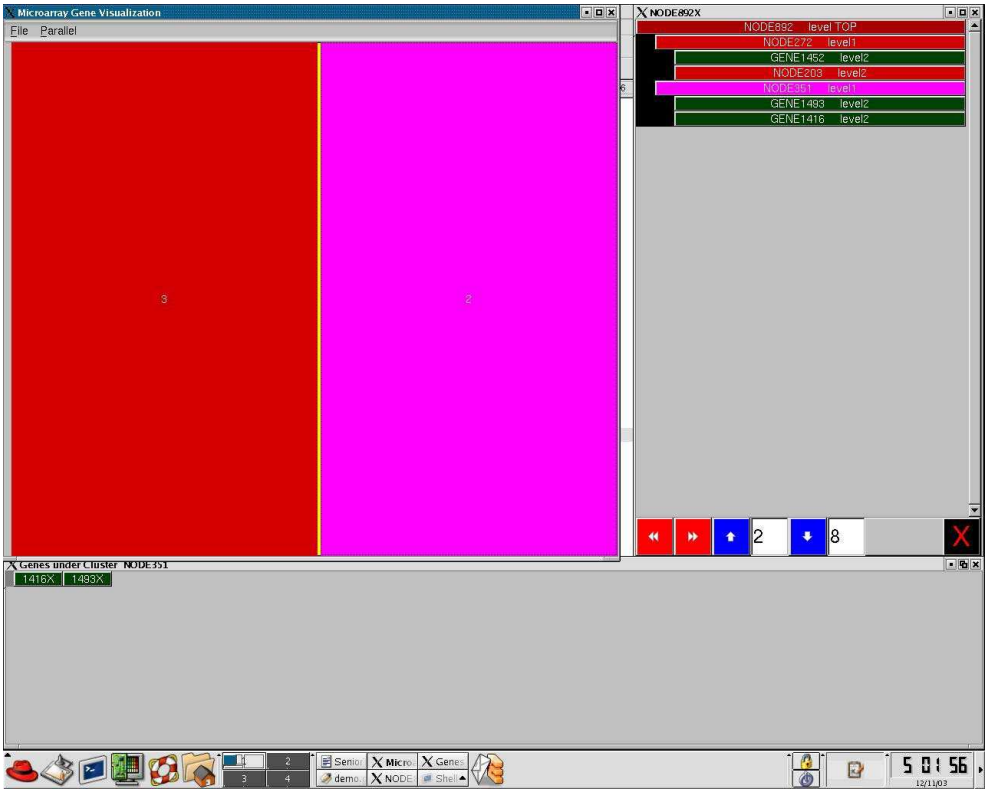



**Figure 3.1 – An Example of the Smallest Subhierarchy: a Cluster at the End of the Overall Hierarchy**

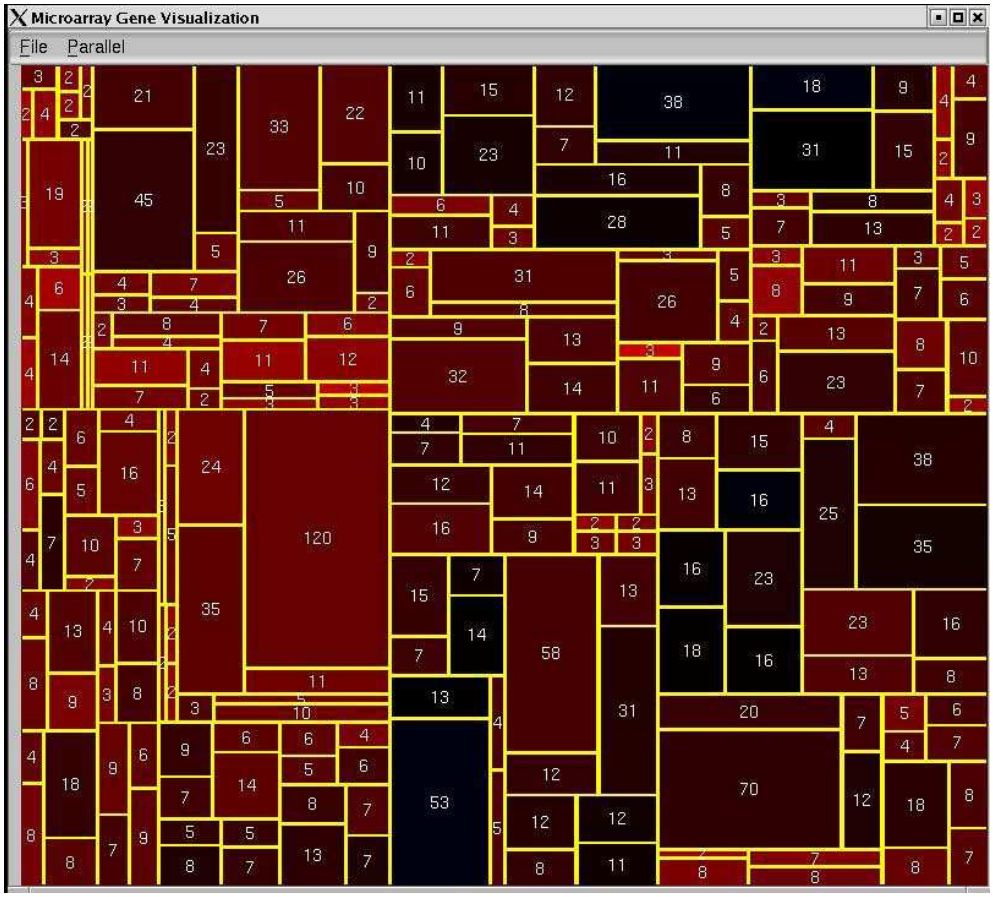



**Figure 3.2 – 8 Levels deep from the root. This shows how cluttered the treemap can be.**

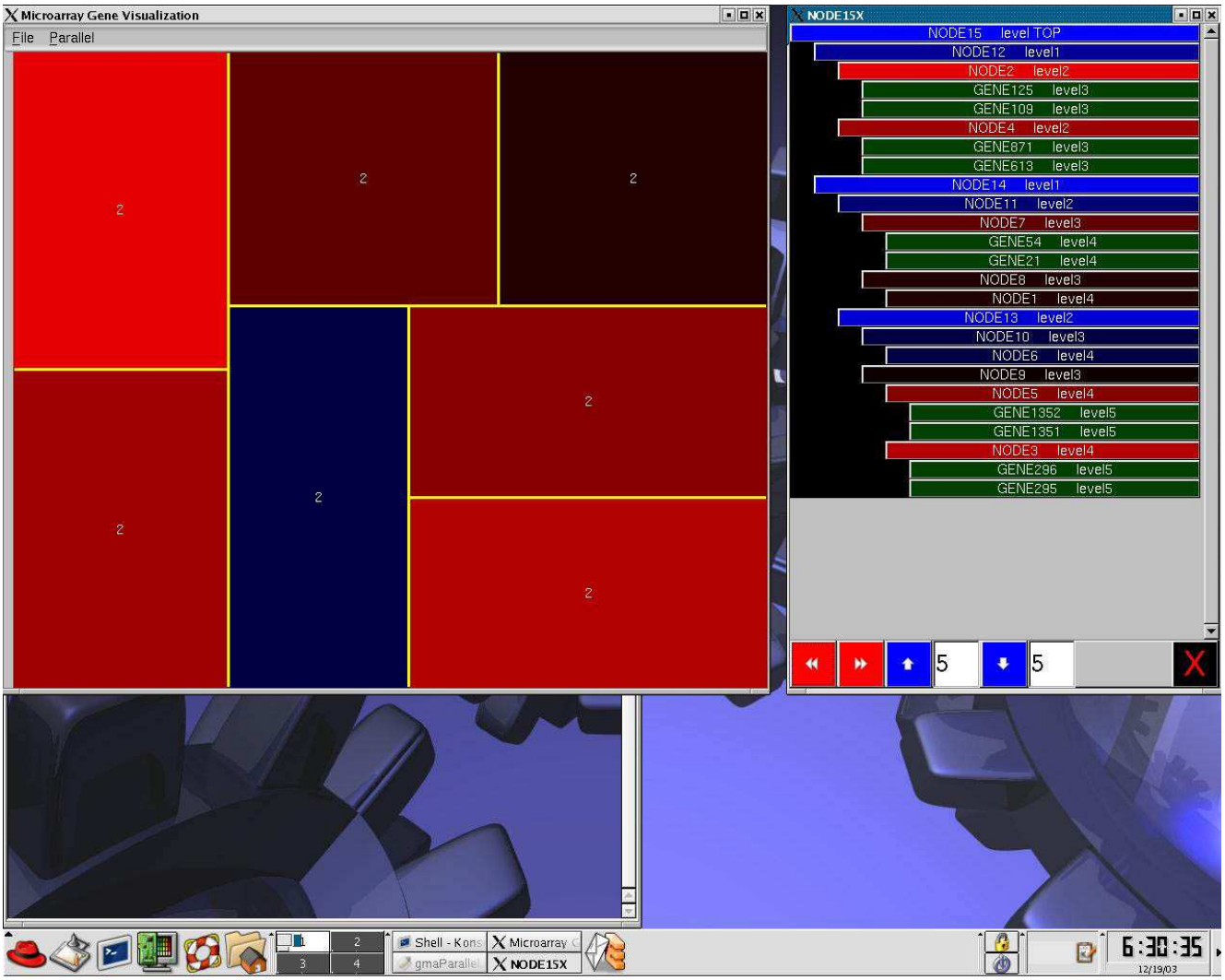



**Figure 3.3 – A Treemap Using the Smaller Data File**

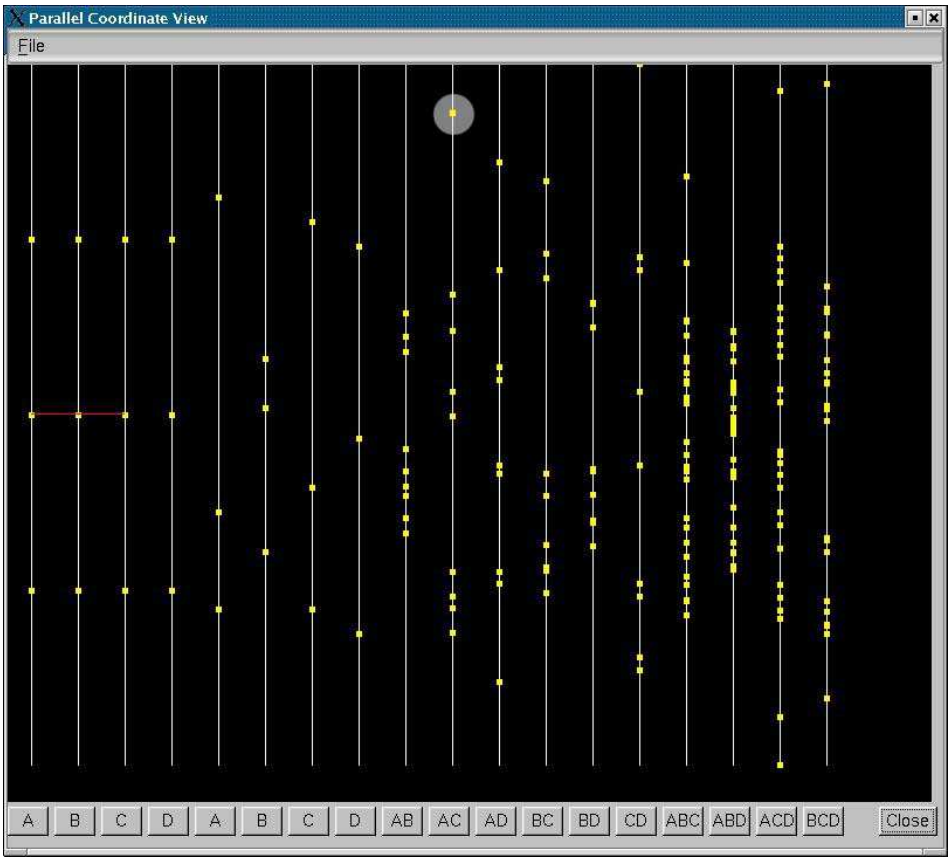
Dr. Subramanian has said that some data sets may get as large as 8000 clusters and a similar number of genes, and the human genome is estimated to be between 30,000 and 45,000 genes. Since this application works well with the large data set it was tested with, there is no doubt it will work just as well with an even larger data set (system memory permitting).

