TAO WU
An Extensible Framework for Developing Visualization Software for Gene Expression Data
(Under the Direction of EILEEN KRAEMER)

A design architecture of a software system that analyzes and visualizes gene expression profiles is proposed. Implementation of core elements in the system is provided and a functional system prototype is developed. This system is capable of visualizing data through a scrollable user interface (UI) that only uses a small amount of memory. Current system includes implementation for viewing data as a rooted tree or colored heat maps and includes support for clustering, filtering and printing.

INDEX WORDS: Gene, Expression, Bioinformatics, Visualization.

AN EXTENSIBLE FRAMEWORK FOR DEVELOPING VISUALIZATION SOFTWARE FOR GENE EXPRESSION DATA

by

TAO WU

B.S., Sun Yat-Sen University, China, 1995

M.S., University of Maryland, Baltimore County, 1999

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirement for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2001

AN EXTENSIBLE FRAMEWORK FOR DEVELOPING VISUALIZATION SOFTWARE FOR GENE EXPRESSION DATA

by

TAO WU

Approved:

Major Professor: Eileen Kraemer

Committee: Daniel Everett
Thiab Taha

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
December 2001

DEDICATION

To my parents, who have given me their love and support for all these years.

ACKNOWLEDGEMENTS

Thanks to all my committee members for their support and guidance for my project. I would also like to thank Jinhua Guo and Mihail Tudoreanu for all their wonderful suggestions.

TABLE OF CONTENTS

Page

CHAPTER 1

BACKGROUND

With the rapid development of genomic science, immense amounts of biological data have been produced. This data provides unprecedented opportunities for advancing biological research. However, the sheer amount of this data also poses challenges to researchers who wish to analyze the data and make new discoveries.

One of the new methods in genomic research is to study the effects of stimuli on living cells manifested as changes in expression levels of genes. First developed by Schena, *et al* [1] in 1995, the DNA micro-array technology has been widely adopted by biology researchers[2]. This technology has proven to be powerful in tasks such as classifying previously unknown genes, discovering new functions of known genes and identifying novel drug targets[3].

In the microarray procedure, DNA sequences representing genes in an organism are first spotted onto glass slides or nylon membranes to form a micro-array. Then, mRNAs, the expressed form of genes, are extracted from the organism under a certain experimental condition and labeled with one of two fluorescent dyes (the green dye), while mRNAs extracted from the untreated organism are labeled with the other (the red dye). The two batches of mRNAs are mixed and applied to the DNA microarray.

The mRNA species then bind to their DNA counterparts on the microarray, and thus attach the fluorescent labels onto the target genes. Upon laser activation, the two labeled mRNAs each emits a different color light, indicating that the target gene is

expressed. Green or red fluorescent filters are the0n applied to measure the intensity of each color light for every spot on the microarray.

Experimental conditions are planned such that the amount of any DNA species on the microarray is able to absorb all mRNA that can bind to it. Thus, the intensity of the fluorescent light is an indicator of the amount of mRNA present for the DNA sequence (gene) in that particular position on the DNA microarray. After normalization, the ratio of the two light intensities (green vs. red), or the amount of mRNA for that gene under the experimental condition vs. that under the control condition, becomes an indication of whether the expression of that gene has increased or decreased (Fig. 1). The entire hybridization result of one micro-array would then be a snapshot of the expression profiles of all available genes in the organism under this particular experimental condition.

ORF	t0 g/r ratio	t0.5 g/r ratio	t2 g/r ratio	…
YHR007C	0.82	0.65	0.36	…
YOL109W	0.54	1.16	0.62	…
…	…	…	…	…

Fig. 1 Sample result of a gene expression profile experiment

Typically, the number of genes studied in a microarray experiment is quite large. For example, we used the yeast expression data set from Chu, *et al*. [4], which contains the fluorescent readings of over 6,000 genes under 9 experimental conditions. When researchers study other more complex organisms, the number of genes may run much higher. With such information in mind, it becomes obvious that computer aided visualization and interaction can be very useful tools in understanding the results of gene expression experiments and in directing further analysis of the experimental data.

In this thesis, we describe the design architecture and implementation of a software system, the ExpressionProfiler, for visualizing and analyzing gene expression data. This software has been developed to visualize a large amount of data and the design architecture has been chosen to allow easy future functional extension to the software.

CHAPTER 2

METHODS FOR EXPRESSION DATA ANALYSIS

Genes in an organism are not isolated entities but rather, they interact with each other. Most often, genes that are functionally related express in similar patterns. For example, under a given condition, genes in the same metabolic pathway often have either universally elevated or repressed expression levels, depending on whether the pathway is activated or inhibited under certain conditions.

The most common analysis performed with expression data is clustering, the grouping together of genes with similar expression patterns. This is helpful in several ways:

1. If a gene whose function is unknown is closely grouped in a cluster with genes known to function in a certain metabolic pathway, it may indicate that the new gene is involved in the same pathway too. This is called functional discovery.
2. If a drug treatment results in a global expression pattern similar to that pattern observed when disruption of a certain metabolic pathway occurs, it may indicate that some of the pathway components could well be targets for this drug. This is called drug target identification.
3. Fine grain categorization of diseases or other phenotypes can take advantage of expression profile analysis: Often, even for the same clinically diagnosed disease, several subtypes could exist that are indistinguishable, based solely on clinical syndromes.

However, an examination of the gene expression patterns for these subtypes often reveals visible differences. This helps doctors decide the best treatment for each individual case.

The first step in clustering is to define how to measure the distance between profiles. In other words, we must choose a distance metric.

Calculating Distances

The expression profile consists of expression ratios of each gene under different experimental conditions (Fig. 1). The distance between two profiles is basically a function of these expression ratios. Researchers may choose from among several different distance metrics. These different metrics focus on different aspects of the characteristics of the expression data.

Given two profiles *A* and *B,* each with n expression ratio values, and their *ith* expression ratio denoted by A_i and B_i, respectively, we list some of the distance metrics we considered:

1. Correlation Coefficient Distance Metric

The correlation coefficient[20] between profile *A* and profile *B* can be used as a measure of how closely related they are:

$$r = \frac{1}{n}\sum((A_i - \overline{A})/S_A)((B_i - \overline{B})/S_B) \qquad (1)$$

where $\overline{A}$, $\overline{B}$ are the mean values of *A* and *B,* respectively, and S_A, S_B are the standard deviations for all expression ratios of *A* and *B*.

The correlation coefficient ranges between -1 and +1. A correlation coefficient of 1 would mean that *A* and *B* change in exactly the same pattern, and a value of -1 would mean that they run in opposite directions. A value of 0 means that *A* and *B* are

independent (or that no correlation is detected). The closer to +1 the correlation coefficient between two genes, the closer they are considered to be to each other.

For example, if we have the following three expression profiles:

	t1	t2	t3	t4
A	1.1	1.2	1.3	1.4
B	1	0.9	0.8	0.7
C	2	4	8	16

Table 1 Sample profiles for calculating correlation coefficient

After calculation, we will see that $r_{A,b}$ = -0.75, $r_{A,C}$ = 0.71. Thus, $d_{A,B} > d_{A,C}$. This is because A and C show the same trend of increasing over the conditions, while B displays a trend of decreasing.

However, a researcher would see that A and B are actually closer together than A

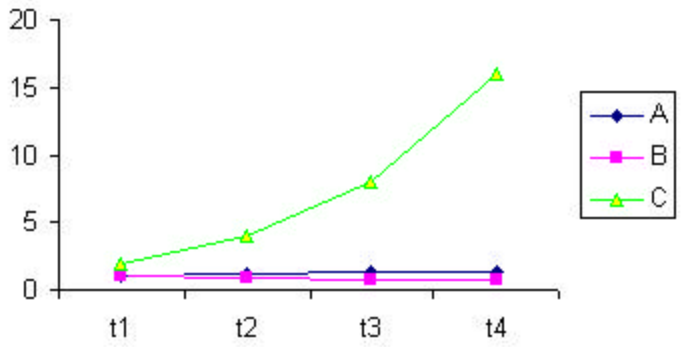

Fig. 2 Curve view for the sample profiles in Table 1

and C, after these values are plotted on a two dimensional coordinate system (Fig. 2), Thus, if a researcher wishes to focus on the value difference between the three profiles, correlation coefficient may not meet the needs of the researchers. Another metric may be more useful.

2. Euclidian Distance Metric

$$d_{A,B} = \sqrt{\sum (A_i - B_i)^2} \qquad (2)$$

The Euclidean distance metric evaluates the square of the difference between each pair of expression ratios in two given profiles before summing them up. It is directly derived from the method in measuring the distance between two points in an n dimensional space. For the example given in Table 1, the Euclidean distance metric would report $d_{A,B} = 0.91$, $d_{A,C} = 16.33$. Clearly, under this distance metric, $d_{a,B} < d_{A,C}$.

3. Hamming Distance Metric

The Euclidean distance metric, however, still has limits. Consider the following situation:

	t1	t2	T3	t4	t5
A	1.1	1.2	1.3	1.4	1.3
B	1.2	1.3	1.4	3.1	1.3
C	2	2	2	2	2

Table 2. Sample profiles for comparing Euclidean distance metrics with Hamming distance metric

For such an example, the Euclidean distances for the three profile pairs are:

$d_{A,B} = 1.70$, $d_{A,C} = 1.67$. Thus, $d_{a,B} > d_{A,C}$.

