University of Crete
School of Sciences
Department of Computer Science

Using Treemaps to visualize Gene Microarray and Phylogenetic Data

Adam Arvelakis

Supervising Professor: Ioannis G. Tollis

Master of Science Thesis

Heraklion March 2005

Using Treemaps to visualize Gene Microarray and Phylogenetic Data

Abstract

With the invention of the microarray (DNA chip), genomic research has increased massively and a burst of discoveries has begun. But as data get ever more specific, researchers need more precise tools to come to useful conclusions. Phylogenetic research is an area of biology which is characterised by the large number of hierarchical data which are produced during the process of determining the ancestral relationship between species.

Treemaps is a novel visualization method which uses space efficiently to display trees by replacing the classic node representation with rectangles which take up 100 % of the space made available to them.

Treemaps can be used in order to visualize the data produced by microarrays and phylogenetic research. The aim of applying treemaps to these branches of biology is to create a useful visualization which translates the huge amount of numbers into something more understandable.

What we propose in this thesis is to use the treemap visualization in order to display microarray and phylogenetic data. We show that we can visualize clustered microarrays using animating treemaps. In fact, since two trees can be constructed by clustering the microarray data, two animating treemaps can be produced by one microarray. For phylogenetic trees which are a static tree structure this is not possible, but the fact that the root of the tree is uncertain leaves a margin for implementing features concerning the designation of the root.

A program has been developed which implements the above proposal as well as numerous features which benefit the user during this visualization. This includes features which alter the way treemaps are displayed without altering the structure. It also includes features which alter the structure of the treemap as well as features for specific non-treemap purposes such as microarray clustering.

Οπτικοποίηση microarrays και φυλογενετικών δεδομένων με τη χρήση treemaps

Περίληψη

Με την εφεύρεση του microarray (ή αλλιώς DNA chip), η έρευνα στην γενετική έχει γνωρίσει τεράστια ανάπτυξη και μια πληθώρα ανακαλύψεων έχει αρχίσει να πραγματοποιείται. Όμως καθώς τα δεδομένα γίνονται ολοένα και πιο εξειδικευμένα, οι ερευνητές χρειάζονται πι ακριβή εργαλεία για να φτάσουν σε χρήσιμα συμπεράσματα. Η φυλογενετική έρευνα είναι ένας κλάδος της βιολογίας που χαρακτηρίζεται από τον μεγάλο όγκο ιεραρχικών δεδομένων που παράγονται κατά τη διάρκεια του καθορισμού των προγονικών σχέσεων μεταξύ ειδών.

Τα treemaps είναι μια μέθοδος οπτικοποίησης που χρησιμοποιεί τον χώρο αποτελεσματικά ώστε να απεικονίσει δέντρα αντικαθιστώντας την κλασική αναπαράσταση κόμβων με ορθογώνια που καταλαμβάνουν το 100 % του χώρου που τους διατίθεται.

Τα treemaps μπορούν να χρησιμοποιηθούν ώστε να οπτικοποιήσουν τα δεδομένα που παράγονται από microarrays και φυλογενετική έρευνα. Ο στόχος της εφαρμογής των treemaps σε αυτούς τους κλάδους της βιολογίας είναι να δημιουργηθεί μια χρήσιμη οπτικοποίηση που να μπορεί να μετατρέψει τους πολύ μεγάλους όγκους δεδομένων σε κάτι πιο κατανοητό.

Η πρόταση μας σε αυτή τη διατριβή είναι να χρησιμοποιηθούν τα treemaps για την οπτικοποίηση των δεδομένων που δημιουργούνται από microarrays και φυλογενετική έρευνα. Θα δείξουμε ότι μπορούμε να οπτικοποιήσουμε clustered microarrays χρησιμοποιώντας animating treemaps. Μάλιστα, αφού δημιουργούνται δυο δέντρα κατά το clustering των δεδομένων του microarray, για κάθε ένα microarray μπορούμε να εμφανίσουμε δυο animating treemaps. Για φυλογενετικά δέντρα τα οποία είναι μια στατική δενδρική δομή αυτό δεν είναι δυνατό, αλλά το γεγονός ότι η ρίζα του δέντρου δεν είναι αυστηρά καθορισμένη αφήνει περιθώρια για την υλοποίηση λειτουργιών που επιτρέπουν στον χρήστη να καθορίσει εκείνος την ρίζα του δέντρου.