### 3.2 –Parallel Coordinate Loglinear Graph

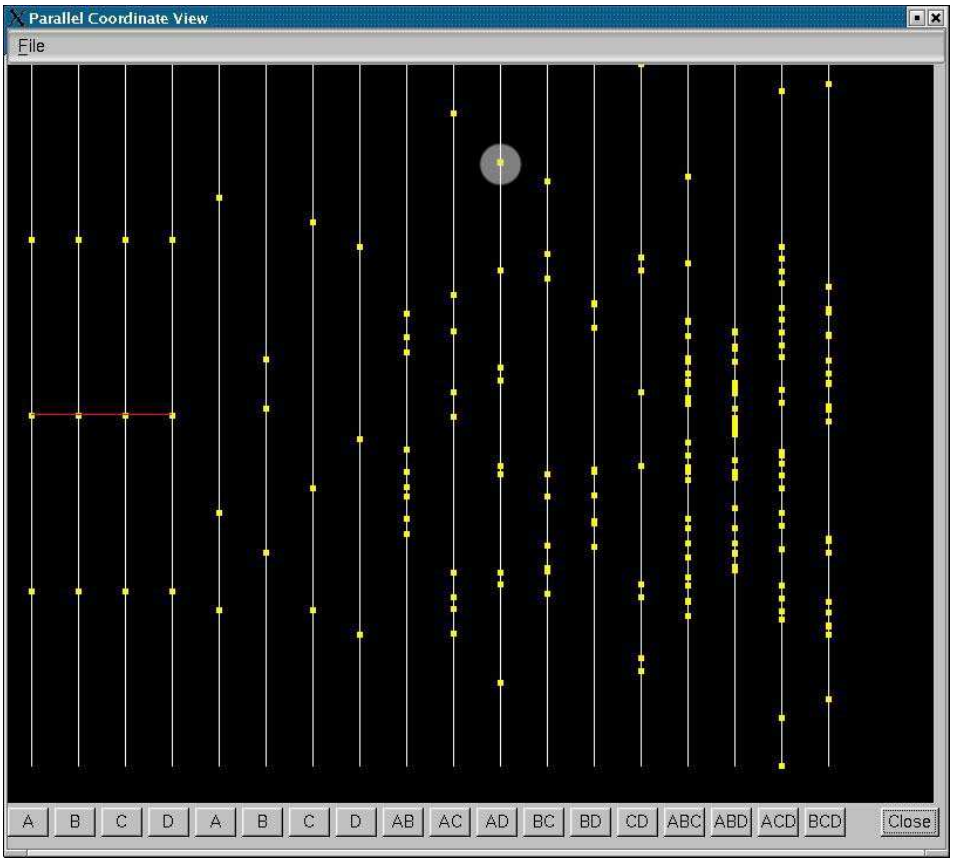
The parallel coordinate graph is the newest feature of the application, having been started less than a month before this paper was written. For having such a short development cycle, it is surprisingly functional, effective and stable. Currently, it can read loglinear data files comprised of 4 genes with 3 expression levels, 2 genes with 3 expression levels, and 3 genes with 2 expression levels. It has been tested with a data file for each of those combinations. The smaller data files (with 2 and 3 genes) can not yield much useful data, but they were instrumental in the rapid testing and debugging phase. The larger data file (with 4 genes) has more potential to uncover elusive k-way gene relationships. One method of finding these relationships is to match similar gene/expression-level combinations with similar correlation values. This process would be highly impractical and tiresome if done by manually reading a data file. This application allows the user to immediately identify combinations with similar correlation values, and, upon picking a combination (or

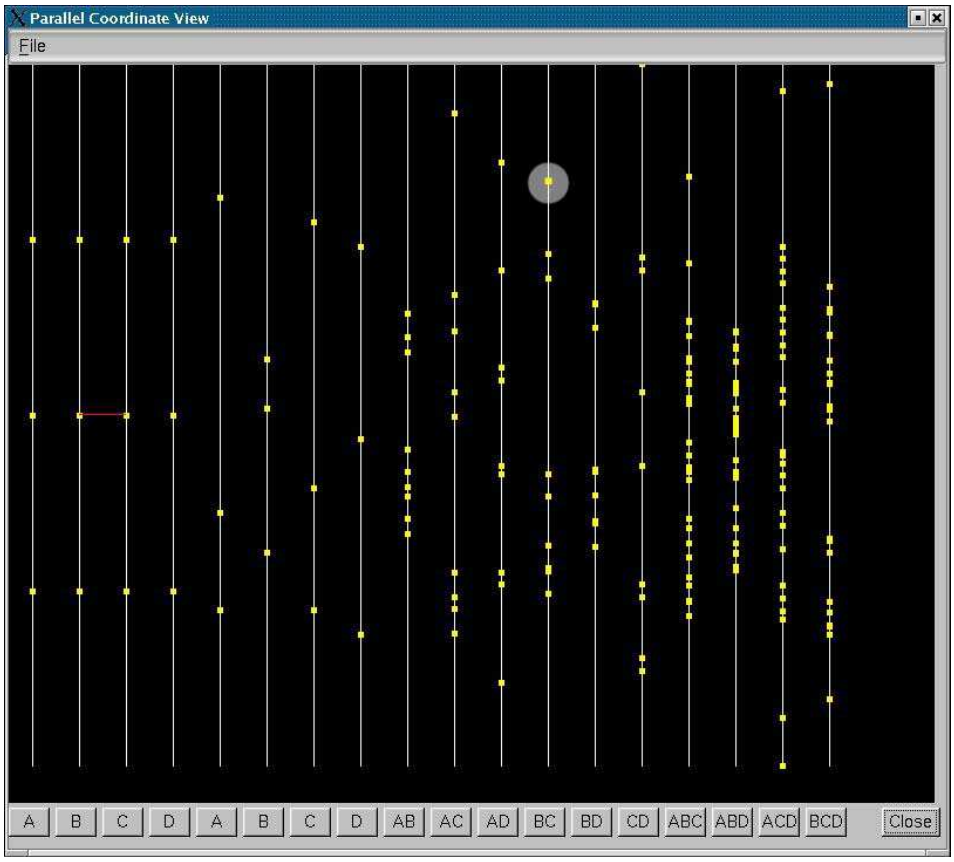
several nearby combinations), that/those combination's gene/expression-level composition is graphed, allowing the user to instantly recognize trends.

Using the larger data file (with 4 genes), a trend has been recognized among combinations that possess similar, high correlation values. When genes A, B, C, and D are all at normal expression level, any combination of them, aside from combinations possessing both genes A and B together, comprise all of the highest correlation values in the dataset, with the combination "gene C[normal] + gene D[normal]" yielding the very highest correlation. Figures 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9 show this trend. Note that these figures have been modified: highlighting was added to the chosen plotted square to show exactly which combination was selected from the axis identified in the caption. It is a safe assumption that more than just this one trend exists in the 4 gene, 3 expression level data set. When even larger data files are available, such as those with 5 or more genes, it is doubtless that this tool will assist in the rapid discovery of many k-way relationships.

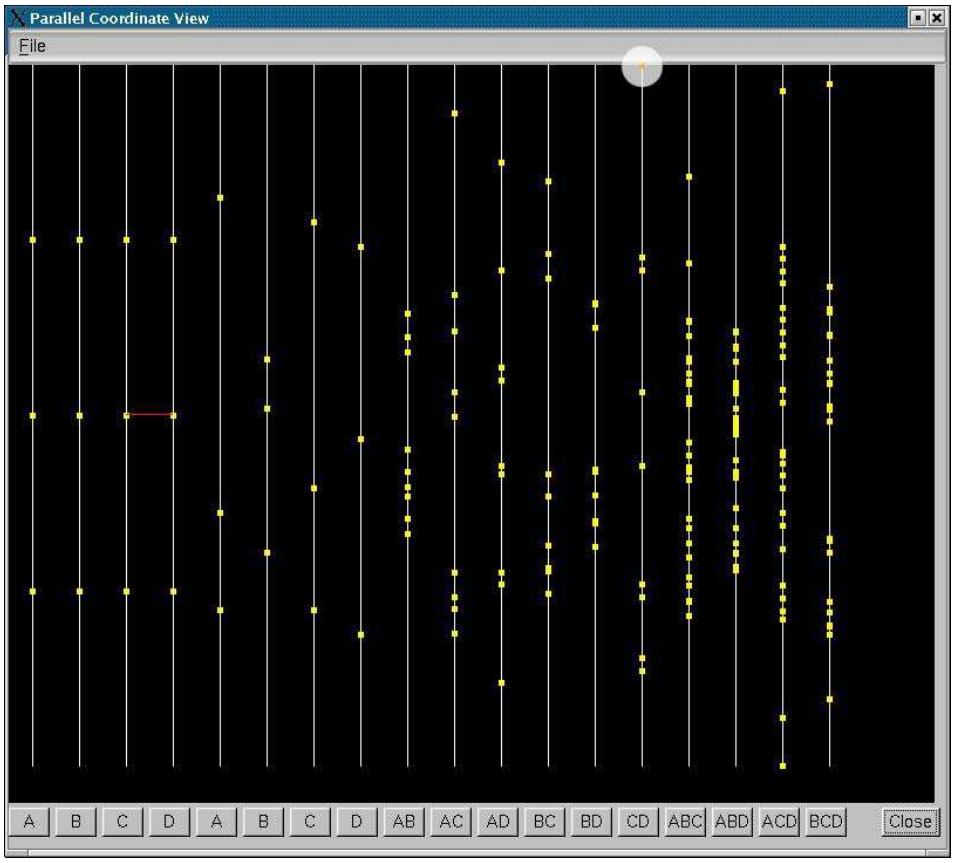


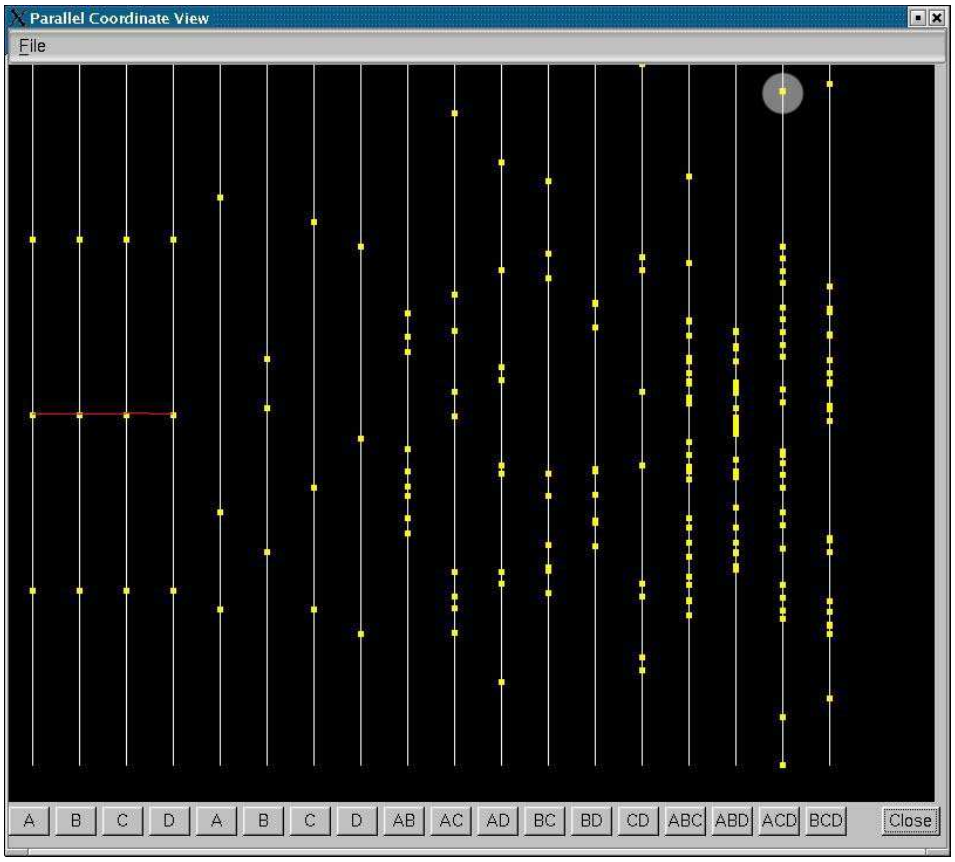

**Figures 3.4 (above) and 3.5 (below) show how these two Similar Gene /Expression-Level Combinations have Similar Correlation Values. They are also Similar to those in Figures 3.6, 3.7, 3.8, and 3.9.**



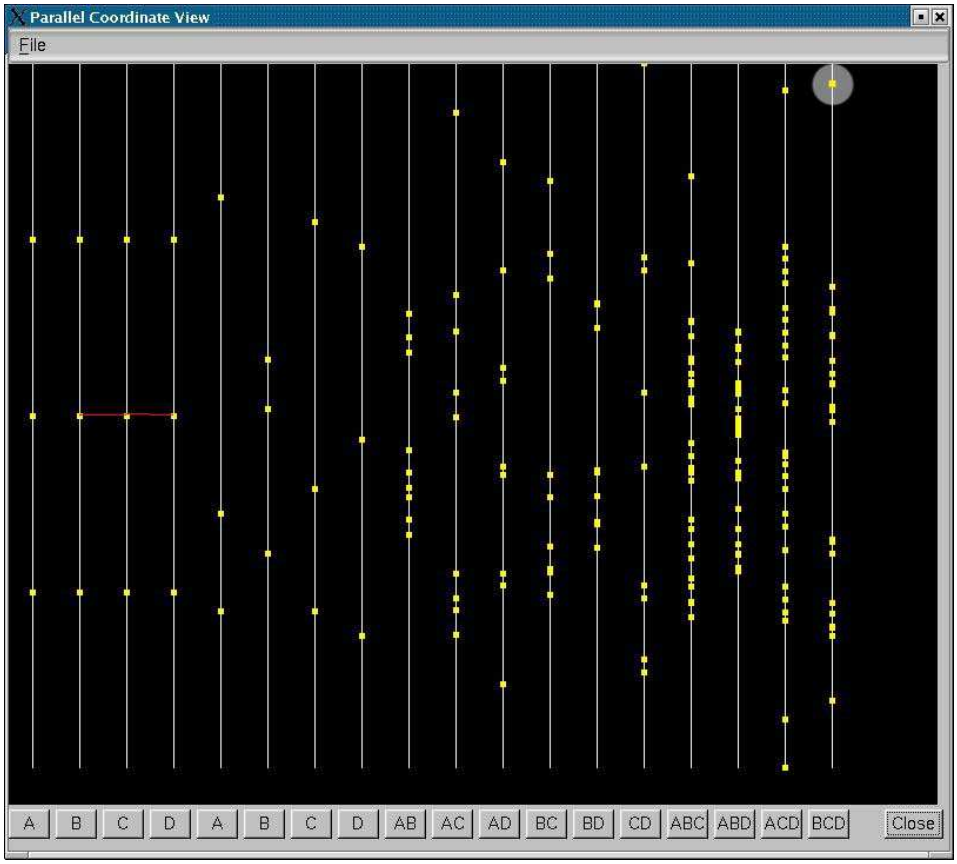



**Figures 3.6 (above) and 3.7 (below) show how these two Similar Gene/Expression-Level Combinations have Similar Correlation Values. They are also Similar to those in Figures 3.4, 3.5, 3.8, and 3.9.**