However, genes A and B have shown very similar expression ratios under most of the experimental conditions, while gene C differs from A and B under all observed conditions. Thus, a researcher might consider that gene A and B resemble one another more than B and C (Fig. 3).

To accommodate such preference from researchers, we employed the Hamming distance metric.

Hamming distance was originally developed to measure the difference between two strings of 0s and 1s[5], where it sums up the number of positions in the two strings

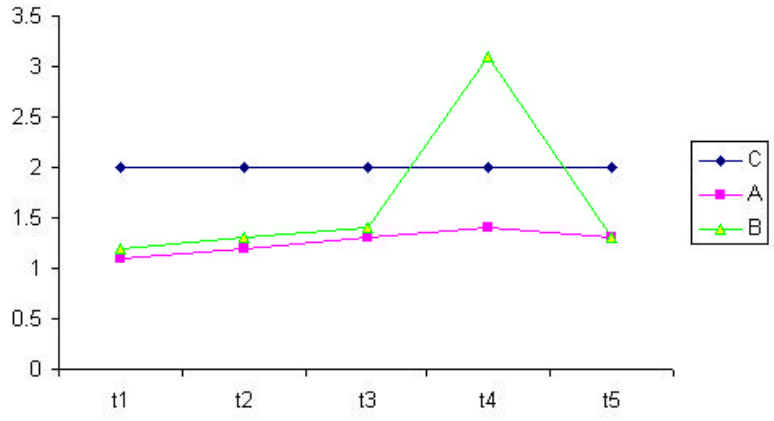

Fig. 3 Curve view for the sample profiles in Table 2.

that are different. We make a slight modification on this function to make it applicable to our expression data.

$$d_{A,B} = \sum \left(\left| A_i - B_i \right| / \max(A_i, B_i) \right) \qquad (3)$$

If $Ai = Bi$, then the contribution of the ith variable to the total difference is 0. On the other extreme, if $Ai >> Bi$ or $Ai << Bi$, then the contribution is close to 1, indicating maximal difference between these two positions. The contribution of any position to the overall distance is limited to between 0 and 1. In this way, a small number of significantly different variables will not be able to dominate the overall distance value between the two objects.

In the example in Table 2, our Hamming Distance function would give the following: $d_{A,B} = 0.78$, $d_{A,C} = 1.85$. Thus, $d_{a,B} < d_{A,C}$.

Clustering Methods

After deciding the distance metric, the next step in clustering expression data is to choose a clustering algorithm. The three most often used clustering algorithms in gene expression data analysis are UPGMA[6], K-Means[7] and Self-Organizing Map[8].

UPGMA stands for Unweighted Pair-Groups Method Average. This method has been extensively used in phylogenetic research to order species into an evolutionary tree structure and is commonly used in clustering gene expression data. It has proven to be useful in analyzing gene expression data[3, 4, 9-11]. However, this is also a clustering method where we find space for improvement in existing visualization software systems.

As will be described in later chapters, we have implemented the UPGMA algorithm and provided visualization for its results. Our design architecture has also allowed for the addition of new clustering algorithms, such as K-means clustering and SOM clustering, as well as the visualizations that are specific to them. However, although implementation and visualization of these other clustering methods would be useful in gene expression analysis, they are beyond the scope of this thesis.

The basic UPGMA algorithm runs as follows:

1) Compute all pair-wise distances between all objects and fill in a distance matrix with these distances. At this first step, all objects are assigned to individual clusters each of size n_k=1. These initial clusters become the leaves of the resulting tree.

2) From the distance matrix, find two clusters, *i* and *j*, that have the least distance between them.

3) Create a new cluster *(i, j)* and assign $(n_i + n_j)$ as its size.

4) Connect clusters *i* and *j* to the new cluster *(i, j)*. The two old clusters become obsolete and the new cluster acquires a set of distances to the remaining clusters according to the following rule, where *D* denotes the distance function between clusters:

$$D_{(i,j),k} = \frac{n_i}{n_i + n_j} D_{i,k} + \frac{n_j}{n_i + n_j} D_{j,k} \qquad (4)$$

5) Delete the rows and columns represented by the old cluster *i* and *j* and add a new row and a new column for the new cluster *(i, j)*. After this step, the number of clusters is reduced by one.

6) Repeat step 2 to step 5 until there is only one cluster left.

7) Output the resulting tree.

The basic UPGMA algorithm, also used in our implementation, requires $O(n^2)$ space to store the distance matrix, and $O(n^2)$ time to execute, since it needs to complete n-1 iterations and does O(n) work to update the distance matrix at each iteration.

In practice, the space requirement for the basic algorithm could be reduced to about $(n^2 - n) / 2$, by storing only the upper triangular half of the distance matrix. However, this can still be problematic.

Consider the size of the data that we used to test ExpressionProfiler. The entire data set contains about 6,000 genes and their expression ratios under multiple experimental conditions. Thus, on a Intel based workstation, for example, clustering the 6,000 genes would require $6{,}000 \times 6{,}000 \times 4 \ / 2$ bytes, or roughly 68MB of RAM for storing the distance matrix alone, provided that we store each distance value as a floating point number. For a general workstation with 128MB RAM, we are faced with the problem of consuming much of the available memory.

The UPGMA algorithm dictates that we must somehow update the distance matrix after each iteration. This makes reducing running time difficult. However, we can reduce the memory requirement.

For example, we could choose not to store the matrix in memory and compute the distance from membership information of the two clusters to be joined. Such a distance function can be deduced from formula (4).

Let *d(p, q)* denote the distance between object *p* and *q* at the leaf level and C_i, C_j denote cluster *i*, *j*. Then

$$D_{i,j} = \frac{1}{n_i \times n_j} \sum_{p \in Ci} \sum_{q \in Cj} d(p,q) \qquad (5)$$

This scheme could reduce the space requirement to only O(n) if we only store the membership information of the clusters, for example, by using nested parenthesis to store the tree structure in the Newick format[12], which we describe in Chapter 3. However, at each step, we must calculate the distances between every possible pair of clusters. The average running time becomes $O(n^2)$ at each step and the overall running time increases to $O(n^3)$.

At least one variant of the above scheme for implementing UPGMA clustering has been proposed that uses O(n) space. However, the cost saved by not storing the entire distance matrix is paid back in lengthened running time[13].

CHAPTER 3

RELATED WORK

Because of the large size of expression profile data, one common problem in visualizing it is how to present the data that allows easy viewing of different parts of it in detail and at the same time, provides a view from a distance to display the overall characteristics.

Over the few years since the invention of micro-array technology, numerous software systems have been developed to address this problem in different ways. We now describe several well known software systems in gene expression analysis.

TreeView

TreeView[14] was developed by Eisen at Stanford University and is a publicly available software system for viewing expression data. It is probably the earliest and the most well known tool for expression data analysis.

TreeView offers a compact binary tree view for hierarchical clustering of both genes (rows) and conditions (columns) and a color grid to represent the value of each reading of a gene under a condition. Correlation between the tree leaves and the color grid cells is maintained when the user interacts with a scroll bar in the UI. However, because the software displays the entire clustering result in a very compact view, there is not enough detail near the leaf level of the tree when the amount of data exceeds a few hundred genes on multiple experimental conditions. The user can zoom in but is

restricted to selecting some of the leaves and seeing a zoomed-in view in a separate panel.

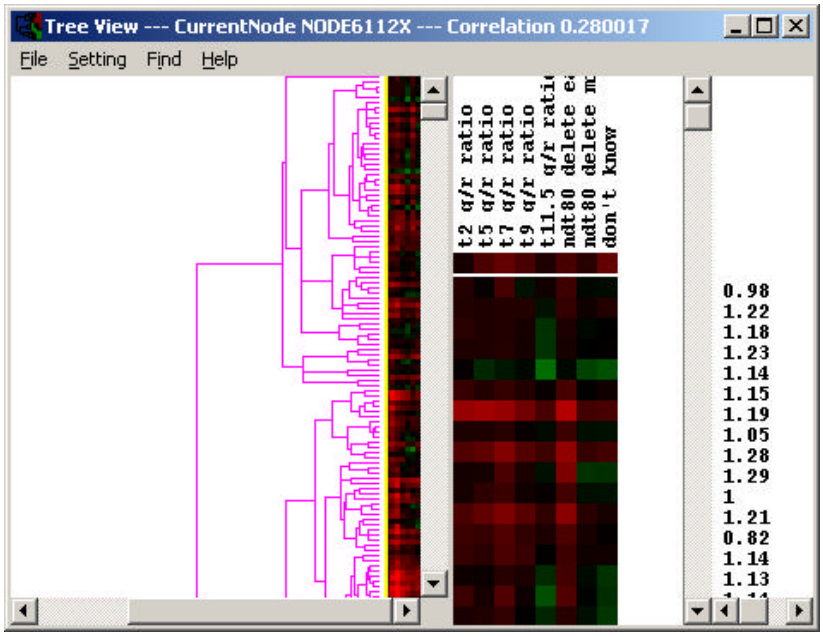

Fig. 4 Screenshot of TreeView

TreeView does not support other user interactions, such as editing the data before clustering so that the user can focus on the part of data in which he or she is most interested, or manipulating the resulting tree structure (for example, reversing a subtree) to have a different view of the data. All this limits TreeView's use in analyzing the gene expression profile.