Έχει αναπτυχθεί λογισμικό που υλοποιεί την παραπάνω πρόταση καθώς και αρκετές λειτουργίες που ωφελούν τον χρήστη κατά τη διάρκεια της οπτικοποίησης. Αυτό περιλαμβάνουν λειτουργίες που επηρεάζουν το πως εμφανίζονται τα treemaps χωρίς να επηρεάζουν την δομή του treemap. Επίσης περιλαμβάνει λειτουργίες που αλλάζουν την δομή του treemap καθώς και λειτουργίες για συγκεκριμένους σκοπούς άσχετους με treemaps όπως το clustering ενός microarray.

1. Introduction

1.1. Motivation

With the advancement of the mapping of the human genetic code since the late 90s and the invention of the microarray (DNA chip), genomic research has increased massively and a burst of discoveries has begun. But as the data gets ever more specific, researchers need more precise tools to come to useful conclusions.

Phylogenetics is an area of biology which is characterised by the large number of hierarchical data which is produced during the inference process of ancestral relationships between species. The main use of computers in this area is for executing the computationally intensive algorithms for determining common ancestors. Besides that, experts benefit from visualizations which display this hierarchy efficiently.

Treemaps is a novel visualization which uses space efficiently to display trees by replacing the classic node representation with rectangles which take up 100 % of the space available.

Treemaps can be used in order to visualize the data produced by microarrays and phylogenetic inference. The aim of applying treemaps to these areas of biology is to create a useful visualization which translates the huge amount of information into something more understandable.

1.2. Outline of the thesis

In Section 2 we provide background theory and related work concerning microarrays (including gene expressions and clusterings), phylogenetic trees and finally treemaps. Section 3 contains our proposal about using treemaps and animating treemaps in order to display the most prominent tree structures found in microarray and phylogenetic inference. In Section 4 we present the details concerning the most useful features of the developed implementation/software. Section 5 shows results from using this approach regarding microarrays. In Section 6 we list the more common features found in the program. Section 7 contains the conclusions of this work and any

improvements/variations. Section 8 contains references while the Appendix is a manual for the developed software.

2. Background and Related Work

2.1. Microarrays

A DNA microarray (also DNA chip or gene chip in common speech) is a piece of glass or plastic on which pieces of DNA have been affixed in a microscopic array. Scientists use such chips to screen a biological sample for the presence of many genetic sequences at once. The affixed DNA segments are known as probes. Thousands of identical probe molecules are affixed at each point in the array to make the chips effective detectors.

A typical microarray sampling procedure is displayed below [1]:

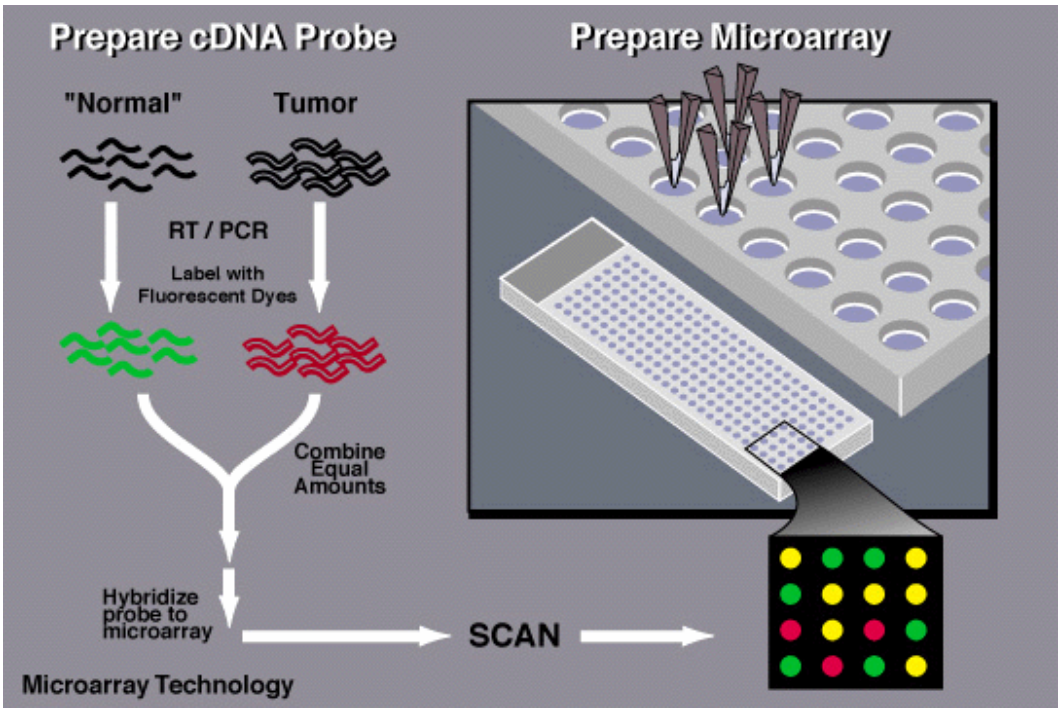


Figure 2.1.1: The process of using a microarray

Typically microarrays are used to detect the presence of mRNAs that may have been transcribed from different genes and which encode different proteins. The RNA is extracted from many cells, ideally from a single cell type, and then converted to cDNA or cRNA. Fluorescent tags are enzymatically incorporated into the newly synthesized strands or can be chemically attached to the new strands of DNA or RNA. A cDNA or

cRNA molecule that contains a sequence complementary to one of the single-stranded probe sequences will hybridize, or stick, to the spot at which the complementary probes are affixed. The spot will then fluoresce (or glow) when examined using a microarray scanner. The quantitative expression of this fluorescence is called gene expression (this definition of gene expression concerns microarrays only, generally gene expression means the process of a gene converting its information into the structures and functions of a cell) [2][3].

Because many proteins have unknown functions, and because many genes are active all the time in all kinds of cells, researchers usually use microarrays to make comparisons between similar cell types. For example, an RNA sample from brain tumor cells might be compared to a sample from healthy neurons. Probes that bind RNA in the tumor sample but not in the healthy one may indicate genes that are uniquely associated with the disease. Typically in such a test, the two samples' cDNAs are tagged with two distinct colors, enabling comparison on a single chip. Researchers hope to find molecules that can be targeted for treatment with drugs among the various proteins encoded by disease-associated genes [2].

Since there are hundreds or thousands of distinct probes on an array, each microarray experiment can accomplish the equivalent of thousands of genetic tests in parallel. Microarrays have therefore dramatically accelerated many types of investigations.

Microarrays are also being used to identify genetic mutations and variation in individuals and across populations. Short oligonucleotide arrays can be used to identify the single nucleotide polymorphisms (SNPs) that are thought to be responsible for genetic variation and the source of susceptibility to genetically caused diseases. Generally termed "genotyping" applications, chips may be used in this fashion for forensic applications, rapidly discovering or measuring genetic predisposition to disease, or identifying DNA-based drug candidates [2].

Clustering algorithms are used on the gene expressions in order to identify co-expression of genes by relying solely on the gene expressions themselves. There is also the case of external categorization by the provider of the microarray data (the researchers/doctors who sampled the patients), for example whether the patients whose

brain cells where sampled have cancer or not. This is called *classification* and is almost always used with 2 labels (whether a patient is suffering from a disease or not, in other words, a boolean variable). Current research efforts aim to develop efficient clustering algorithms by analyzing the data provided by clusterings and classifications of the current microarray [4] [5] [6].

Visualization attempts so far include the following:

The classic red-green visualization as extracted from raw microarrays:

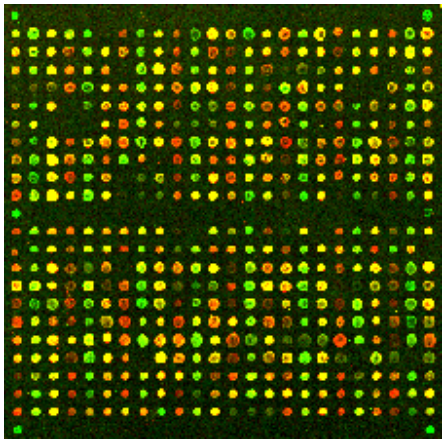


Figure 2.1.2: A raw microarray image

A microarray with labels:

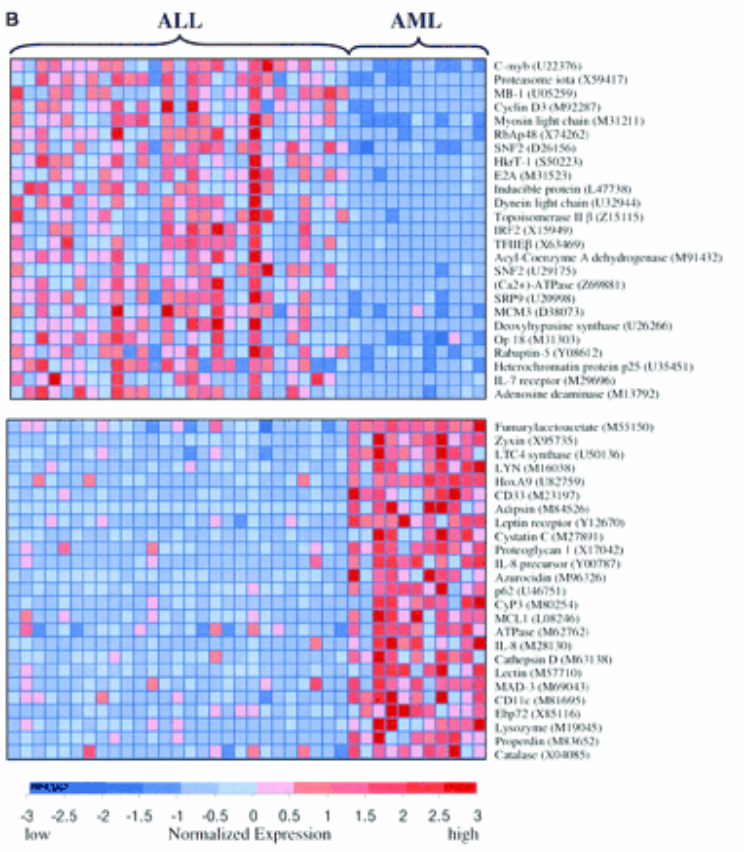


Figure 2.1.3: A computer-generated microarray image with labels

A 2D comparative analysis of 2 genes [7]:

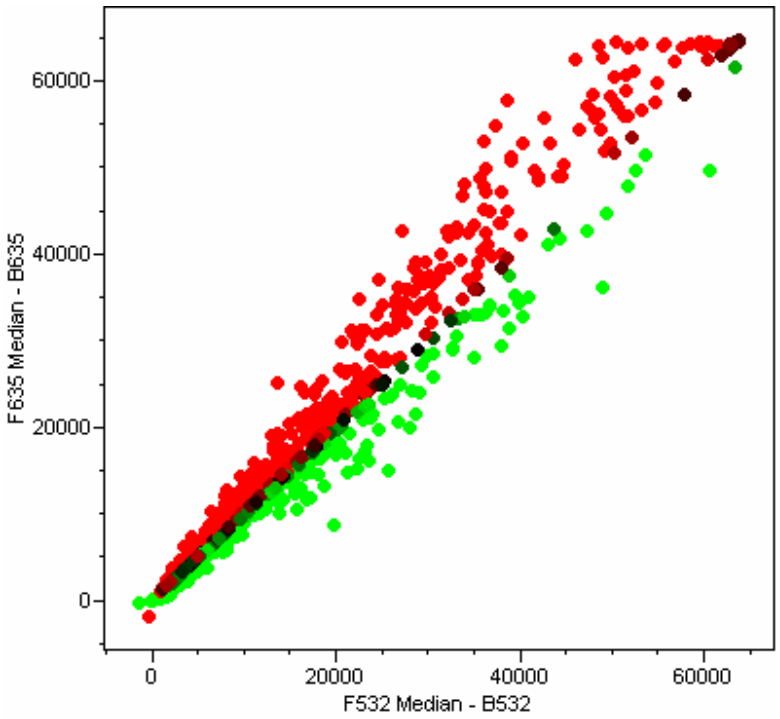


Figure 2.1.4: A comparison of 2 genes

A clustered microarray:

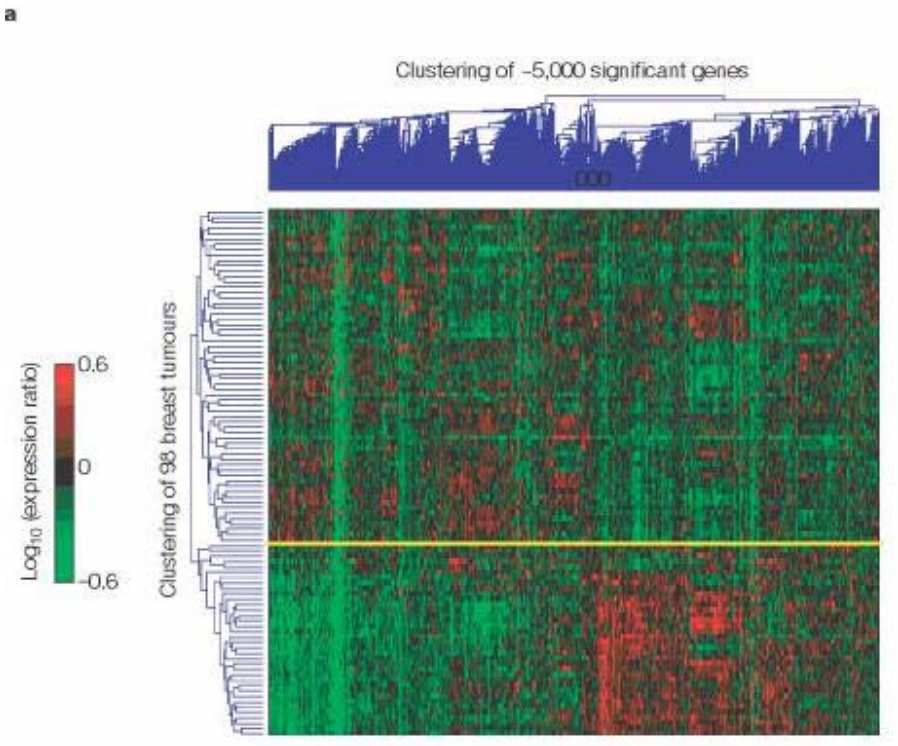


Figure 2.1.5: A microarray image with double clustering trees

A 2D comparative analysis of 2 clusters [7]:

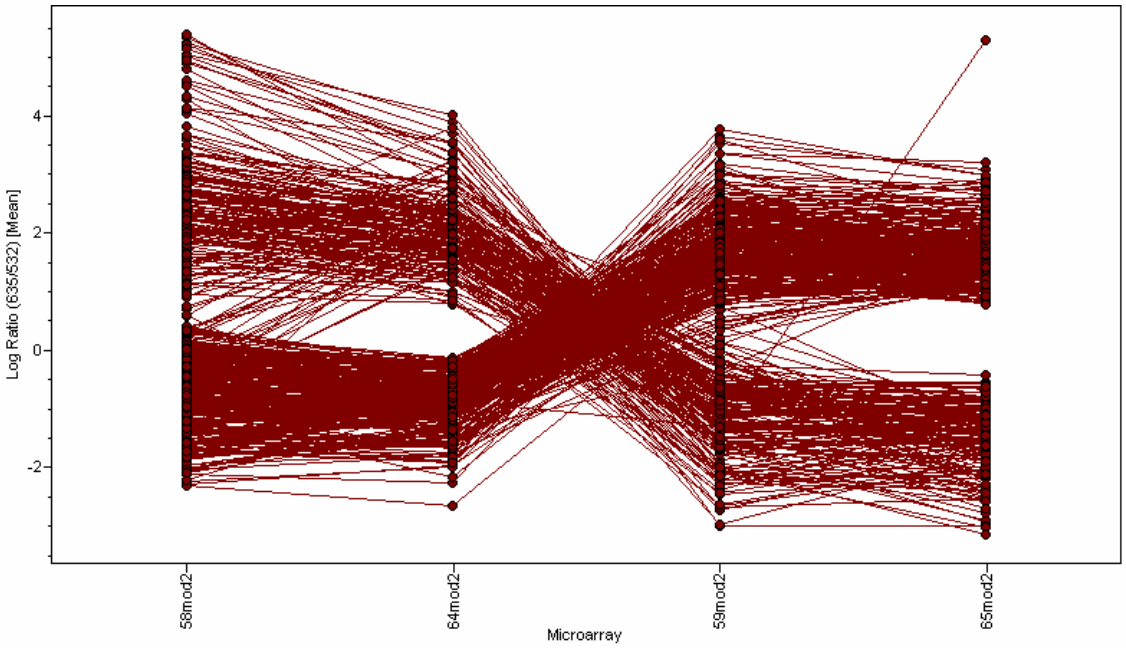


Figure 2.1.6: A comparison of 2 clusters

Using treemaps to provide organized data from the Gene Ontology database [8]:

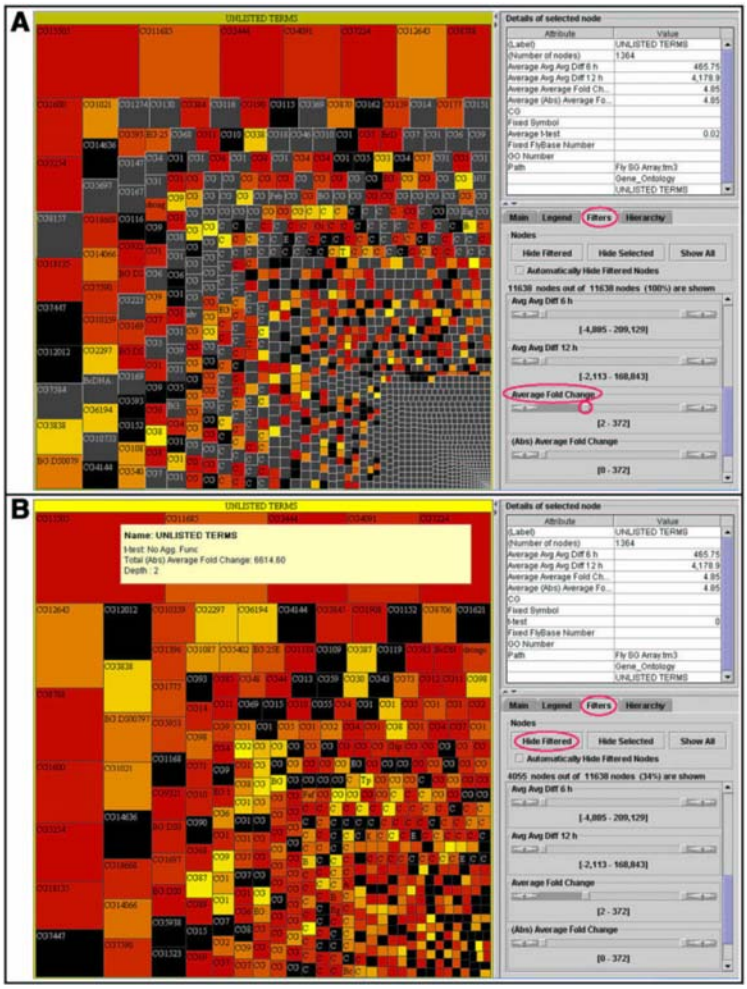


Figure 2.1.7: An approach using treemaps

2.2. Phylogenetic Trees

A phylogenetic tree is a tree which describes the evolutionary relationship among a set of species. In a phylogenetic tree, each node represents the most recent common ancestor of its children, and edge lengths correspond to time estimates (under certain models of evolution). Each node in a phylogenetic tree is called a taxonomic unit. Internal nodes are generally referred to as Hypothetical Taxonomic Units (HTUs) as they cannot be directly observed [9].

A *rooted* phylogenetic tree is a directed tree where a specific node corresponds to the most recent common ancestor of all the leaves of the tree. A rooted phylogenetic tree of all species without many details is shown below:

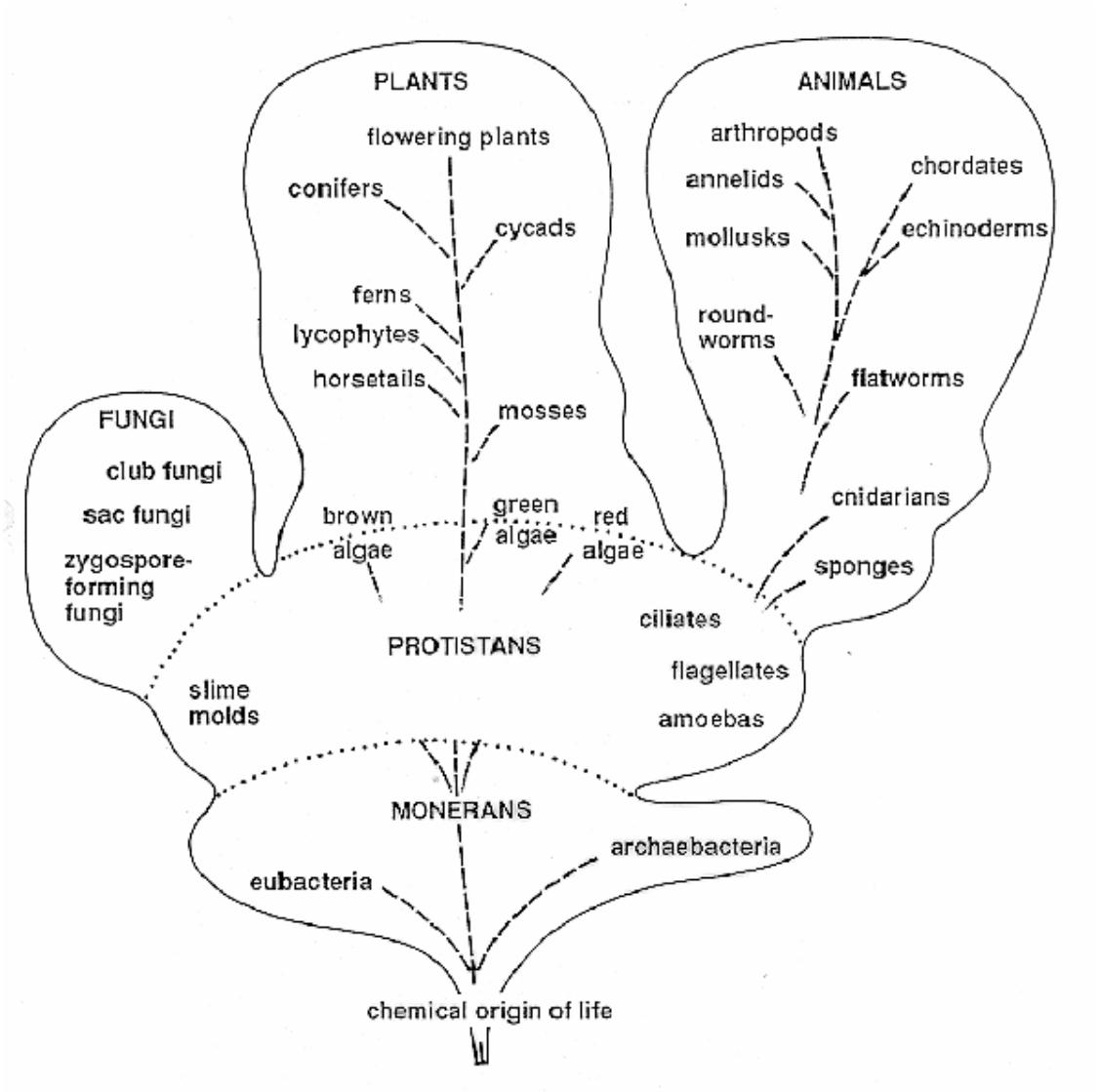
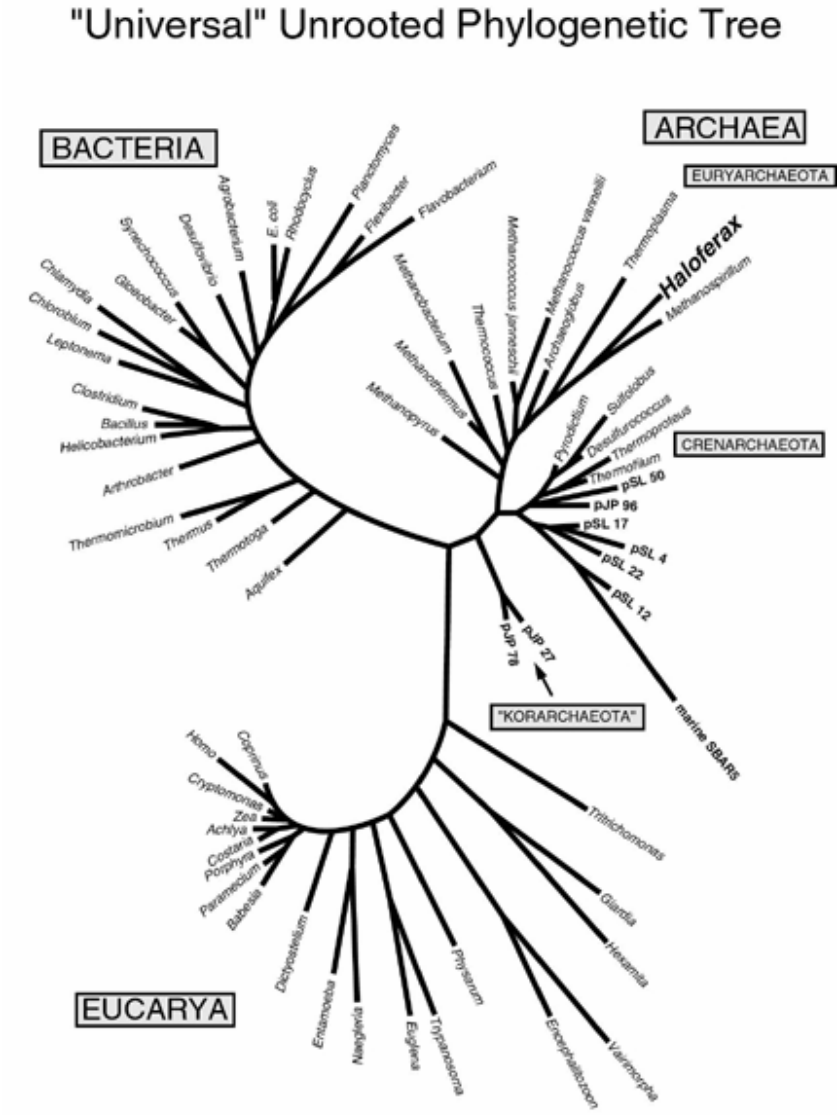