**Figures 3.8 (above) and 3.9 (below) show how these two Similar Gene/Expression-Level Combinations have Similar Correlation Values. They are also Similar to those in Figures 3.4, 3.5, 3.6, and 3.7.**



# CHAPTER 4 – IMPLEMENTATION

## 4.1 – The Beginning

In April, 2003 Dr. K.R. Subramanian created a visualization program that graphically showed the binary trees of the microarray data. He programmed it in C++, using the Visualization Toolkit (VTK) libraries [5]. He developed it on a SGI Octane II graphics workstation. This initial implementation was intended only as a "Beta" version to be expanded upon, and hence, had many shortcomings. It could only display a single level in the hierarchy, which was given by the user as a command-line argument. Displaying a different level of the cluster hierarchy required exiting and restarting the application. There was no labeling of what was being visualized. The rectangles comprising the graphical representation of this level of the cluster hierarchy were almost unidentifiable to the user, because the 3D picking that was employed made it exceptionally difficult to select the rectangle the mouse pointer was directly over. The program was hard-coded to read a particular data file, and there was also no support for the parallel coordinate graph of loglinear data files. The goal of this project was to expand and upgrade the work begun by Dr. Subramanian, and effectively reconcile the aforementioned shortcomings. Most time was spent on the treemap visualization, and when that was completed in early December, 2003, work was begun on the parallel coordinate graph.

## 4.2 – Treemap Visualization

The treemap visualization provides an interactive and intuitive tool for navigating the microarray cluster hierarchy. It consists of three parts: the treemap, the sidebar, and the genebar (figures 2.2, 4.1, and 4.2). The treemap shows details about the cluster hierarchy at a particular level down from a chosen cluster, the sidebar shows what that level is and what other clusters are on that level and above (closer to the chosen cluster), and the genebar shows which genes make up a cluster and provides for future integration with Yong Ye's program.

### 4.2.1 – The Treemap

This part of the visualization shows a treemap representation of part of the cluster hierarchy. It shows the details of the hierarchy, in treemap form, a chosen number of levels down from a chosen cluster (default is root). Figure 2.2 shows the treemap five levels down from the root cluster. To achieve this, the cluster hierarchy is traversed under the chosen cluster without exceeding the depth level. Each cluster visited is put on a growable array (vector). Each time a cluster at the specified depth is visited (or if a visited cluster has an individual gene as a child) during the traversal, it is also put on the vector of clusters, but its ID is also added to an additional vector to mark that gene as being on the bottom level in the treemap. When the traversal is completed, the program looks at the array of clusters and forms a new treemap from it.

In a treemap, a

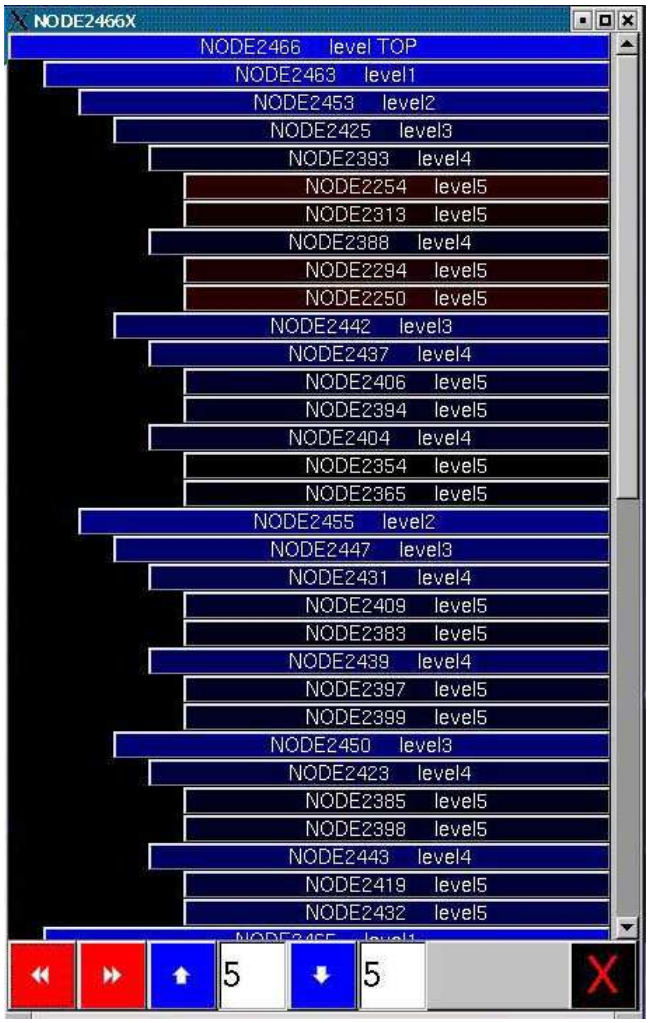



**Figure 4.1 – The Sidebar**

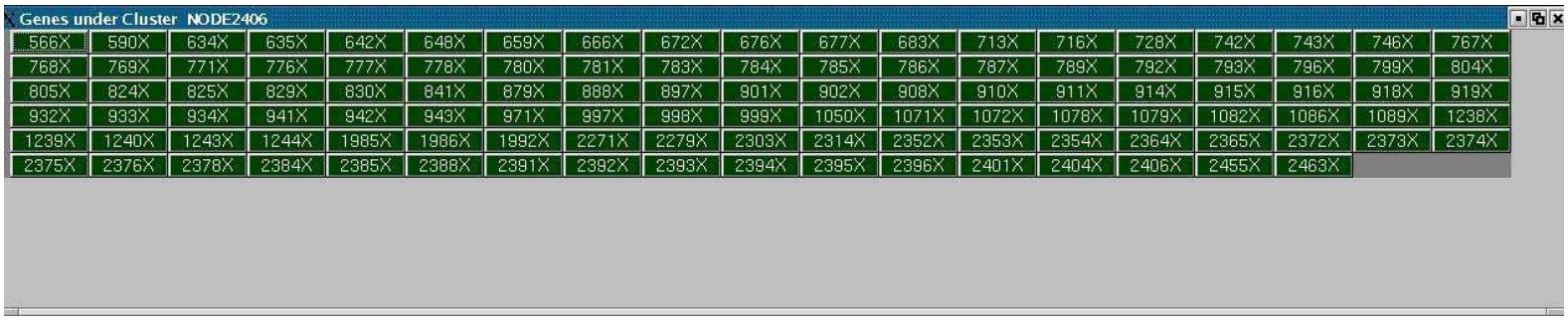



**Figure 4.2 –The Genebar**

parent rectangle has 100% of the space that can be allotted to each of its children. In this program, the parent rectangle's space is partitioned to its two children based on the number of genes that comprise each child cluster (each rectangle represents a cluster in the hierarchy). A child rectangle is allotted the percentage of its parent's space equal to the percentage of the parent's genes that come from that child.

Interface widgets and windows were added to the application to give a foundation for a graphical user interface (gui). The treemap is located in an interface-window widget, henceforth referred to as the "main window". Originally, this window was a hybrid interface-window/VTK render window. These hybrid windows allow the display of VTK 3D graphics inside a user interface. This was needed at the time, because the treemap was programmed using VTK classes (vtkCubeSource, vtkPolyDataMapper, vtkActor, vtkRenderer, vtkFlRenderWindow, and vtkFlRenderWindowInteractor). It consisted of various-sized 3D rectangular shapes (depicting the bottom-level cells in the treemap, which in turn represent clusters in the hierarchy) resting on a large yellow 3D rectangle, whose color shows through between the rectangles in the treemap, forming yellow boundaries between them. The treemap was rewritten to use light-weight, two-dimensional button widgets, and a window widget, eliminating all VTK objects. With the treemap now visualized with buttons on top of a yellow window, certain benefits were

reaped. The application was smaller, it ran much faster, the treemap could no longer be rotated in 3D space, and most importantly, the use of 2D picking was available. Accidentally rotating the treemap in the 3D space would ruin the straight-ahead view of the visualization, causing the yellow 3D background rectangle to no longer show through between the smaller 3D rectangle cells. If neighboring cells were of similar enough color, distinguishing between them would be impossible, and ruin the usefulness of the treemap. The user would then have to stop navigating the cluster hierarchy and attempt to undo the rotation (which can take time). The 2D picking is built in to the button widgets and eliminates the difficulties with 3D picking. The buttons use callbacks to handle events (like clicking on the button with a mouse). The buttons also allow for labeling and tool tip text, which the VTK cubes did not. The labeling and tool tip text add two additional pieces of data per cell to be visualized! Currently, the application uses the label to show how many genes comprise that cluster, and the tool tip text identifies the cluster without the need to click on it (figure 4.3).

Another feature built into the treemap is highlighting. When the user picks a rectangle by clicking on it, it turns the rectangle pink (figure 4.4). This useful feature aids the user by eliminating the need to remember the location of the rectangle that has been picked. If the treemap is showing the hierarchy several levels deep from a cluster, it is quite

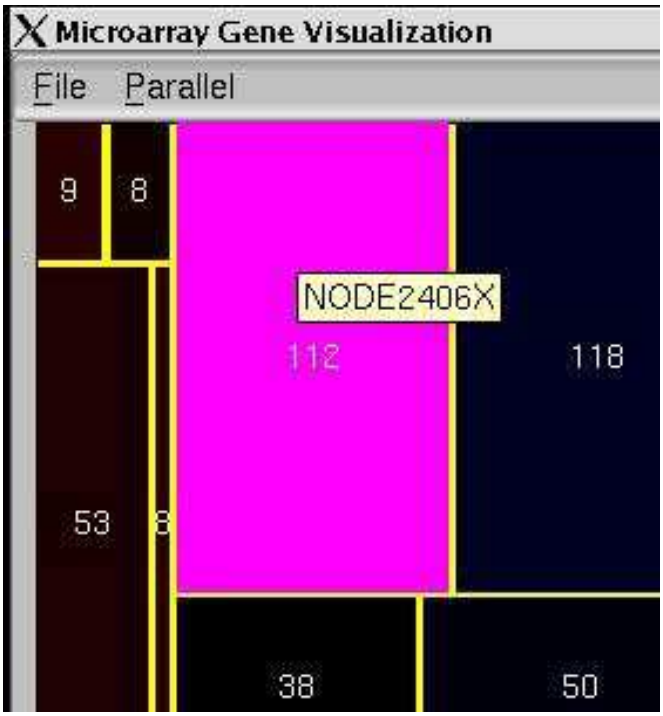



**Figure 4.3 – The Tool Tip Text**

cluttered (figure 3.2), and the researcher would have to locate the rectangle of interest each time his or her eyes left the screen. Highlighting also takes place in the sidebar. These features are connected and when a rectangle (or its cluster) is highlighted in one window, its corresponding cluster (or rectangle) is highlighted in the other window automatically (figure 4.4). In addition to highlighting, when a rectangle is clicked (picked), it generates a genebar window for that cluster. This feature will be detailed in section 3.2.3. It also marks that cluster to be the currently picked cluster. Other than marking and storing the currently selected cluster, the treemap can not initiate any changes in the visualization; that job belongs to the sidebar.


**Figure 4.4 – Highlighting**

### 4.2.2 – The Sidebar

The sidebar, shown in figures 4.1 and 4.5, is a powerful addition to the treemap visualization. It is a navigation tool that shows the subhierarchy that is displayed in the treemap in a different format. Like the treemap, it represents clusters using button widgets. Unlike the treemap, it represents every cluster in the subhierarchy with a button. The sidebar is supplied data from the hierarchy traversal described in section 3.2.1. Before a new traversal takes place, the current sidebar is deleted, and a new, empty one is allocated to take its place. During the traversal, each time a cluster in the hierarchy is visited, a call is made to the sidebar object, telling it to take on this cluster. Unlike with the treemap, the sidebar will take on clusters with children that are individual genes, and even individual genes themselves: as long as they are within the traversal bounds set by the user. Figures 4.1 and 4.5 show the cluster that was selected to be the root of the subhierarchy as the top button of a vertical sequence of button widgets. Each button is colored according to its cluster's correlation value, just as in the treemap, and each button is labeled with the name of the cluster, and its depth from the selected root. The button widgets are not added directly to a window in the sidebar. The sidebar has a window widget that acts as a wrapper for everything on the sidebar, but these buttons are added to a special scrolling window, which is added to the outer window.