Commercial software systems usually do provide zoomable pictures. We now describe three of them. These are systems developed by major bioinformatics software companies for expression data and have been discussed in software reviews[15].

GeneSpring

Silicon Genetics's GeneSpring[16] offers a host of visualization and analysis functions, including the display of the position of a gene on a chromosome or plasmid,

the expression level of that gene and also the result of the UPGMA clustering. When visualizing the UPGMA clustering algorithm, GeneSpring displays the entire tree structure in a fixed-size area and allows the user to zoom in on a selection.

This approach, however, has two major drawbacks:

1.There is no information about either the gene names or the experimental conditions associated with the tree structure. It is not helpful to the user to know only that there exists a subtree of a certain shape, without such labeling information.

2. When the user zooms in on that particular area, he/she gets only a view of that area and loses the global view and thus the context of the zoomed in view.

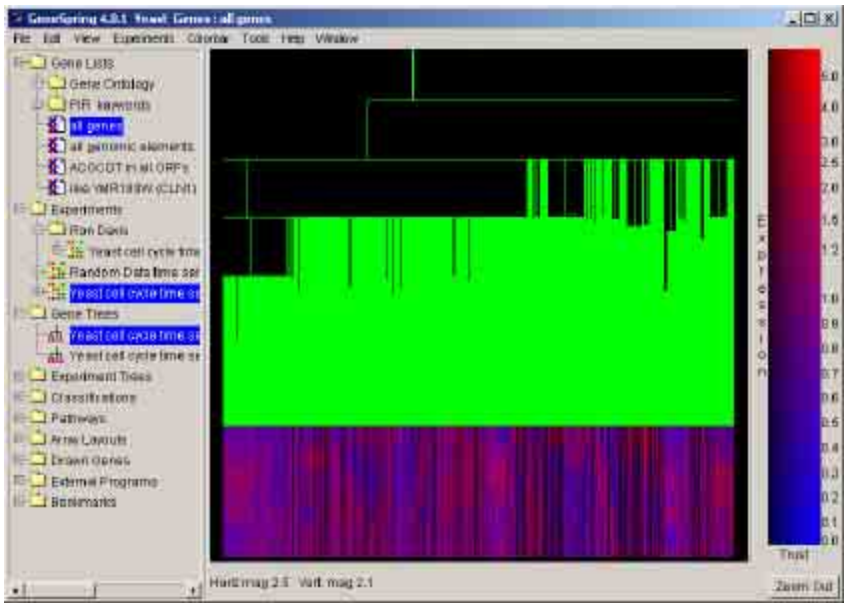

Fig. 5 GeneSpring screenshot after zooming out

GeneSpring does not support editing the data before clustering and the user ‘s only interaction with the cluster tree is zooming in on a selection. This, again, limits the user’s ability to focus on a particular part of the data and to explore the result of the clustering algorithm.

DecisionSite

Spotfire's DecisionSite[17] is deployed by a number of major biotech and pharmaceutical companies for gene expression data analysis. It supports a wide range of

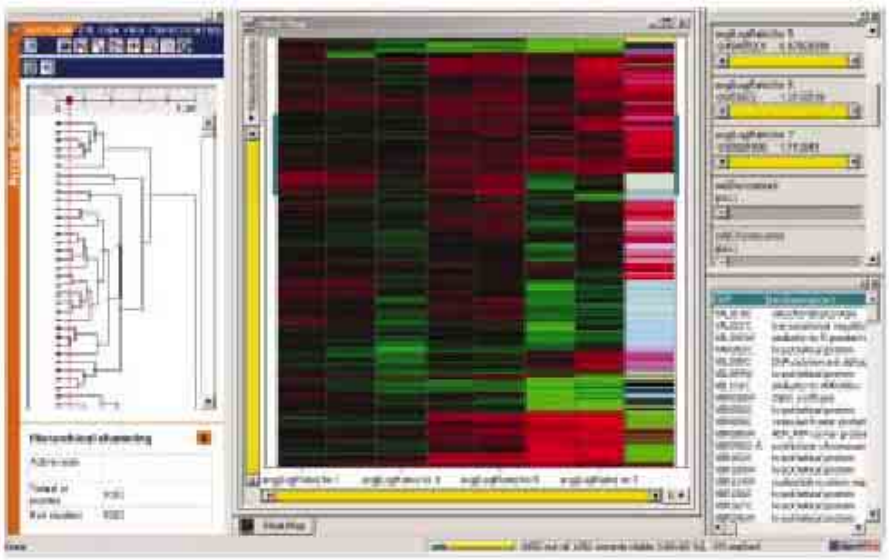

Fig. 6 Screenshot for DecisionSite

distance metrics and clustering methods and allows users to perform complex queries on the data. DecisionSite is a powerful query tool for users to select the genes whose profile matches a user-defined pattern. It also provides visualization of different clustering results. However, in its visualization, the user is limited to making a selection within the data and to view information associated with the selection. The user is unable to manipulate the graphics elements in the visualization, for example, to select a subtree in the clustering result and reverse it, which would be useful in assisting the user in exploring the clustering results.

DecisionSite provides a color grid to show the expression data itself. However, the color grid is displayed without gene name or condition information. DecisionSite also provides a visualization of the hierarchical clustering result as a tree structure, but the tree

structure is displayed, again, without gene name information and does not scroll along with the color grid, which makes it difficult for user to relate a cluster in the tree to the particular red and green stripes in the color grid that correspond to that cluster.

Although DecisionSite provides the ability to zoom in on the tree representation, it only applies to zooming on the vertical dimension along the breadth of the tree. Horizontal zooming capability is not supported in the current version. This only half-solves the problem of losing detail near the leaf level, since the tree structure could still be over-compressed along its height.

GeneExplore

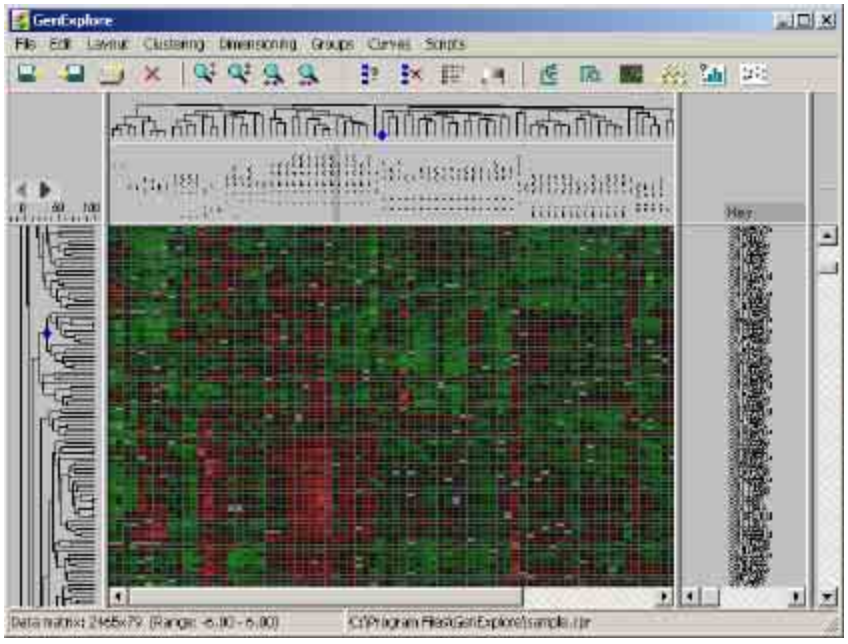

Fig. 7 Screenshot of GeneExplore after applying the UPGMA clustering algorithm

GeneExplore[21] is developed by Applied Maths and offers tree structures that are nicely correlated with the color grid. It also displays the associated gene names and conditions, and supports zoom functionality on the color grid. When the user zooms on either the X-axis or the Y-axis of the color grid, the tree structure automatically zooms accordingly.

However, the application stops short of providing independent zooming capability for the clustered tree. Thus, the horizontal tree can only be zoomed on the X-axis, while the vertical one can only be zoomed on the Y-axis. This method of displaying the entire height of the tree within a fixed area would result in a too compact tree structure and loss of detail.

Summary

While most software systems for gene expression visualization try to address the problem of visualizing the data through a scrollable graphical interface, few of them have allowed the user enough flexibility in zooming on a visualization area to obtain detail of the data on different levels, while at the same time, maintaining the correlation between different aspects of the visualized data.

We attempt to address this aspect of the visualization problem through the design of our software system, the ExpressionProfiler. ExpressionProfiler provides very flexible zooming on the graphical representation of the data and supports numerous user interactions. We achieve independent as well as coordinated zooming and scrolling of different aspects of the data visualization, which is not available in the other software systems we have examined. Our design architecture also supports the addition of new visualization methods as well as the possible user interactions associated with those methods. ExpressionProfiler is be described in the following chapters.

CHAPTER 4

USER WALKTHROUGH

When a user starts ExpressionProfiler, the natural first step is to load in a data file representing the result of a microarray experiment.

Fig. 8 Graphical interface after user selects the “File” menu item

ExpressionProfiler expects that the expression data be in a simple format: entries in each row are the ratios of the expression level of a gene under the experimental

Orf	Condition1	Condition2	Condition3	Condition…
Gene1	V11	V12	V13	V1…
Gene2	V21	V22	V23	V2…
…	…	…	…	…

Fig. 9 Input file format for ExpressionProfiler

condition to the expression level of the same gene under the control condition. These expression ratios are tab delimited and each row ends with the new line character, as seen in Fig. 9.

After the file is read in, ExpressionProfiler creates a grid view of the data. The grid view is divided into three areas: an area that displays each gene name, an area that displays each experimental condition name, and one that displays the values of the expression ratios as color grid cells.

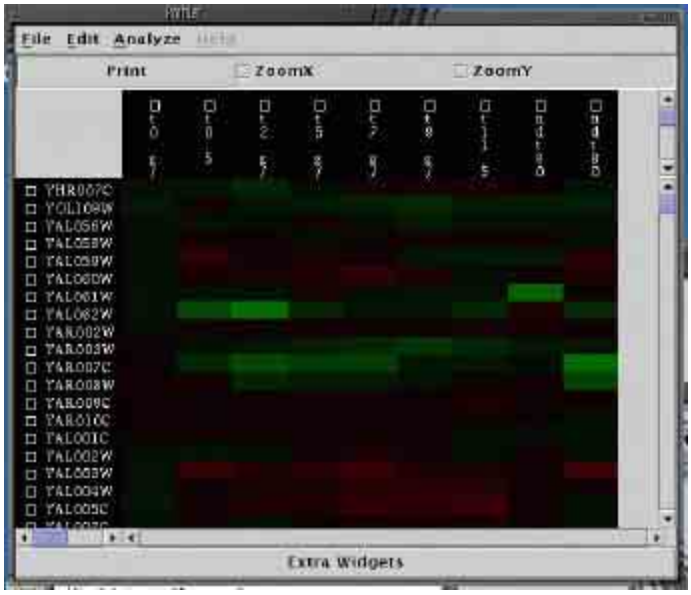

Fig. 10 GridView after data file is loaded

Each of these three areas is independently scrollable. For example, the gene name area could be scrolled horizontally and the condition name area could be scrolled vertically, to reveal the full names of the genes and experimental conditions. When such scrolling occurs, the color grid view does not change and thus, visual correlation between a gene's name and expression ratios of that gene and the correlation between a condition's name and expression ratios for all genes under that condition are maintained.

Scrolling is also cooperative: when the gene name area is scrolled vertically, the area for the color grid cell is updated automatically. Horizontally scrolling of the condition name area results in a similar update to the grid area. In this way, the correlation between gene name, column name and the corresponding grid cell is always maintained.

GridView:

User interactions associated with the GridView are as follows:

Select:

Data editing is often preceded by making a selection in the visualization. The user can select either one row/column or a contiguous range of rows/columns of the table. He then can choose to move the selection to another position in the table, reverse the selection or delete the selection.

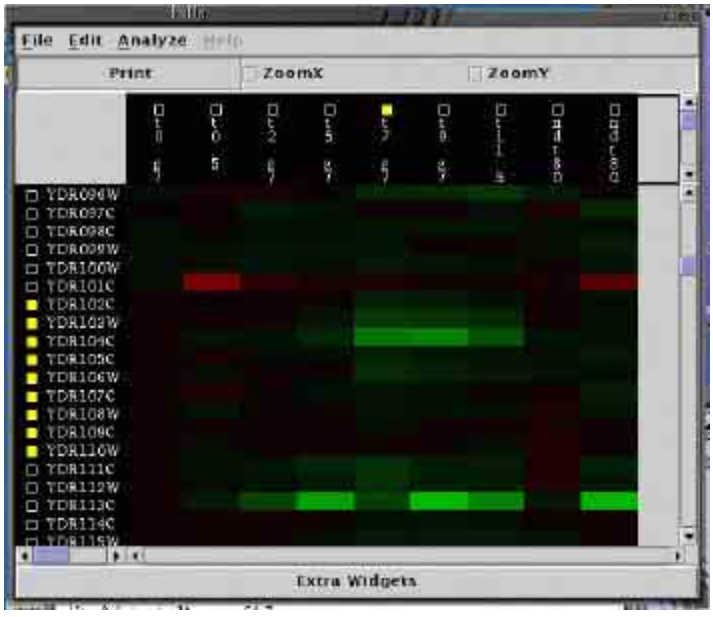

Fig. 11 GridView after user selects multiple rows and columns

These operations allow the user to edit the data before starting the clustering process and are easily accessible by clicking the "Edit" button in the menu bar. Each of

these operations is accompanied by the automatic re-ordering of the color grid cells corresponding to the row and/or columns being worked on.

If the user selects an area in the area showing color grid cells, a new window that displays the selected part of the table as a series of line curves is created. Such a curve view, as seen is Fig. 12, is useful in making direct comparison of the expression levels between genes.

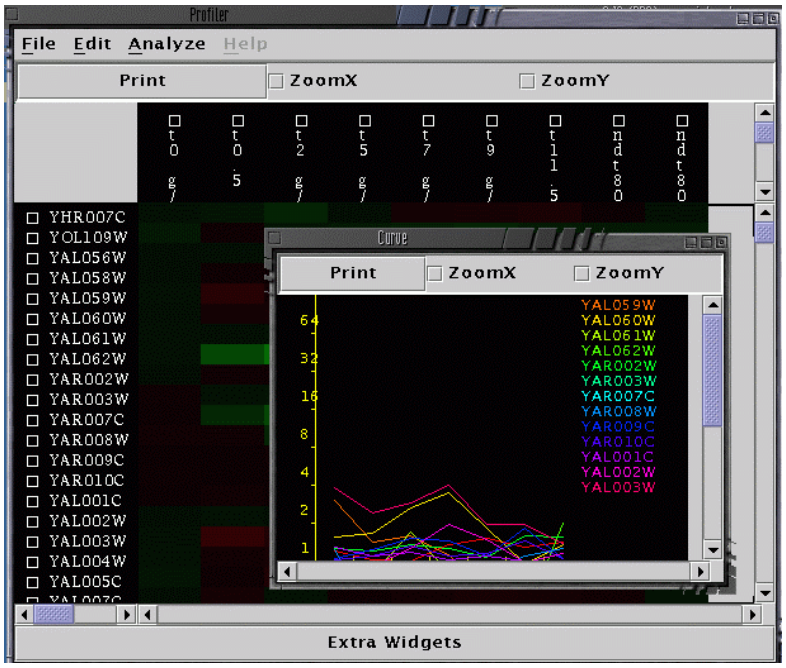

Fig. 12 CurveView after user selects an area in the color grid

Filter:

Sometimes, the user is interested only in genes that exhibit significant activation or repression. We support filtering genes whose overall expression changes exceed a user-given threshold. Currently, the implemented filter is the Root-means-square (RMS) filter, as is used in the research by Chu, *et al*[4].

RMS is calculated for each gene as shown below, where X_i is the expression ratio of a gene under the *ith* condition:

$$rms = \sqrt{\frac{\sum \log_2^2 X_i}{n}} \qquad (6)$$

If the RMS value of a gene is greater than the cutoff value given by the user, that gene is retained; otherwise it is excluded from further analysis.

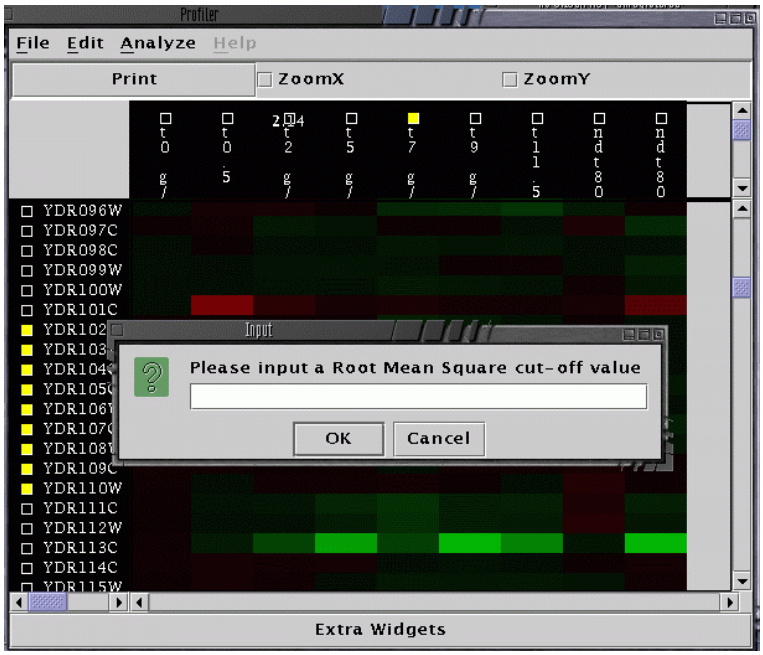

Fig. 13 Filter dialog

Taking the logarithm of X_i gives equal weight to ratios bigger than 1 and those less than 1. For example, a 2-fold increase in expression level (a ratio value of 2) evaluates to 1 after logarithmic conversion, while a 2-fold decrease in expression (a ratio value of 0.5) evaluates to -1. After squaring, they both result in the same value 1. RMS calculated this way is an indication of the degree of change in a gene's expression. By filtering the data against different RMS cut-off values, the user selects the part of the

data that are most interesting to him or her. This is another form of data editing that ExpressionProfiler supports.

Save to File:

After filtering and clustering, the user can save the resulting table to a file for exchange, comparison, or to use as input to another tool.