Scrolling window widget is just like a regular window widget, but it can map coordinates

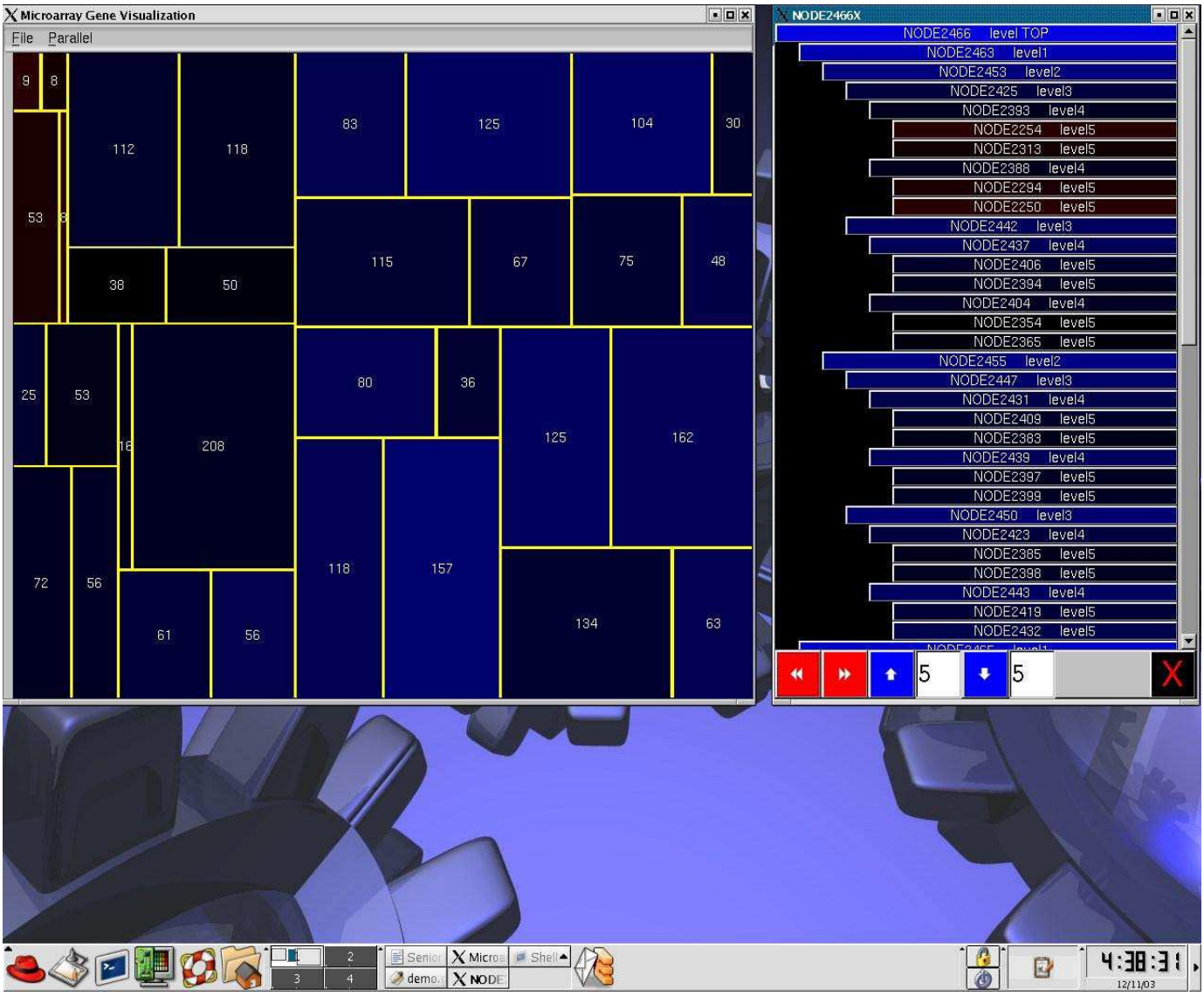



**Figure 4.5 – The Treemap and the Sidebar five levels deep from the root**

outside of its dimensions. When widgets are added to such locations in window, a scroll bar appears on the right, and/or on the bottom of the scroll window, allowing the user to scroll around a much larger window.

For each level of depth a cluster is from the chosen root, its button widget is indented. The clusters shown in the sidebar that are the furthest indented (and hence, are labeled with the deepest depth level) are those clusters shown on the bottom level of the treemap (those which have buttons assigned to them). As mentioned in section 3.2.1, the treemap will not show a cluster if one or both of its children are individual nodes, instead showing such a cluster's parent. If the parent's other child is a gene, it will show the parent's parent, and so on. The sidebar allows the user to recognize those cases, whereas with the treemap the distinction can not be made. To allow the distinction to be made, the sidebar will also show the individual gene-children of a cluster. Genes have no correlation value, and their buttons are all colored bright green to distinguish them from clusters. Clicking on a button in the sidebar highlights it by coloring it pink, and set that cluster to be the currently picked cluster. If the picked button corresponds to a cluster that is on the bottom level in the treemap, the program will automatically highlight the button widget in the treemap.

In addition to supplementing the information provided by the treemap, the sidebar also acts as the primary navigation tool of the application. Figures 4.1, 4.6, and 4.7 show the navigational area at the bottom of the sidebar. This takes in user input and updates the treemap and the sidebar with a new view of the cluster hierarchy. The buttons marked with arrows control navigational direction: The down button navigates down-arrow the cluster hierarchy, the up-arrow button navigates up the hierarchy, the back-arrow button goes back to the previous navigation choice (if applicable), and the forward-arrow button undoes the back-arrow button. The value-input fields tell the program how many levels down from the selected cluster to show, and how many levels up from the selected cluster to begin counting the levels down (setting a new root cluster n levels up from the selected cluster). The default values are both five. If the down-arrow button is clicked, the program will ignore the "up" value and proceed to traverse m levels deep from the selected cluster. If the up-arrow button is clicked, the program does not ignore the "up" value, and will go up n levels in the hierarchy, set that cluster to be the new selected cluster, and then proceed to traverse m levels deep from that cluster. These buttons are attached to callbacks that execute code when an event on them occurs. The input fields are not attached to any callbacks, since changing the values does not imply the user wishes to immediately proceed with an action. The down-arrow button's callback calls a method that gets the chosen cluster from the treemap


**Figure 4.6 – The Sidebar's Navigation Buttons**

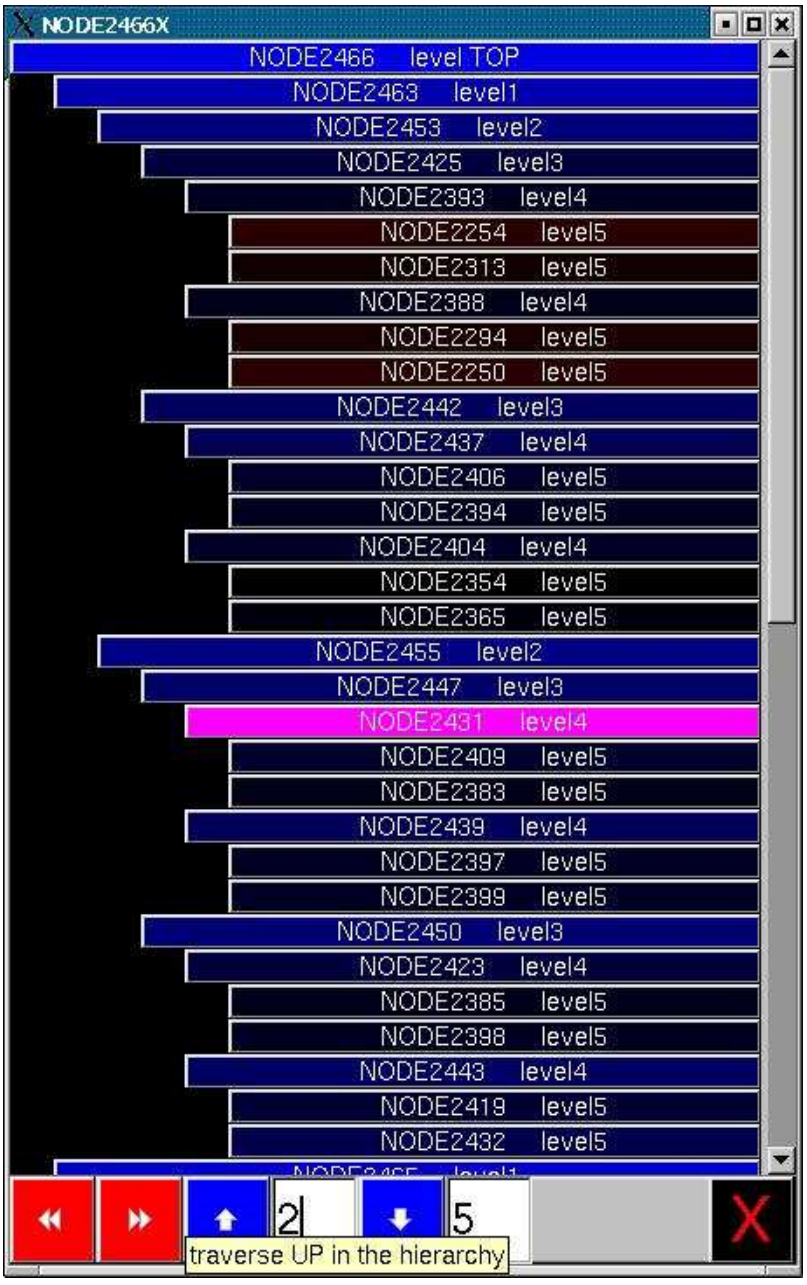



**Figure 4.7 – NODE2431 Selected. Instructions are to go UP Two Levels from that Cluster, and View the Fifth Level DOWN from There**

(an object of class Geometry), where that information is stored, and then initiates a new traversal to occur from the selected cluster, m levels deep. The up-arrow button does the same thing, but first tells the treemap that the new selected cluster is n levels up from the current one. The back-arrow ("undo") button's callback calls a method that tells the program to display the previously chosen visualization of the cluster hierarchy. The forward-arrow ("redo") button's callback calls a method that tells the program to display the visualization from where the user clicked the back-arrow button. These buttons undo and redo navigation choices, just like "undo" and "redo" undoes and redoes actions in a word processing program or a paint program. The undo/redo data is stored in a vector which is initially empty. Each time a new navigation action is performed, the root node and the levels of depth is added to that vector. Clicking "undo" initiates a hierarchy traversal using the root node and levels of depth stored in vector cell preceding the one storing the information of the currently displayed visualization. Clicking the "redo" button does the same thing, but it uses the information stored in the vector's cell immediately succeeding the one that corresponds to the current visualization. The user can "undo" all the way back to the first visualization, and "redo" all the way to the last visualization. The "undo" button has no effect if the user is viewing the initial visualization (from the hierarchy's root), and the "redo" button has no effect if there is no next visualization selection. Also, if a user performs some "undo's" and

the initiates a new visualization with the “up” or “down” buttons, the navigation history vector is erased from that point onward, and the new selection immediately succeeds the last “undo” state in the vector.

### 4.2.3 – The Genebar

Then the user clicks a button widget in the treemap, it brings up a window widget called the “genebar”. The genebar contains a scroll window widget that displays all of the genes that comprise the cluster the selected button represents (figures 4.2 and 4.8). Each gene is represented by a button widget and is colored bright green. The button is also labeled with that gene’s name. The buttons are assigned to a callback to handle events on them. Currently this callback does not execute any code. In the future, it will call a program written by PhD student Yong Ye that visualizes the same microarray data with the graph drawing technique described in section 2.1.3.

## 4.3 – The Parallel Coordinate Graph

The parallel coordinate graph is a special window widget that supports OpenGL graphics. When activated via a heading on the main window’s toolbar, this object reads in a loglinear data file. This information is stored in objects of a class called “LogLinear_Piece”. Each piece represents a gene/expression-level combination, such as “Gene A[low] + Gene B[high] + Gene C[normal]”, and its loglinear correlation. The


**Figure 4.8 – The Genebar with the Other Windows**

program generates a series of axes, one axis per gene combination, like "Gene A + Gene B + Gene C". Each axis stores the pieces that are functions of that gene combination in a vector. For example, piece "Gene A[low] + Gene B[high] + Gene C[normal]" would be stored in and graphed on axis "Gene A + Gene B + Gene C", as would all other expression-level combinations of those genes. The length of each axis measures the loglinear correlation value (transformed into screen coordinate), and the pieces are plotted on those grounds. There needn't be three different expression levels, or three different genes. These are defined in the data file, and realistically may be of any number. Right now, however, the program only supports data files that have four genes and three expression levels.