Print:

We support printing of a view through the simple click of "Print" menu item under "File". The current print capability is limited and can only print the part of the view visible on the screen and does not support printing the entire logical view on multiple pages. However, the user could scroll through several screens and manually print out the screenshots.

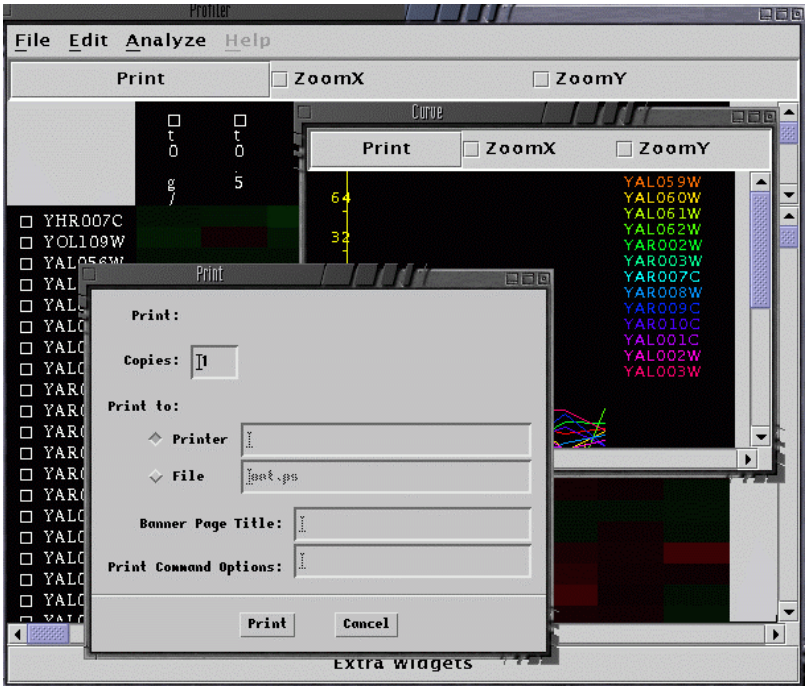

Fig. 14 Print dialog

Zoom:

By selectively checking two zoom boxes, the user can zoom in or zoom out on an area, on either axis, or on both axes. When zoom is selected, the left click of the mouse triggers a zoom-in action, which attempts to center the zoomed in view at the point of the click. Zooming out is as simple: the user just needs to right-click.

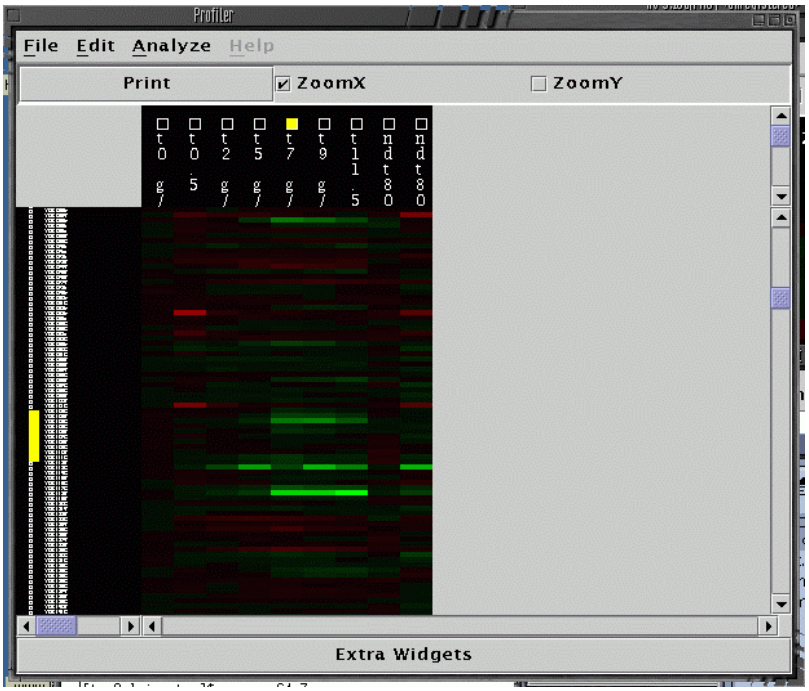

Fig. 15 GridView after zooming out on X and Y axis

ExpressionProfiler attempts to zoom the picture and then position it so that the point where the user clicked appears at the center of the area in the View. This helps the user in orienting himself after the zooming and is a unique feature that we have not seen in other software systems.

Find:

The user might be interested in studying the expression changes for a particular gene. To find a gene, the user first selects the area that displays the gene names, by clicking in it. Then, she opens the "Edit" menu and selects "Find" and types in the gene

name. ExpressionProfiler attempts to find a gene whose name matches with the user input. If such a gene exists, ExpressionProfiler scrolls the entire view and attempts to position it so that the target gene is at the top of the view.

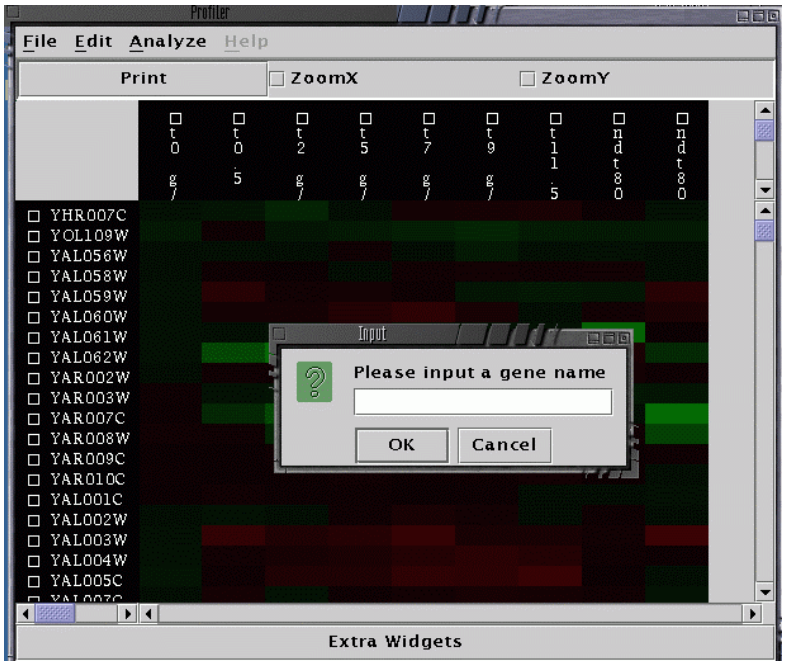

Fig. 16 "Find" dialog

Undo:

ExpressionProfiler provides the ability to undo certain kinds of operations (currently Delete and Filter). Such a function is accessible through the menu item "Undo" in the "Edit" menu.

Cluster Analysis:

After editing the data by deleting and filtering, the natural next step is to start the clustering process. The user needs to click on "Analyze" and choose "Cluster". A new window is then generated with a list of the available clustering algorithms.

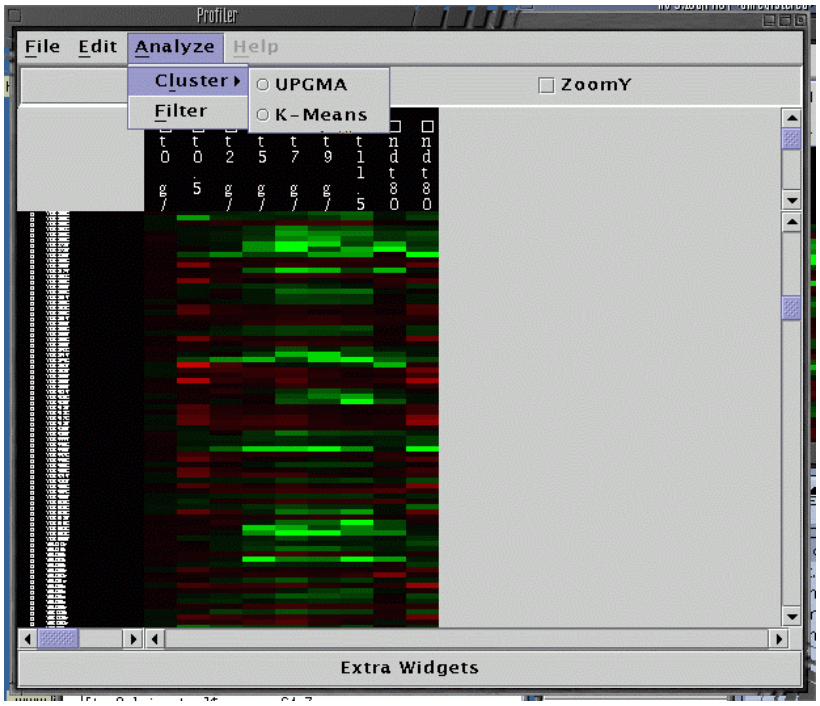

Fig. 17 Cluster Options

After the user selects an algorithm, additional window(s) are displayed and the user is prompted for the necessary information (options) for that particular clustering

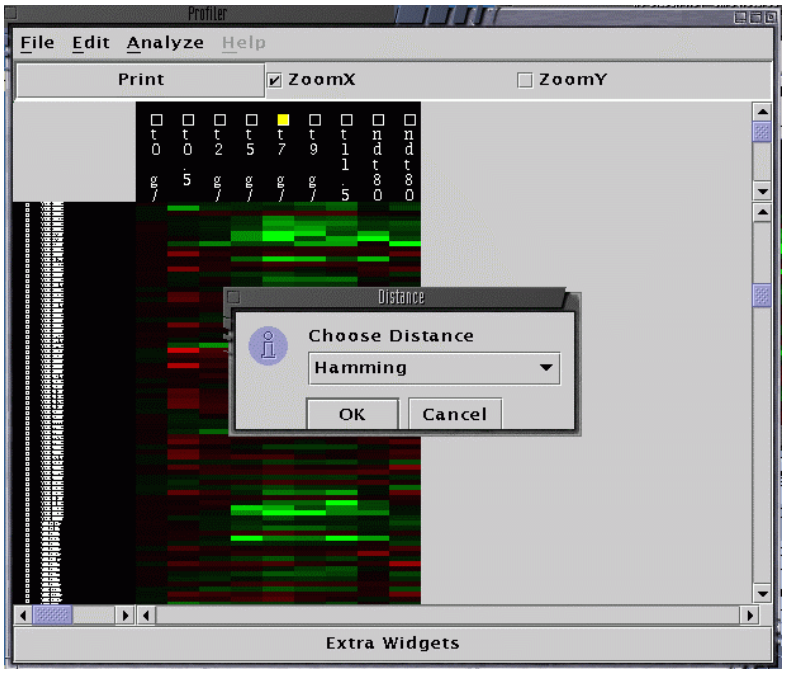

Fig. 18 Choosing distance metric

algorithm. For example, in Fig. 18, ExpressionProfiler asks the user for a choice of distance metric.

When ExpressionProfiler has collected all necessary information, it initiates the clustering process and a progress dialog is created to convey the progress to the user, in case the execution of the algorithm causes considerable delay in perceivable response time.

TreeView:

When clustering is finished, a new view, the TreeView, replaces the GridView in the ExpressionProfiler.

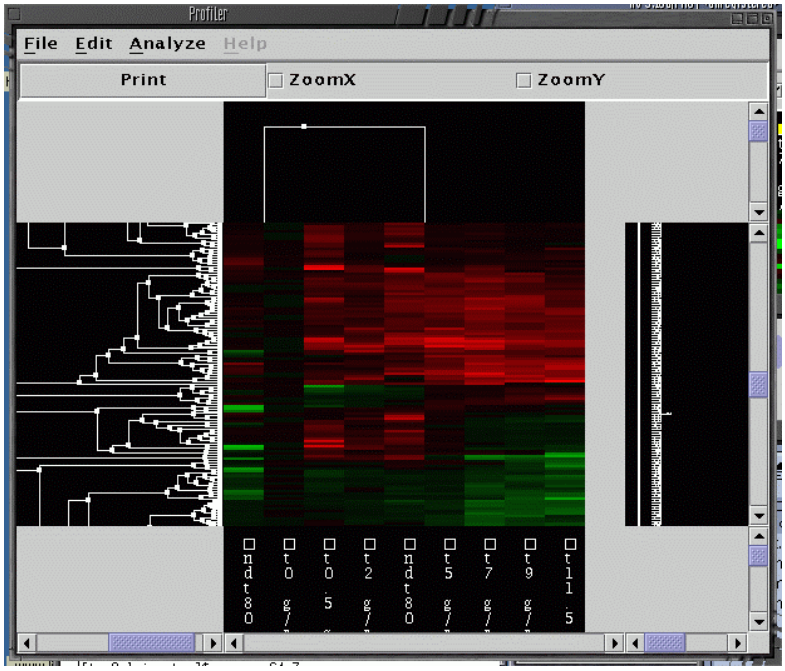

Fig. 19 TreeView after zooming out on Y axis

We have implemented the popular hierarchical clustering algorithm, UPGMA, which generates a binary tree representing the cluster relationship between individual elements and/or element clusters. A tree structure represents the clustering result of the

rows (genes) and another tree structure represents the clustering result for the columns(experimental conditions).

For such a tree view, the following user interactions are possible:

Zoom:

Again, the user is empowered with the capability to zoom in and out on the tree structure and explore the clustering result at different levels of detail.

Select:

By clicking on a node, the user can select the entire sub-tree rooted at that node. Then, he can reverse the entire subtree (obtaining a mirror image of the subtree) by choosing "Reverse" under the "Edit" menu. Reversal of a sub-tree also triggers the reversal of the rows/columns corresponding to the leaves in the sub-tree.

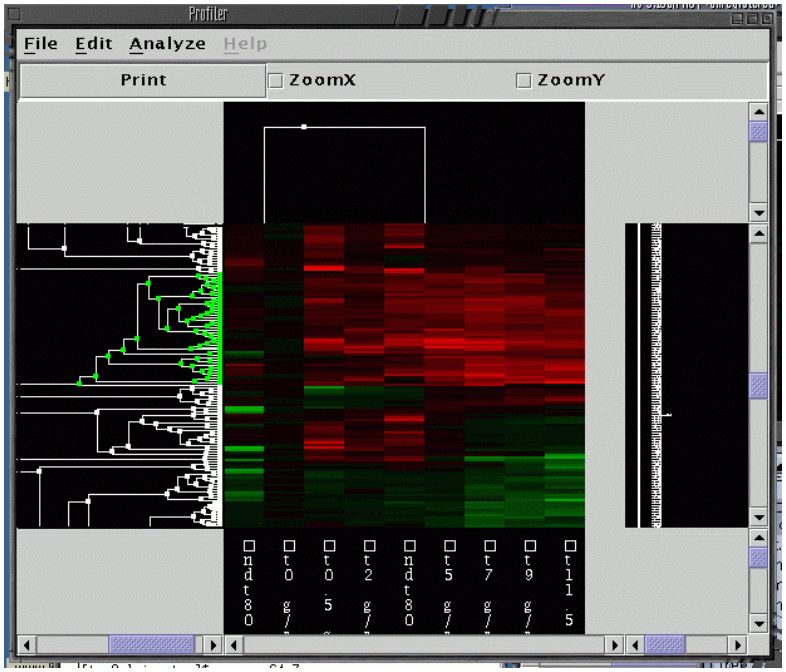

Fig. 20 Selection in TreeView

Also, as in the grid view, when the user selects an area in the color grid, a new window is created with the information in that part of the table displayed as a collection of line curves.

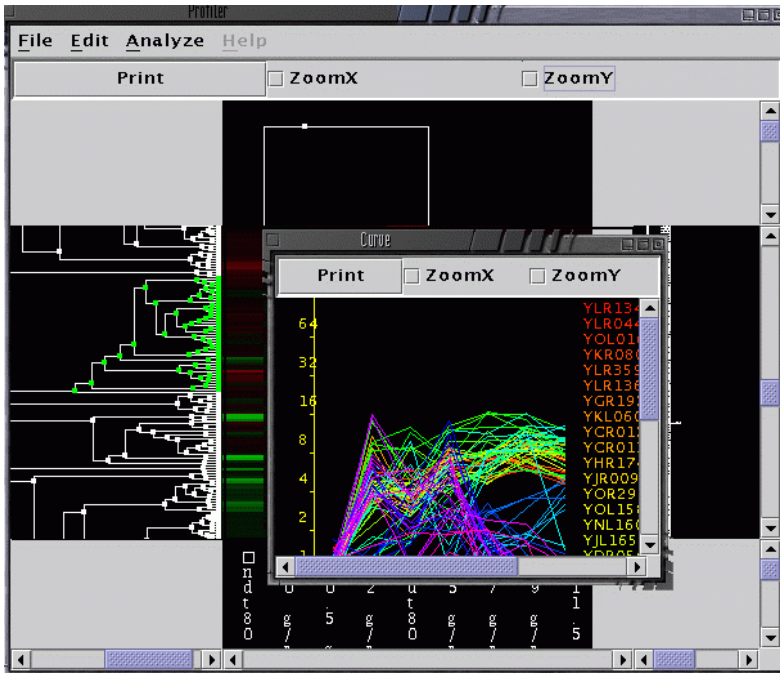

Fig. 21 CurveView after user selects an area of heat map in TreeView

Find:

As in the grid view, the user can find a gene by giving a substring contained in that gene name. Correlation across different displays (row names, column names, color grid cells and tree structures) is well maintained even when the tree view has scrolled to the position of the found gene.

Save Tree to File:

We support saving the tree structure obtained from the clustering process in the popular Newick format.

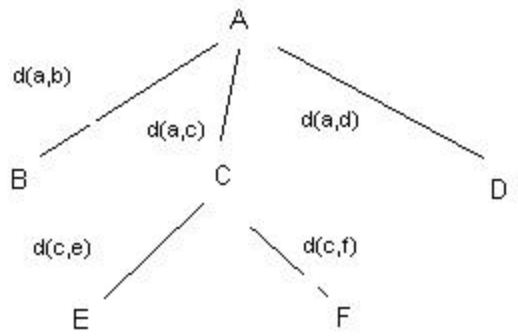

Fig. 22 A tree example

For example, the above tree could be represented with nested parentheses as follows, where the letter A to F denote the nodes in the tree and d(x, y) denotes the distance between node x and node y:

(B:d(a, b), (E:d(c, e), F:d(c, f)):d(a, c), D:d(a, d)).