The OpenGL window has two coordinate systems, one for user-interface widgets, and the other for OpenGL. In this program, the OpenGL coordinate system was set to match the inflexible widget coordinate system (which always matches screen coordinates). This allows the use of user interface event handling to pick OpenGL graphics primitives in the window without having to do a coordinate transformation. The picking is done by first considering the x coordinate of the mouse click. Each axis stores its own x screen coordinate, and program compares these to the mouse click's x. When axis nearly (or precisely) matching that value is found, the program searches that axis' vector of

LogLinear_Pieces for one that nearly (or precisely) matches that y coordinate. When the data file is read and the pieces are created, the loglinear correlation values are stored in those pieces. These values are also transformed into the system used by the OpenGL window (screen coordinates), and the Loglinear pieces store this value as well. This way, the transformation only occurs once per piece and never more. This is the y coordinate the program searches for. When a match is found, the piece is plotted on the far left of the window, where there is a separate set of empty axes (figures 3.4 through 3.9). Each of these axes represents a single gene, and has n possible graph-points. The graph points represent the discrete expression levels in the data set (a data set with 3 expression levels will have 3 graph points per axis). The LogLinear_Piece is plotted by its definition, for example the graph in figure 4.9 represents the combination "Gene A[low] + Gene D[normal]" in a four-gene (genes A...D), three expression-level (low, normal, high) dataset. Each LogLinear_Piece stores its definition (its combination) in a vector. Each vector is m cells long, where m is the number of genes in the data set. In each cell of the vector an integer value is stored, which corresponds to an expression level. This program uses the value "-1" in a cell to show that combination excludes a gene. For example, "Gene A[low] + Gene D[normal]" is represented in the vector as [1, -1, -1, 2], and "Gene B[high] + Gene C[low] + Gene D[low]" would be stored as [-1, 3, 1, 1]. If more

than one LogLinear_Piece nearly matches the screen point clicked, all of those pieces will be graphed, each with a different color.

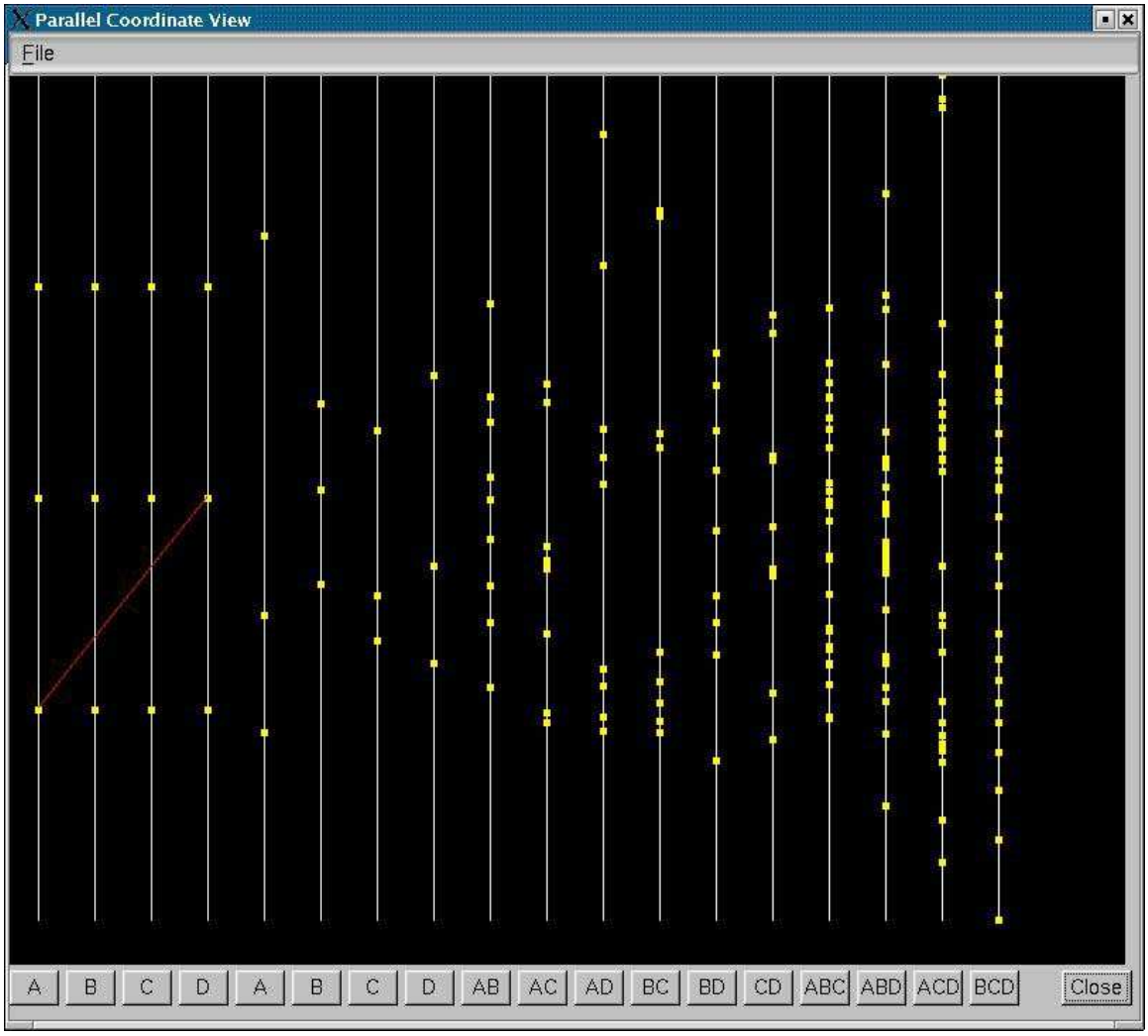



**Figure 4.9– The Parallel Coordinates Graph with a Square Picked and Graphed (the Red Line representing Gene A [low] + Gene D [normal]). The Data used here contains 4 Genes with 3 Expression Levels**

# CHAPTER 5 – RUNNING THE APPLICATION

## 5.1 – Starting the Program

The program is started by executing the file named “gm”. On Linux, this can be done in two ways. The first uses the console by going to the “linux” subdirectory of the “microarray” directory and typing “gm” and then return. The other is through one of the many graphical file browsers available and double-clicking on the file “gm”. On the SGI, executing the program is done through the console, going to the “sgi” subdirectory of the “microarray” directory, typing “gm”, and then return. The user will be shown a graphical file selection window and asked to choose a data file (figure 5.1). This remainder of this paper will show the application using the data file called “demo.gtr”. The program then loads the data file and shows the user the initial-state screen (figure 5.2). The initial state screen shows a plain yellow background and is awaiting user input of which of the two rectangle partitioning methods to use for the treemap visualization. Then the partitioning method is selected (figure 5.3), a pop-up window appears to ask the user “how many levels from the root node should be displayed?” The default is 5. When the user is finished, the program will show the treemap and the sidebar, starting with the root node, at the depth level chosen by the user. This paper shows this with the depth level of five (figure 4.5). Figures 2.2 and 4.1 show a closer look at the treemap and the Sidebar. The Sidebar doesn’t just show the clusters at the selected level of the hierarchy. It shows the

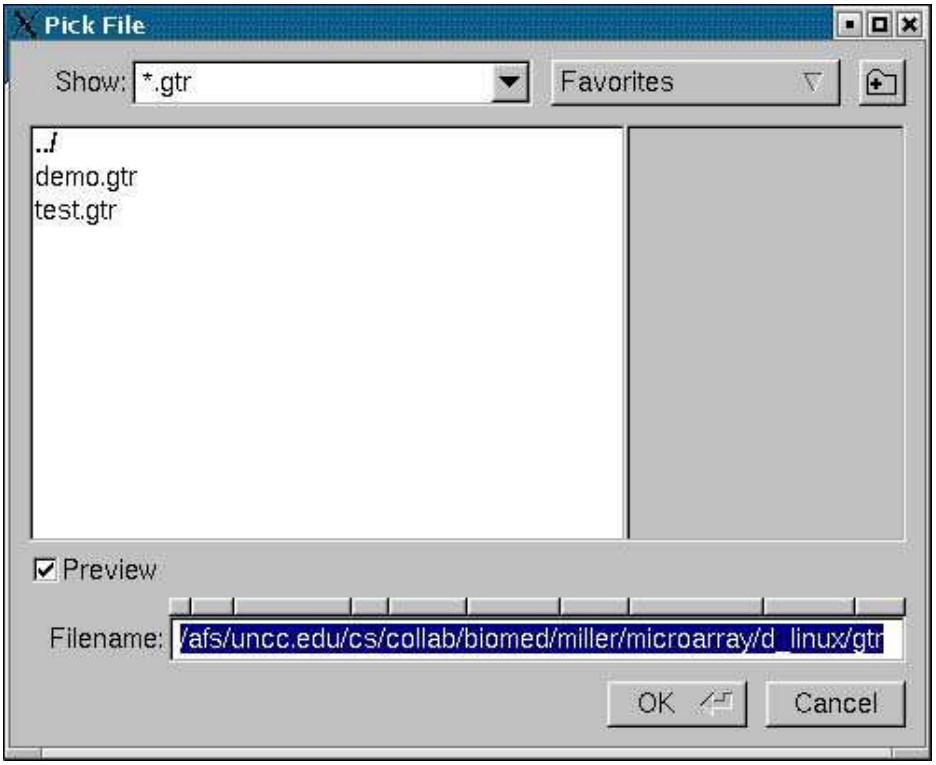



**Figure 5.1 – The File Selection Window**

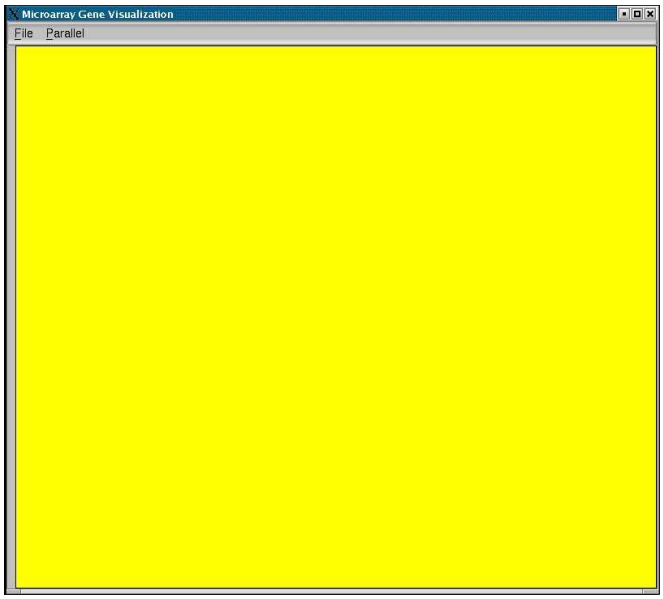



**Figure 5.2 – The Initial State**

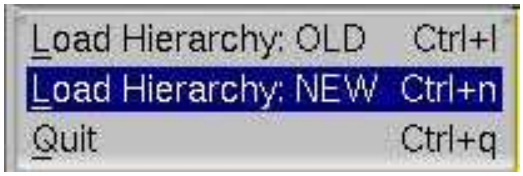



**Figure 5.3 – Select Partitioning Method**

subhierarchy with the root, or previously selected cluster at the top, and all the clusters between that and the selected depth level from the top. The clusters at the selected depth are indented the most, and are present in the treemap. Notice how the rectangles in the treemap are of different colors. These colors represent the cluster's (that the rectangle represents) correlation value. A rectangle that is black, or near black, has a correlation value near zero. This means that the individual genes that comprise that cluster have almost no correlation between them. A rectangle that is blue has a positive correlation between its genes: the lighter and brighter the blue, the stronger the positive correlation. Red rectangles are the same as blue ones, except that they show a negative correlation between genes: the lighter and brighter the red, the stronger the negative correlation. Each rectangle also has a number on it. This number tells the user how many individual genes comprise that cluster.

### 5.2 – Navigating the Hierarchy

This application was designed to give the user flexibility in navigating the hierarchy. The most basic way to navigate the hierarchy is to select a cluster on either the treemap or the Sidebar, input the number of levels down from that cluster to navigate, and then click the down-arrow button. When a cluster is selected, either in the treemap, or in the Sidebar, that cluster is highlighted in pink in both windows (figure 4.4). This helps the user see which rectangle in the treemap belongs to which

TreeNode. When selecting a TreeNode on the treemap, a new window, called the Genebar, appears (figures 4.2 and 4.8). The Genebar shows all of the individual genes that make up the cluster (TreeNode) selected. Also, the rectangles in the treemap are assigned tool tip text, which means, when the mouse pointer is stationary over a rectangle in the treemap for two seconds, a small piece of text will appear next to the mouse pointer (figure 4.3). The rectangles' are programmed to display the name of the cluster they represent.