The nested parentheses in the Newick format match the recursive characteristic of a tree. Tree structures in memory can be easily converted into files in the Newick format and vice versa. For the UPGMA clustering algorithm, the result is simply a binary tree.

CHAPTER 5

DESIGN ARCHITECTURE AND IMPLEMENTATION

Our software system, the ExpressionProfiler, addresses the zooming problem in a different way. We implemented a Buffer class, which encapsulates the concept and functionality of a buffered image. This Buffer class maintains a buffered window of the real image and executes client requests for drawing an area in the real image. Since the entire real image is not kept in memory, we also eliminate the limit on how big the real image can be and thus greatly increase the amount of data that we are able to visualize.

We have also designed a BufferManager class to manage concrete Buffer objects. The Buffer and BufferManager classes cooperate to visualize the data in an area in a window.

Below is a diagram of the major components in our system. Next we will explain the roles played by each of them and how they interact.

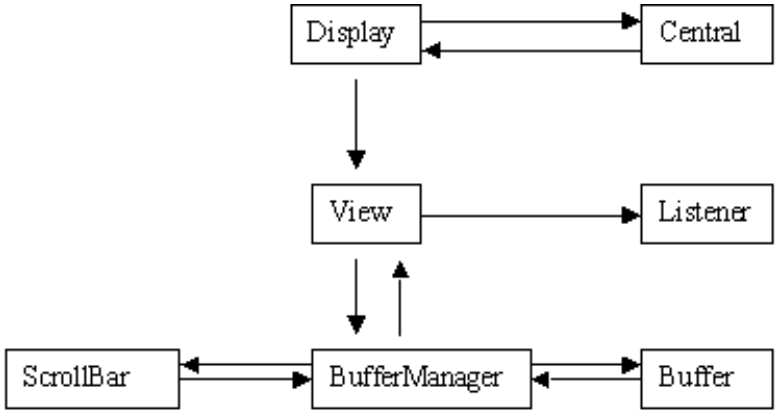

Fig. 23 Design Architecture for ExpressionProfiler

The user interacts with the visualizations in View through the interface of Listeners (mouse listener, for example). View provides a logical view on the highest level of abstraction for the profiling data. Each View realizes visualizations for multiple aspects of the same profiling data through Buffers managed by BufferManagers. A BufferManager controls the visualization in an area assigned to it by View.

Each BufferManager manages one Buffer, an object that actually prepares the visualization. Each BufferManager is also associated with a number of ScrollBars. The appearance of a ScrollBar gives an indication of the entire size of the visualization as well as the portion of this size that is currently visible.

The class Display serves as a wrapper for View and its purpose is mainly to house other possible visual aids relevant to the particular View object. Messages from View objects are sent through the Display objects and are eventually received by the class Central. Central then processes such messages and decides whether to forward them to some View through its Display wrapper or it may decide to discard it.

Next, we give a detailed description of these classes and their functions.

View:

Logically, a view of an expression profiling experiment may consist of different aspects of the data. For example, each set of expression data consists of expression ratios of thousands of genes under multiple experimental conditions. We must maintain some correlation between visualization of these different aspects for the same data. Thus, when the user scrolls to see ratios of a gene under different conditions, the name of the gene should always be on the same row with its ratios.

There is also another aesthetic consideration regarding scrolling: ideally, when the user scrolls the visualization in one area in a View, all related visualizations in other areas of the View should scroll simultaneously.

If we implement each of the different visualization areas in a View as a Java component, smooth scrolling would become difficult to achieve. This is because when the entire View is refreshed upon user interaction, there will be an order in which these drawing components are painted by the Java system and the user will see a noticeable delay in the refreshing of different drawing areas.

To solve this problem, we decided to exploit the fact that Java Swing components are double buffer enabled and implement View as a subclass of JPanel. We divide the area of this JPanel to display the different aspects of the expression data. When the user scrolls one sub-area, refresh requests for other related sub-areas are issued. Although there is still an order in which these different areas are refreshed (painted), the JPanel will not display the entire picture until it is completed. This hides the intermediate drawing process from the user and generates an impression of smooth scrolling.

Buffer:

The Buffer class is the ultimate recipient of drawing requests. It is responsible for drawing the requested area in the real picture onto clients. The Buffer class buffers a part of the real picture (bigger than the size of viewable area) and adjusts this buffered image to show different parts of the real picture when necessary.

The Buffer class provides most of the functionality for drawing and maintaining the buffered image. Concrete Buffer classes only need to define a small number of functions. Thus, we design different concrete Buffer classes that handle the drawing of

each aspect of the expression data (a class for drawing gene names, a class for drawing conditions, one for drawing the readings, for drawing the clustering relationship between genes, and so on).

Currently, we have implemented concrete Buffer classes to support three different Views: classes that display the gene names and conditions associated with readings, classes that display a binary tree structure for the genes and the conditions associated with the readings and classes that display the values of readings as line curves. However, more Views can be easily added by defining new concrete Buffer subclasses that render the data in other ways.

Each concrete Buffer class must minimally implement the method to draw a requested area in the real picture:

abstract void PrepareBuffer(int x1, int y1, int x2, int y2);

where *(x1, y1), (x2, y2)* are the coordinates that delineate the area in the real picture.

This method is relative easy to implement when the real picture follows a regular pattern, such as iterating through rows and column and drawing red or green rectangles. However, it becomes more complicated when it comes to calculating an area within a picture of a binary tree structure and drawing it onto the Buffer. Here, we developed a recursive algorithm to draw out any area within a binary tree structure.

Assume the leaves of the tree are positioned horizontally at the same height. We first assign each node a horizontal position that is between the rightmost child of its left subtree and leftmost child of its right subtree. This enables us to limit the search range while drawing the requested part of the tree structure.

To draw an area delineated by point (x1, y1) and (x2, y2) in the real picture, we use the following algorithm:

```
AREA: the area delineated by point (x1, y1) and (x2, y2)
LC: left child of a node
RC: right child of a node

DrawTree(Node b)
{
  if ( b is below the bottom of the area )
  {
    /* draw nothing */
  }
  else
  {
    if( b.Size() == 1 )
    {
      DrawNode( b );
    }
    else
    {
      DrawLinkToChildren( b );

      if( b is to the left of AREA )
      {
        DrawTree( b.RC );
      }
      else
      {
        if( b is within AREA )
        {
          DrawNode( b );

          DrawTree( b.LC );
          DrawTree( b.RC );
        }
        else
        {
          DrawTree( b.LC );
        }
      }
    }
  }
}
```

Fig. Code example for recursively drawing a binary tree

BufferManager:

Since we have to split the visible area in View to display these different aspects of the data, we must manage where we should draw each of the Buffer and to what size. We

wish to separate the management of drawing from the actual drawing action itself to minimize the complexity of the Buffer class and reduce the difficulty of future software maintenance. Thus comes the idea of the class BufferManager.

A BufferManager maintains its location and size and is managed by View. A View object may decide to change the location and/or size of a BufferManager(and thus where and how much visualization of the data is available to the user) upon user interactions. A concrete View object creates the BufferManagers and assigns them locations and sizes. Then, it passes all user drawing requests to the BufferManagers.

The BufferManager handles the drawing requests from View and passes the requests on to the Buffer object it manages. A BufferManager can manage any concrete Buffer object. In fact, a BufferManager does not know which Buffer subclass it is managing and only interacts with the concrete Buffer through the interface defined by the abstract Buffer class.

Each BufferManager is also associated with two ScrollBars, one horizontal and one vertical, to allow user interaction and respond to user's requests to see other parts of the real picture. Also, when the BufferManager changes size, the associated ScrollBars should adjust their sizes to reflect such changes.

We use the Observer design pattern[18] to capture the relationship between BufferManager (the Subject) and the ScrollBars (the Observer). This pattern minimizes the amount of knowledge these two classes must have about each other and allows these them to change independently, as long as they abide by the simple interface defined by the Subject-Observer design pattern.

At times, the entire View may need to be refreshed. When this happens, the View object only needs to instruct each BufferManager controlled by this View to, in turn, refresh itself.

The appearance of the areas managed by the BufferManagers in a View must be coordinated to constitute a consistent view of different aspects of the same data. We must make a decision about which object in our system should be responsible for maintaining such consistency. We could either let each BufferManager maintain references to all related BufferManagers or we could shift such responsibility to the owner of the BufferManagers – the concrete View object.

Considering that View objects already have a list of all BufferManagers for all areas in the View for painting purposes, we decided to use the latter option. In our design, we apply the Mediator design pattern, which is particularly suited to capturing such relationships between objects[19]. Following the Mediator pattern, we pay the expense of increased complexity of the concrete View objects, since they now must handle all required coordination between BufferManagers. However, it enables BufferManagers to concentrate on managing the underlying Buffer and to function independently of one another.

The user may interact to resize the View. The concrete View object makes its own decision on how to resize the BufferManagers according to the characteristic of this View.

We also expect the user to want to adjust the size and location of individual areas in the View controlled by BufferManagers. Upon receiving such actions (the user presses

and drags the mouse, for example), View can configure the BufferManagers and instruct them to refresh themselves.

Since we display a visualization of the expression data in an area, we expect that the user would want to interact with the graphics elements in the visualization. The interpretation of user intention and the decision on how to react to it should not be the View's responsibility, since the View does not have enough knowledge of the structure of the underlying expression data or how the data is visualized.

The BufferManager may decide to react to the user in some way (for example, by showing a selection box as the user drags the mouse). However, BufferManager still does not have enough information to fulfill the task on its own (BufferManager does not know the actual content being covered by the selection box, since it does not know which Buffer subclass is being used). It must consult the underlying Buffer to get the additional information.

Concrete Buffer objects are the ultimate drawing machines. They are the only objects that know how to render the expression data, either as rows of gene names, or columns of condition names, or as colored heat maps, or as horizontal or vertical tree structures, or something else.

However, it would be cumbersome to delegate all drawing operations to Buffer. For example, the drawing of a selection box in an area in View in response to the user's action of dragging a mouse does not need to be carried out by concrete Buffer objects and can be fulfilled by the BufferManager assigned to control that area.

Therefore, we decided that the main drawing should be handled in BufferManager and, whenever the BufferManager cannot draw all the requested information on its own, it delegates the drawing task to the underlying Buffer.

Display:

The user may want to configure a View and may also want to have different Views of the same expression data displayed on the same screen for easy comparison and reference. Therefore, on the highest level of visualization, we designed a Display class that encompasses a type of visualization, along with graphical widgets for user configuration of the View(for example, changing the color scheme).

Central:

Since there will be more than one type of visualization of the same data (heat map, tree, K-means clusters, etc.), the visualizations in one Display should ideally be able to react to changes user makes in other Displays. To facilitate the coordination between Displays, we designed a class Central to receive messages sent from individual Displays and relay these messages to other Displays that are interested in the messages.

We again use the design pattern Mediator[19] to capture such interaction between Central and Display.

ScrollBar:

As described in BufferManager, the ScrollBars implement the Observer interface and register themselves with BufferManagers. ScrollBars should respond to the changes in size and/or location of the associated BufferManager.

Listener:

The user might want to interact with the graphical elements presented in View to manipulate the image, and thus the underlying expression profile data. Therefore, we design View to implement a Listener class. The current implementation of View monitors mouse event but the ability to handle other events can be easily added to View by making View implement other listeners.

The user's actions could potentially change the contents of a BufferManager, for example, by signaling to try to include in the selection an area outside the currently viewable window and triggering an auto-scrolling event. When this happens, the position of the associated ScrollBars should be updated. This, again, will be gracefully handled by the Observer design pattern: when auto-scrolling is turned on, the BufferManager only needs to notify ScrollBars of the change by calling their ManagerChanged method. The ScrollBar gets the size and current display position of the BufferManager and adjust the ScrollBar appearance accordingly.

CHAPTER 6

EVALUATION METHODOLOGY

The software, ExpressionProfiler, has been developed to serve the needs of an interdisciplinary research project at the University of Georgia, the Fungal Genome Project. This project was initiated by faculty members from the Genetics Department and the Computer Science Department in the University of Georgia. Therefore, the development process of ExpressionProfiler has been closely linked to our user from the Genetics Department.

During our initial discussion with the user, Dr. Jonathan Arnold in the Genetics Department, we were asked to visualize the expression data in a scrollable graphical interface with zooming capability, so that he could easily view different parts of the data, while at the same time, be able to zoom out to see the overall characteristics of the data.

We first developed a prototype program that displayed the expression data in the conventional red and green color scheme for increased and decreased expression levels and asked the user about the kinds of interactions he would like to have. The user listed the following for the GridView:

1. Select row(s) and column(s)
2. Move selection
3. Delete selection
4. Reverse selection

The user also proposed some interactions for the proposed tree view:

1. Select a subtree
2. Reverse a subtree

Afterwards, we created a mature GridView with the requested interactions and then later the TreeView. Defining the meaning of the interactions on GridView is relatively easy. However, it becomes more difficult for TreeView. Questions in our discussions were 1) should there be a Delete operation for a subtree generated by UPGMA? 2) how should the visualization react to such user requests?

The UPGMA algorithm generates a binary tree after clustering and the tree representation is associated with a hierarchical relationship of proximity between clusters. Therefore, the deletion of even the smallest subtree would make the hierarchical relationship completely obsolete and the original tree structure would no longer convey the correct information. Both the user and the designer felt that it was difficult to restructure the tree representation after the proposed deletion without re-clustering and decided not to support subtree deletion at this stage.

Our user did suggest that we add two functions to our software: Find and Filter, which we have included, as described.

We have also added some other functionality, including the ability to undo operations and to print out the visualization area.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

We have designed a framework for visualizing gene expression data. Our design has enabled us to create two different views, which provide information to the user in a more accessible way than many existing software systems.

ExpressionProfiler has achieved stronger zooming capability and allows the user to explore the expression data on different levels of detail. It also supports interactions useful to the user, such as editing the data before clustering starts and manipulating the resulting tree after clustering finishes (UPGMA). ExpressionProfiler's visualization scheme maintains the inherent correlation between different aspects of the expression data upon user interaction. This ensures that the information conveyed to the user is correct, instead of misleading or confusing.

Our design framework is also extensible in the sense that new views for the data could be added with little or even no changes to the existing system, as long as the implementation for the new views comply with the interface stipulated by the abstract class Buffer.

Also, the idea of the BufferManager class enables new views to be easily created by assembling existing Buffer implementations and placing them under the control of BufferManagers in different areas of the new view.

Such characteristics are important to a software system to be used in gene expression visualization and analysis, because as new methods for analyzing the data are

adopted by the research community, corresponding visualizations will be needed to explore the result of such methods.

Currently, ExpressionProfiler has only implemented the GridView for the unclustered data and the TreeView for the result of the UPGMA clustering algorithm. However, its achievements in terms of visualization effects and user interaction associated with these two views are encouraging. Future work would certainly include implementing more clustering algorithms such as the K-means clustering and the Self-Organizing-Map, as well as providing visualization for the processes and the results of these algorithms.

REFERENCES

1. M. Schena, D. Shalon, RW. Davis, P.O. Brown, *Quantitative Monitoring of Gene Expression Patterns with a Complementary DNA Microarray.* Science, 1995. **270**: p. 467.

2. D.E. Bassette Jr., M. Eisen, M. Boguski. *Gene expression informatics--it's all in your mine.* Nature Genetics, 1999. **21**: p. 51-55.

3. T.R. Hughes, M. Marton, A.R. Jones, C.J. Roberts, R. Stoughton, C.D. Armour, H.A. Bennett, E. Coffey, H. Dai, Y.D. He, M.J. Kidd, A.M. King, M.R. Meyer, D. Slade, P.Y. Lum, S.B. Stepaniants, D.D. Shoemaker, D. Gachotte, K. Chakraburtty, J. Simon, M. Bard, S.H. Friend, *Functional discovery via a compendium of expression profiles.* Cell, 2000. **102**: p. 109-126.

4. S. Chu., J. DeRisi., M. Eisen, J. Mulholland., D. Botstein., P.O. Brown, I. Herskowitz., *The transcriptional program of sporulation in budding yeast.* Science, 1998. **282**: p. 699.

5. R.W. Hamming, *Error detecting and error correcting codes.* Bell Systematic Technology Journal, 1950. **29**: p. 147.

6. P.H.A. Sneath, R.P. Sokal, *Numerical Taxonomy.* 1973, San Francisco: W. H. Freeman and Company.

7. A.K. Jain, R.C. Dubes, *Clustering Methods and Algorithms*, in *Algorithms for Clustering Data.* 1988, Prentice Hall. p. 89.

8. T. Kohonen *The Self-Organizing Map*. in *Proceedings of the IEEE*. 1990.

9. M. Eisen, P. Spellman, P. O. Brown, D. Botstein, *Cluster analysis and display of genome-wide expression patterns.* Proc. Natl. Acad. Sci. USA, 1998. **95**: p. 14863-14868.

10. K. White, S. Rifkin., P. Hurban, D. Hogness, *Microarray Analysis of Drosophila Development During Metamorphosis.* Science, 1999. **286**: p. 2179.

11. A.A. Alizadeh, M. Eisen, R.E. Davis, C. Ma, I.S. Lossos, A. Rosenwald, J.C. Boldrick, H. Sabet, T. Tran, X. Yu, J.I. Powell, L. Yang, G.E. Marti, T. Moore, J. Hudson Jr, L. Lu, D.B. Lewis, R. Tibshirani, G. Sherlock, W.C. Chan, T.C. Greiner, D.D. Weisenburger, J.O. Armitage, R. Warnke, L.M. Staudt, *Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling.* Nature, 2000. **403**: p. 503-511.

12. *http://evolution.genetics.washington.edu/phylip/newicktree.html*.

13. G. Sherlock, *XCluster*. 2000.

14. M. Eisen, *http://rana.lbl.gov/*.

15. M. Brush, *Making Sense of Microchip Array Data.* TheScientist, 2001. **15**(9): p. 25.

16. GeneSpring, *http://www.sigenetics.com/*, Silicon Genetics.

17. DecisionSite, *http://www.spotfire.com*, Spotfire.

18. E. Gamma, R. Helm, R. Johnson, J.Vlissides, *Observer*, in *Design Patterns*. 1995, Addison-Wesley. p. 293.

19. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Mediator*, in *Design Patterns*. 1995, Addison-Wesley. p. 273.

20. H. Bancroft, *Introduction to Biostatistics*. 1957, Paul B. Hoeber. p. 150.

21. GeneExplore, *http://www.applied-maths.com/ge/ge.htm,* Applied Maths.