If the user chooses to navigate down the hierarchy five levels from cluster NODE2401, both the treemap and the Sidebar update to represent this (figure 5.4). If the user decides he or she didn't want to view this after all, there is a "back" button in the Sidebar that will go to the previous visualization. The "back" button is bright red and marked with a left-double arrow (figures 4.6 and 5.4). This button can undo every navigation choice the user made during this running of the application, all the way back to the first visualization of the root cluster. If the user decides again that the "undone" choice was the correct one after all, there is a "forward" button in the Sidebar that will undo any number of uses of the "back" button. This button is bright red and marked with a right-double arrow (figures 4.6 and 5.4). If the user goes "back" and then chooses a new navigation selection, the user can no longer go "forward" from that point; that part of the history is erased. As long as no new

navigation

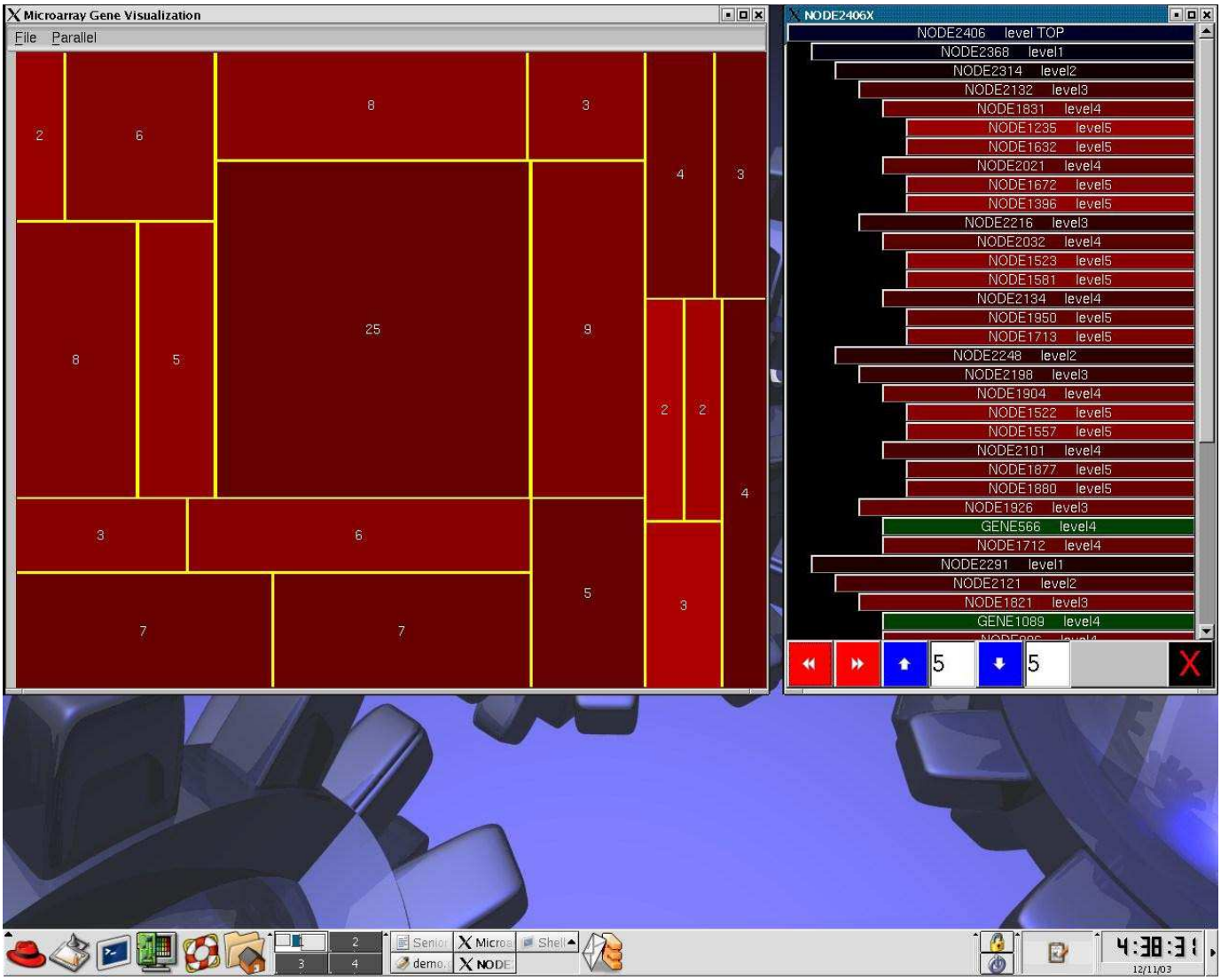



**Figure 5.4 – 5 Levels down from NODE2401**

choices are made, the user can go “back” all the way to the beginning, and then “forward” all the way to before he or she began going “back”.

It is also possible to navigate upwards in the hierarchy. If, from the root cluster, the user wants to show the hierarchy five levels down from the cluster two levels up from cluster NODE2431, the user must simply click on that button on the Sidebar (or on the treemap), click on the value input box next to the blue button marked with an up-arrow, erase its contents and replace them with “2”, click on the input box next to the blue button marked with a down-arrow, erase its contents and replace them with “5” (or leave it alone if it already says “5”), and then click the up-arrow button (figures 4.7 and 5.5).

The user can navigate up and down the hierarchy from the beginning to the end, choosing any number of levels below the selected cluster to be shown. Figure 3.1 shows an example of a cluster at the end of the hierarchy. Notice that both of its children are individual genes; the hierarchy can not be navigated any farther down from this point.

### 5.3 – Rectangle Partitioning Methods

In section 5.1, when describing how to start the application, the user clicked “file/Load Hierarchy: New”. There was also a choice for “Load Hierarchy: Old” (figure 5.3). “New” and “Old” are perhaps poor names to

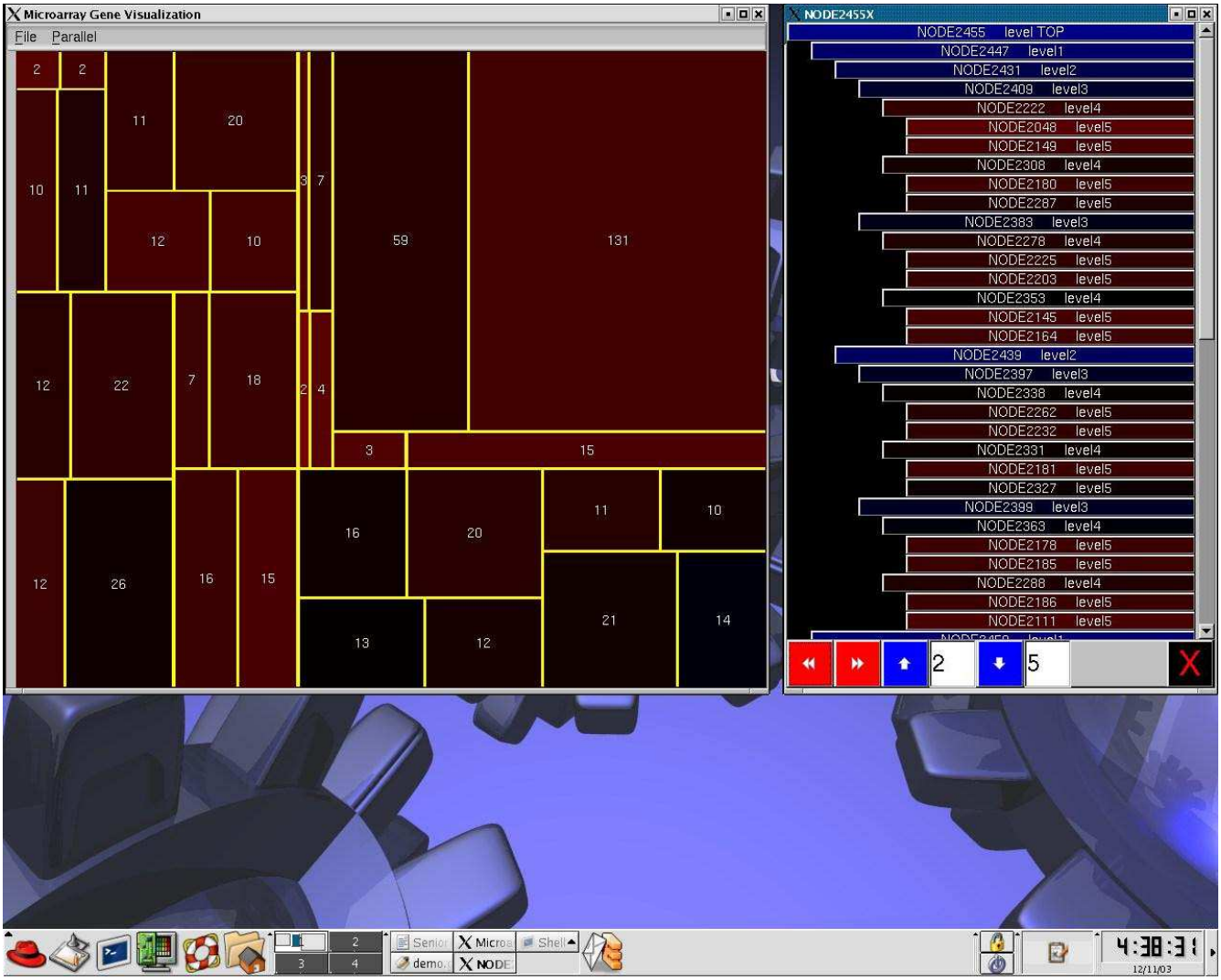



**Figure 5.5 – Instructions from Figure 4.7 AFTER the Up Arrow is Clicked**

describe what each choice means, but each one determines a different way of partitioning the rectangles in the treemap. The "New" way partitions the rectangle based on the number of individual genes comprise the cluster the rectangle represents. The "Old" way partitions the rectangle based on the cluster's correlation value (as is the rectangle's color). These different partitioning methods yield totally different looking treemap visualizations (the number of rectangles and their locations relative to each other are the same, but the sizes are all different). Figure 5.6 shows the root cluster, five levels deep with the "old" partitioning method. Compare this with figure 2.2, showing the same thing, but with the "new" partitioning method. All features described in earlier sections of this chapter apply equally, regardless of the partitioning method chosen.

## 5.4 – Parallel Coordinate Graph

On the menubar of the main window (the one that houses the treemap visualization) there is a button labeled "Parallel". Clicking the "Parallel" button opens a new window that graphs gene/expression-level combinations against the combinations' effect correlation in parallel coordinates (figures 3.4…3.9, 4.9). The y-axis of the graph measures the effect correlation value of a combination, as described in [6]. Each axis represents a combination of genes, and the squares plotted on an axis represent the discrete value permutations of that gene combination.


**Figure 5.6– The Same Visualization Criteria as in Figure 2, but with an Alternate Partitioning Method**

Each gene in a cluster can take one of n discrete expression levels, such as low, normal, and high. Each square plotted on an axis represents a combination of genes and expression levels (such as gene A [high] + gene B [low] + gene C[normal]). If a square is picked, a parallel coordinate graph (on the far left of the window) appears, showing that square's gene and discrete value combination (figures 3.4…4.9, 4.9 – the red line). If there are multiple squares near the one that was picked, each of those squares will be graphed simultaneously (each with a different color). This feature uses a different data file from the other visualizations, because the goal of this one is different: this one is designed to allow the user to rapidly search for correlations between combinations of genes/expression-levels, and those combinations' effect correlation values. A parallel coordinate graph was chosen for this because, as Alfred Inselberg says in [3], "the special strength of parallel coordinates is in modeling relations."

# CHAPTER 6 – FUTURE WORK

This project is rife with opportunity for expansion and further improvement. Most of this opportunity lies in integration with Yong Ye's application and improvements to the parallel coordinate graph.

## 6.1 – Integration with Yong Ye's Application

This application was built with integration in mind from the start. The Genebar was built specifically for integration purposes: it shows all of the genes that comprise a cluster. Yong Ye's application visualizes genes and their relations to other genes in a cluster using a node-link graphing method. The callbacks governing events concerning the GeneButtons in the Genebar are designed to make calls to Yong Ye's program to visualize the selected gene. There are most likely other areas where integration is not only possible, but desired.

## 6.2 – Improvements to the Parallel Coordinate Graph

Labeling the special axes a different color from the other axes would be a nice improvement, albeit low priority. Replacing the huge C++ switch blocks with an algorithmic manner of defining each LogLinear_Piece would be a tremendous upgrade to the application. As is, the application can read loglinear data files that are arranged in a certain order, but a nice improvement would be to modify the file parsing so it no longer demands the loglinear file to be in any order.

### 6.3 – Other Improvements

It is certain that other improvements not mentioned above will be conceived of. Perhaps an even more meaningful rectangle partitioning method will be desired. One nice improvement would be to revamp the scrolling window class used by the sidebar. The current scrolling window class seems to only be able to map roughly thirteen thousand additional pixels along the y axis before it “breaks”, and the Sidebar’s usefulness as a visualization tool is lost, but it still retains usefulness for controlling navigation. When this occurs, all cluster picking must take place on the treemap itself, as picking on the sidebar is impossible.

# APPENDIX A -- ACKNOWLEDGEMENTS

First and foremost, I would like to deeply thank Dr. K.R. Subramanian. He is a wonderful teacher and gives his senior students the opportunity to work on research projects that are interesting, cutting edge, and potentially beneficial to all of humanity. He pushed me farther than any other professor or teacher I've ever had before. His assignments and projects are quite challenging, but he always helps you understand the challenges and to overcome them. I can honestly say that I have learned more in any one course I've taken with Dr. Subramanian than I have in all non-Dr. Subramanian courses I've taken combined.

I would like to thank Josh Foster for helping me troubleshoot problems and lending me his expertise in computer graphics. Thanks Josh!

I would also like to thank Dr. David Bashor and Dr. John Brockway, with whom I worked on previous visualization projects, for helping me develop an interest in biomedical visualization.

Finally, I would like to thank my fiancée Jana for putting up with me during this stress-filled last year of my undergraduate education.

# APPENDIX B -- SOFTWARE ARCHITECTURE

## B.1 – Programming Language and Environment

The current implementation of GM was programmed in C++ using the FLTK 1.1 libraries for X11. Figure B.1 shows the class interactions of the program. It was developed simultaneously on a SGI Octane II graphics workstation running IRIX 6.5.20, and on a PC-Compatible running Red Hat Linux 9.0. It was compiled under CC99 (CC version 99) on the Octane II, and under GNU g++ 3.2.2 on the PC-Compatible. On both platforms, all code was written in a text-editor (Nedit on the Octane II and Kate on the PC-Compatible) and command-line compiled. All user interface code was developed using the FLTK 1.1 libraries for C++. “FLTK (pronounced "fulltick") is a cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and MacOS® X. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL® and its built-in GLUT emulation (1).” FLTK was chosen because of its straight-forward simplicity, small size, and its availability on multiple platforms. With FLTK it is possible to rapidly hand-code effective graphical user interfaces.

## B.2 -- Application Layout

The application GM is divided into many different classes. Large classes are defined in a header file (“.h”) and an implementation file (“.cc”), which bear the name of the class. Small classes are defined and implemented

in files bearing the name of the larger class that makes use of them. The classes are divided into two groups: data classes and user interface classes. Data classes together form the data layer, and the graphical visualization classes with the interface classes form the graphics and user interface layer. Figure B.1 shows the class interaction.

### B.3 – The Data Layer

The data layer exists behind the scenes, invisibly computing the data that makes the application work. These classes make up the middleware of the application, and were programmed and debugged before the classes that graphically represent their data were even started.

#### B.3.1 – Goals of the Data Layer

These classes had to be completely independent of the graphical user interface (GUI). They are required to efficiently perform any combination of: processing data, storing it, and make it available upon request. These classes in this layer are responsible for parsing the data file, converting its contents into node objects, preserving the hierarchical structure of the data file in those node objects, traversing the vector of node objects to yield all nodes at a particular level, determining treemap information for each node at that level, storing navigation data for undoing and redoing navigation choices, storing axis information for the parallel graph, storing loglinear data for a combination of genes, and

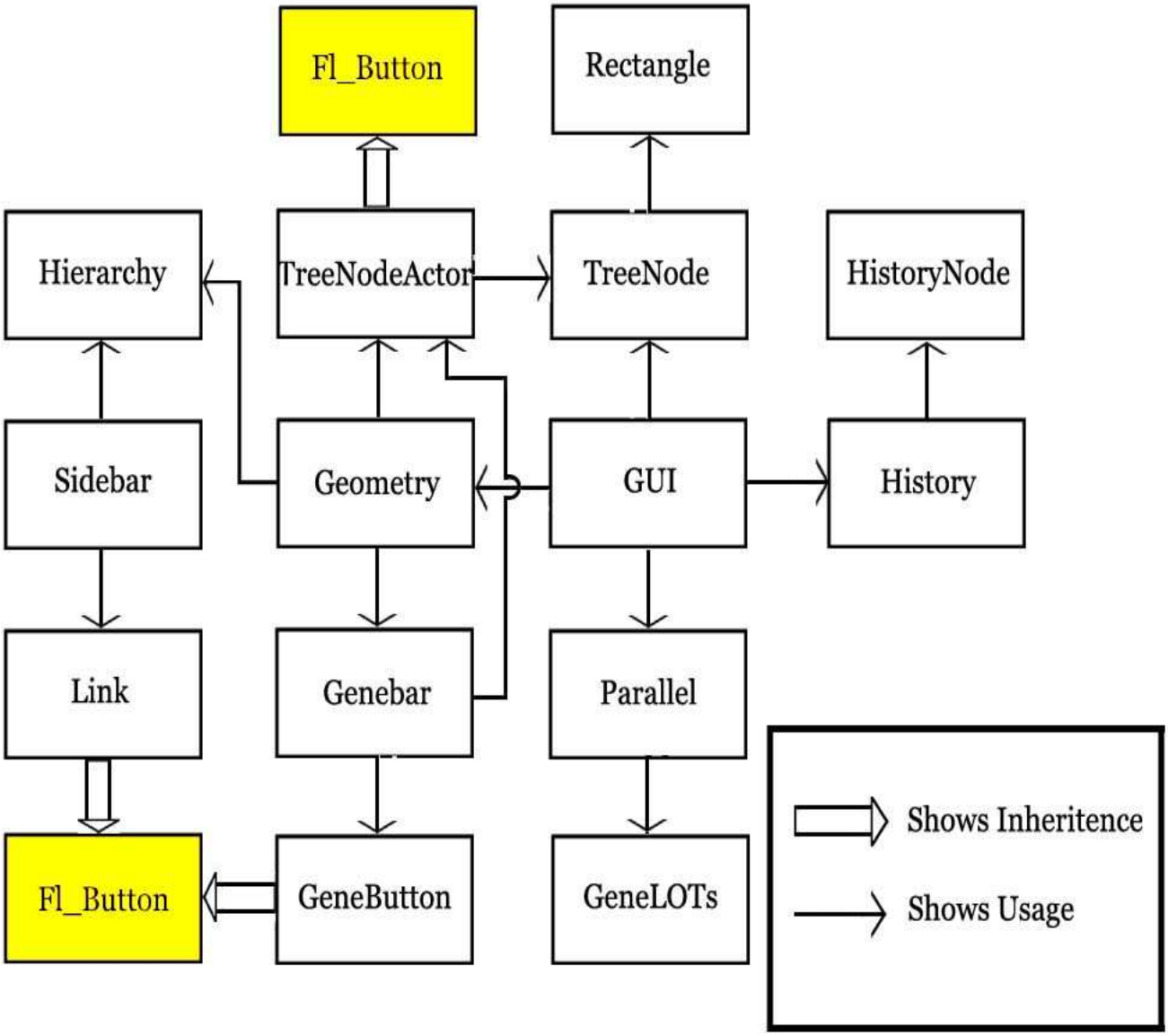



**Figure B.1 – The Class Relationships in the Application**

storing window coordinates for that data.

### B.3.2 – Designing the Data Classes

These classes are designed specifically for the GUI class(es) that will use them. When a new visualization is added to the application, these classes are designed first, providing a solid, debugged foundation on which the graphics can be placed.

### B.3.3 – Description of the Data Classes

#### B.3.3.1 – Rectangle

An object of class Rectangle simply stores five single-precision floating point values for a cluster of genes. Four of these values mark a rectangle's normalized Cartesian coordinates, and are stored in a four-celled array. The other stores a normalized correlation value. The data stored in objects of this class are used by the application for the correct placement of the rectangles in the visualization, and to determine the correct color representation of the rectangle. These represent a cluster's position in the hierarchy, and show "how similar" or "how dissimilar" the individual genes comprising a cluster are to each other. This class has two constructors. The first takes no parameters and creates a rectangle without setting any values. The second takes four parameters and creates a rectangle with its normalized Cartesian coordinates assigned. The correlation value must always be manually assigned. Mutator

methods allow other objects to set an object's data, and, in this application, begin with "Set" or "set". Class Rectangle contains mutators to individually set any of the five values it stores. Accessor methods allow access to an object's data as the return-value of a function. In this application they begin with "Get" or "get". Class Rectangle contains six accessor methods: five to individually access the five values stored, and one that accesses the four normalized Cartesian coordinates by returning a pointer to the four-celled array.

#### B.3.3.2 – TreeNode

Class TreeNode is the elementary storage-structure of the application. Each object is a node in a binary tree. It represents a single gene, or a cluster of genes, in the microarray gene hierarchy. Class TreeNode has only the default constructor (no parameters). In addition to pointers to left and right children TreeNodes, this class contains a pointer to the parent TreeNode. This back-pointer allows for navigating up the hierarchy, instead of only navigating down the hierarchy. Class TreeNode also contains a pointer to a Rectangle, a void pointer to a class that graphically represents a TreeNode, and several primitive types storing the type (gene or cluster), correlation value, a partitioning value (determines where to splitting its rectangle space into two rectangles: one for the left-child, one for the right-child), and a dimension value (determines whether the splitting of the rectangle is vertical or

horizontal). Class TreeNode has its own "<" operator to allow for sorting of the objects in a STL linked list. This class has a full selection of accessor and mutator methods.

#### B.3.3.3 – NodeData

Objects of class NodeData are for temporary storage and conversion of data from the hierarchy data file. The hierarchical data file is ASCII text with each line representing a grouping (cluster) of genes. The lines are comprised of 3 character strings and a floating point number that represents the correlation value of that cluster. The character strings are the names of the cluster being described, followed by its left-child and then right-child (which may be clusters or individual genes). The children character strings are sent to instances of NodeData to extract the numeric IDs which are embedded in the cluster/gene names, and to store this information to correctly set the parent and child pointers in the TreeNode objects.

#### B.3.3.4 – Hierarchy

Class Hierarchy is responsible for reading the hierarchical data file, creating the TreeNode objects to represent the data in that file, storing those TreeNodes that represent clusters of genes (as opposed to individual genes) on a STL vector, and finally, traversing the hierarchy. It has no member data of its own that is of any consequence, preferring

to work with and modify data passed to it through its constructor. The constructor takes two parameters: a pointer-to-character storing the name of the data file to be used, and a pointer-to-vector-of-*TreeNodes. The Read() method parses the hierarchical data file, creates a new TreeNode for every parent cluster it reads in, and creates objects of NodeData and places them on one of four vectors for later setting the parent and child pointers to establish the tree hierarchy in the TreeNode objects. After the data file is completely read in, the NodeData vectors are traversed, and the parent/child relationships are set. Individual genes are only referenced as the children of clusters in the data file, and no TreeNode object is allocated for them during the file parsing. This is done while the vectors are traversed and relationships defined. Individual genes are important for determining the make-up of a cluster, and this information should not be discarded. On the other hand, the treemap visualization only shows clusters, and it would not be prudent to store the individual genes on the same vector. This is the reason the TreeNodes representing leafs (individual genes) are created during the relationship assessment process. This way they are still attached to their parent clusters via parent and child pointers, but are not present on the TreeNode vector. The hierarchy traversal methods are recursive algorithms for starting at a given root TreeNode, and returning all TreeNodes that occupy a specified level of the hierarchy and represent clusters of genes. One method, called “GetTreeMap_R_Old” traverses the

hierarchy and sets each rectangle's partitioning value based on that cluster's correlation value. The other traversal method, called "GetTreeMap_R_New", sets each rectangle's partitioning value based on the number of individual genes that comprise that cluster. The user can select which traversal method to use within the program's menus. The traversal methods also call the Sidebar's "addLine" method at every recursion, supplying the Sidebar with the subset of the overall hierarchy the user is currently interested in.

#### B.3.3.5 – HistoryNode

This class simply stores two integers: the ID number of the current root cluster (the cluster at the top of the hierarchy of what is currently being visualized on-screen in the treemap), and the level in the hierarchy (relative to the root node) to show in the visualization. There is only the default constructor, and both members are public.

#### B.3.3.6 – History

Class History stores the user's navigation choices on a STL vector. This data allows the user to undo previous navigation choices and then either redo the choices, or make new choices. It manages the vector automatically, erasing HistoryNodes when applicable, creating them when applicable. There is only the default constructor, and public methods include those used to add a navigation choice on to the vector,

or return the previous or next choices.

#### B.3.3.7 – LogLinear_Piece

This class is a hybrid between a data class and a graphical user interface class. An object of this class represents a unique combination of genes and its loglinear correlation value. The loglinear value is stored in two forms: one being the actual floating-point value, the other scaled to the y-axis of the window where this information will be displayed. It also contains a vector of integers that represents the combination of genes. It uses the default constructor.

#### B.3.3.8 – Axis

This class represents a group of combinations of genes as an axis in parallel coordinates. Since each gene in a combination can take one of n discrete values, there is more than one combination of the same genes. An Axis represents all the combinations of the same grouping of genes, and stores the LogLinear_Pieces that represent each combination. For example, if two genes, gene A and gene B, are grouped, and each can take three discrete values, an Axis would represent combination gene A + gene B, and it would store the nine LogLinear_Pieces that comprise that combination: gene A [1] gene B[1], gene A[1] gene B[2], gene A[1] gene B[3], gene A[2] gene B[1], all the way through gene A[3] gene B[3]. The combination is stored on a STL vector of LogLinear_Pieces. Other data

stored is the Axis' x-coordinate position in window coordinates, and a character string for the Axis' name. There is only the default constructor, and since all members are public, there are no accessors or mutators.

#### B.3.3.9 – Pair

Objects of class Pair simply store two double-precision floating point values. These values represent an (x,y) coordinate in window coordinates. This class has only the default constructor, and since both members are public, has no accessor or mutators.

### B.4 – The Graphics and User Interface Layer

The graphical user interface layer exists on top of the data layer. The classes that comprise this layer together make the front-end of the application. They have FLTK widgets or openGL code, and either react to user input or simply graphically represent data. Each class is programmed and debugged only after the data classes it uses are completely debugged.

#### B.4.1 – Goals of the Graphics and User Interface Layer

The classes that comprise the graphics and user interface layer must take data from classes in the data layer and blend it with a graphical user interface, creating a user-friendly, intuitive, and interactive

visualization of the data. The classes that comprise this layer are responsible for creating the different FLTK windows in the program, such as the main window (contains the treemap visualization), the sidebar window, the graphbar window, and the parallel coordinate FLTK-openGL hybrid window. They are also responsible for handling mouse and text input via a framework of callbacks.

### B.4.2 – Designing the Graphics and User Interface Classes

These classes must be built on top of, and are completely dependent on the data classes. This ensures that the data classes are not dependent on the graphics classes, which is of paramount importance: if the style of graphical representation of the data in the data layer changes, the classes in the data layer should not need changing.

### B.4.3 – Description of the Graphics and User Interface Classes

#### B.4.3.1 – GUI

The class GUI is the main user interface class and the main control class. The allocation of an object of this class starts the application. It contains the main window, on which the treemap visualization is placed. This class has, as members, many large and important objects, such as the vector of TreeNodes, the treemap visualization, the parallel coordinate visualization, the navigation history, and the file selection window. The constructor takes, as parameters, the command line

arguments given with the executable. As of this version of the application, there are no supported command line arguments; the name of the hierarchical data file is now selected from a graphical window, instead of being supplied as an argument. The constructor first creates the file selection window, and then creates a new Hierarchy object, which automatically parses the data file and populates the TreeNode vector. The constructor then creates the main FLTK window and the Geometry object, which, when told, will display the treemap visualization. The Geometry object is then bound to the main window, which then waits for user interaction.

#### B.4.3.2 – TreeNodeActor

Class TreeNodeActor inherits from class Fl_Button (FLTK Button). It contains a pointer to an object of class TreeNode, and acts as a graphical representation of that TreeNode object. TreeNodeActor was originally a subclass of vtkActor, which represents an object in a 3D scene, and "combines object properties (color, shading type, etc.), geometric definition, and orientation in the world coordinate system (5)." TreeNodeActor gets part of its name from its former parent class. When it was updated to subclass an FLTK button, the old name was kept, because it still adequately describes the relation between an object of this class and the TreeNode it represents. Its constructor takes only those parameters it passes to its parent class: integers for the x and y

coordinates of the upper-left corner of the button, integers for the button's width and height, and a character string for the title. Its only member data is the pointer to a TreeNode object. It does, however, include the same accessor and mutator methods that class TreeNode possesses. These accessor and mutators only call their counterparts in class TreeNode, and only exist as convenience functions.

#### B.4.3.3 – Geometry

Class Geometry is responsible for the main treemap visualization. It stores a vector of TreeNodeActor objects, assigns them each a TreeNode, and also assigns the actor to the TreeNode (so they can both point to each other). It contains its own Fl_Window (FLTK Window), on which it places the actors. Its constructor takes the following parameters: a void pointer to the main window in class GUI, integers marking the x,y position and width and height of where the Geometry window will be placed, a pointer to the Hierarchy object, and a pointer to the TreeNode vector. It uses the pointer to the main GUI window in order to add or remove its own window to or from the GUI window. This pointer is casted to void so file "gmaGeometry.h" need not import (#include) file "gmaGui.h", which would create an infinite dependency loop during linking, because "gmaGui.h" imports "gmaGeometry.h". The file "gmaGui.h" is instead imported by "gmaGeometry.cc", where the void pointer is cast back to type GUI. This class is responsible for the

creation of the sidebar and the genebar windows. Class Geometry contains few accessor or mutator methods, only setting and getting the currently-picked-TreeNode. The sidebar and genebar pointers are public members. It contains a powerful method, called "BuildHierarchyGeometry", which is called when the treemap is to be updated. This method momentarily "hides" (makes invisible) the Geometry window, deletes the current Sidebar object (if allocated), creates a new Sidebar object, calls the Hierarchy's "GetTreeMap" method (with the currently-picked-TreeNode as the root), and uses the new treemap to set the TreeNodes' rectangles' dimensions. It then "shows" (makes visible) the Geometry window, which now contains the updated treemap visualization. This class also contains a static callback function for the TreeNodeActors (Fl_Buttons), which is called when one of these buttons is clicked. This callback calls a method, which in turn, creates a new instance of class Genebar, showing all the genes that comprise the cluster the TreeNodeActor represents.

#### B.4.3.4 – Link

Class Link also inherits from Fl_Button. It stores only two integers and a STL string, representing a TreeNode's ID, that TreeNode's parent's ID, and the TreeNode's name, respectively. It has two constructors, the default constructor, and one that takes the aforementioned Fl_Button geometric parameters, and the ID's and name. It has no accessors or

mutators: all data is public.

#### B.4.3.5 – Sidebar

This class is almost solely responsible for taking navigational user-input, and setting in motion the update of the treemap visualization. It has only the default constructor, which creates an Fl_Window, an Fl_Scroll (a window that has scrollbars on the side, allowing more objects to be placed in that window than will in the window's bounds), Fl_Buttons to handle navigation up and down the hierarchy, as well as forward and backward in the navigation history, and it also creates two value-input boxes, so the user can dictate how many levels up the hierarchy to navigate, and how many levels down from the selected TreeNode to show in the treemap visualization. The Sidebar contains a STL map of Link objects, which maps each Link object to an ID value. Sidebar objects are created by the Geometry object, which also calls the Hierarchy to traverse the TreeNodes. While traversing, the Hierarchy sends TreeNode pointers to the Sidebar via the "addLine" method, which creates a new Link for each TreeNode, labels it according to the name of the TreeNode it represents, colors it accordingly, and adds it to the Sidebar's Fl_Scroll window. When the Hierarchy stops sending new TreeNodes, the Sidebar adds the Fl_Scroll window to its main window, and makes its main window visible. It then takes user input via the navigation buttons, input fields, and the many Links (Fl_Buttons) it contains. The Sidebar

will also show the individual genes (in green) that exist at that level of the hierarchy, unlike the treemap visualization, which will instead show the cluster in the level immediately above.

#### B.4.3.6 – GeneButton

The GeneButton class is yet another subclass of Fl_Button. It contains only a single integer representing an individual gene's ID, and a STL string containing that gene's name. Its constructor takes the standard Fl_Button parameters, plus a character string for the name, and an integer for the ID. There are no accessors or mutators, as both the name and ID are public.

#### B.4.3.7 – Genebar

Objects of class Genebar show all of the individual genes that comprise a cluster. It contains an Fl_Window, an Fl_Scroll, and a STL linked-list of TreeNode objects. A Genebar object is created when a TreeNodeActor (Fl_Button) in the treemap visualization is clicked. The Genebar is given a pointer to the TreeNode that actor represents. From there, the Genebar uses its own recursive Hierarchy traversal method, "findGenes_R", to return only the individual genes (TreeNode objects) that exist in that subhierarchy with the selected TreeNode as the root. Each gene (TreeNode) returned by the traversal method is placed in a vector. When the traversal is finished, the vector of genes is sorted in incrementing

order on its ID number. Then the vector is traversed, and for each gene (TreeNode) in the vector, a GeneButton is created, labeled according to the gene's name, colored green (to mark it as an individual gene, as opposed to a cluster), given a size (constant), a position inside the Fl_Scroll, and is finally added to the Fl_Scroll. This class contains a static callback for the GeneButton objects. The callback calls a method that determines the ID of the selected GeneButton. This method will be the bridge to Yong Ye's application that shows information about individual genes.

#### B.4.3.8 – Parallel

Class Parallel is a subclass of a hybrid FLTK/OpenGL window (Fl_Gl_Window). These hybrid classes allow both FLTK widgets and OpenGL graphics to be displayed in the same window. Objects of class Parallel display a parallel coordinate graph of a Loglinear data file. "The special strength of parallel coordinates is in modeling relations. (3)" "The display of multivariate datasets in parallel coordinates transforms the search for relations among the variables into a 2-D pattern recognition problem. (3)" Class Parallel contains a STL vector of Axis objects, which contain a STL vector of LogLinear_Piece objects. This class uses OpenGL to draw lines representing the axes and plots squares representing gene combinations (LogLinear_Piece) on those axes. The y dimension of the axes represents the loglinear values. It also has a number of axes that

show exactly which combination of genes and each gene's discrete value a square represents. The square that is graphed on these special axes is chosen via picking. The OpenGL canvas dimensions are set to match the window dimensions (in screen coordinates) of the Fl_Gl_Window, allowing the use of FLTK mouse event handling on OpenGL primitives. Its constructor takes only the standard FLTK widget parameters, four integers (x, y, width, and height), and a character string representing the name.

#### B.4.3.9 – Pwin

Class Pwin contains an Fl_Window and an object of class Parallel. When the user clicks the "Parallel" button in the menu bar of the main window in class GUI, the callback handling that event creates a new instance of this class. The constructor of this class takes four integers and a void pointer as parameters: the window's x and y position, the window's width and height, and a pointer back to the object that created the new Pwin. Pwin creates a new Fl_Window and a menubar for that window. It then creates a new instance of Parallel and places it within its Fl_Window. Future work with this application will undoubtedly add more functionality to this menubar.

# APPENDIX C -- REFERENCES


[1] "FLTK" Fast Light Toolkit. December 2003. <http://www.fltk.org>

[2] Y. Fua, M. Ward, and E. Rundensteiner. "Structure-Based Brushes: A Mechanism for Navigating Hierarchically Organized Data and Information Spaces". IEEE Transactions on Visualization and Computer Graphics; April-June 2000.

[3] A. Inselberg. "Multidimensional Detective". IEEE Symposium on Information Visualization; October 1997.

[4] X. Wu, K. Subramanian, and L. Zhang. "GenExplore: Interactive Exploration of Gene Interactions from Microarray Data". Computer Science Department. The University of North Carolina at Charlotte. ICDE 2004; March-April 2004 (To Appear).

[5] W. Schroeder, K. Martin, and B. Lorensen. The Visualization Toolkit: 3rd Edition. Kitware Inc. Clifton Park, New York, USA. 2002. <http://www.kitware.com>

[6] X. Wu, D. Barbara, L. Zhang, and Y. Ye. "Gene Interaction Analysis Using k-way Interaction Loglinear Model: A Case Study on Yeast Data". ICML03 Workshop; August 2003.

[7] P. Gwynne and G. Page. "Microarray Analysis: The Next Revolution in Molecular Biology". Science Magazine. 6 August 1999. 15 December 2003. <http://www.sciencemag.org/feature/e-market/benchtop/micro.shl>

[8] P. Gwynne and G. Heebner. "Technologies in DNA Chips and Microarrays I". Science. 297: , 2001. <http://www.sciencemag.org/feature/e-market/benchtop/dnachips.shl>

[9] P. Gwynne and G. Heebner. "Technologies in DNA Chips and Microarrays II". Science. 296: , 2001. <http://www.sciencemag.org/feature/e-market/benchtop/technologies_dnachips.shl>

[10] P. Gwynne and G. Heebner. "DNA Chips and Microarrays Part 1". Science. 292: , 2001 <http://www.sciencemag. org/feature/e-market/benchtop/technologies_dnachips.shl>

[11] A. Ben-Dor, R. Shamir, and Z. Yakhini. “Clustering Gene Expression Patterns”. Journal of Computational Biology. (3/4):281–297, 1999.

[12] Y. Xu, V. Olman, and D. Xu. “Clustering Gene Expression Data Using a Graph-Ttheoretic Approach: an Application of Minimum Spanning Trees”. Bioinformatics.18(4):536–545, 2002.

[13] E. Hartuv and R. Shamir. “Clustering Agorithm Based on Graph Connectivity”. Information Processing Letters.76(4-6):175–181, 2000.

[14] R. Shamir and R. Shamir. “Click: A Clustering Algorithm for Gene Expression Analysis”. In Proceedings f the Eighth International Conference on Intelligent System for Molecular Biology (ISMB00), 2000.

[15] E. Andersen. The Statistical Analysis of Categorical Data. Springer Verlag, Berlin, Heidelberg. 1994.

[16] B. Schneiderman. “Tree Visualization with Tree-Maps: 2-d Space-Filling Approach”. ACM Transactions on Graphics. 11(1): 92-99. 1992.

[17] E. Lander, et al. “Initial Sequencing and Analysis of the Human Genome”. Nature. 409:860-921, 2001.

[18] J. Venter, et al. “The Sequence of the Human Genome” Science. 291:1309-1351, 2001.

[19] C. Jeong and A. Pang. “Reconfigurable Disc Trees for Visualizing Large Hierarchical Information Space”. Proceedings of IEEE Information Visualization 1998. October 19-20, Research Triangle Park, NC. 1998.

[20] A. Inselberg and B. Dimsdale. “Parallel Coordinates: A Tool for Visualizing Multi-Dimensional Geometry”. Proceedings of IEEE Visualization 1990, October, San Fransisco, CA. Pages 361-375. IEEE Computer Society, 1990.

[21] Y. Ye, et al. “GenExplore: Interactive Exploration of Gene Interactions from Microarray Data”. Department of Computer Science, University of North Carolina at Charlotte. 2003.