Figure 2.2.1: A rooted phylogenetic tree of all species, very few actual species are shown. Names in capitals represent orders of species

An *unrooted* phylogenetic tree is, loosely speaking, a tree derived from a rooted phylogenetic tree by omitting the root. More precisely, it is a forest of rooted phylogenetic trees depicted so that the individual roots are all linked to a species representing the origin of life on earth. An unrooted phylogenetic tree of all species without many details is shown below:



Barnes, S.M. *et al.*, 1996, Proc. Natl. Acad. Sci. USA, **93**: 9188-9193.

Figure 2.2.2: An unrooted phylogenetic tree of all species, very few actual species are shown. The framed names represent orders of species

The most commonly used methods to infer phylogenies include *cladistics* [10], *phenetics* [11], *maximum likelihood* [12], *bayesian inference* [13], *neighbor joining* (in fact an application of the nearest neighbor algorithm [14]) and *maximum parsimony* [15], the latter four are computational methods while the first two are not. The computational methods can be further categorized into character based methods (maximum likelihood, bayesian inference), parsimony based methods (maximum parsimony) and distance based methods (neighbor joining).

In addition to the rooted and unrooted radial trees shown above previous visualization attempts include the following:

Cladograms, originally used in cladistics:

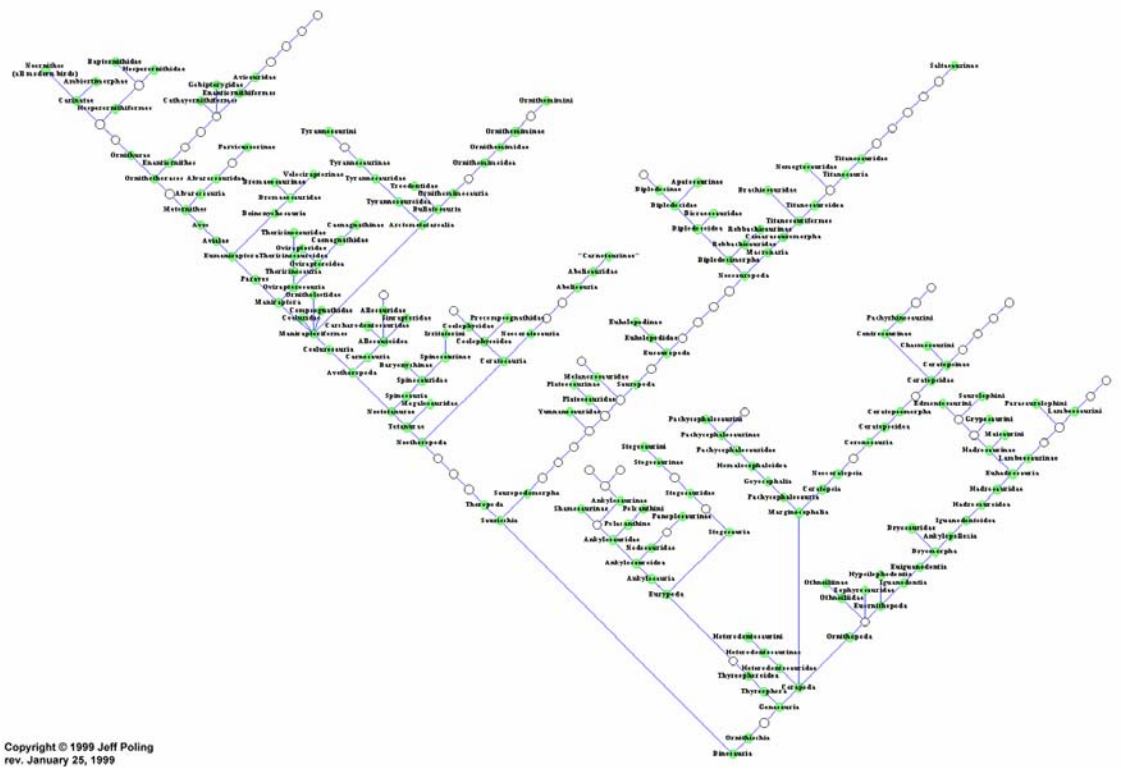


Figure 2.2.3: A phylogenetic cladogram (http://www.dinosauria.com/pics/clados/clado.gif)

Phenograms, originally used in phenetics:

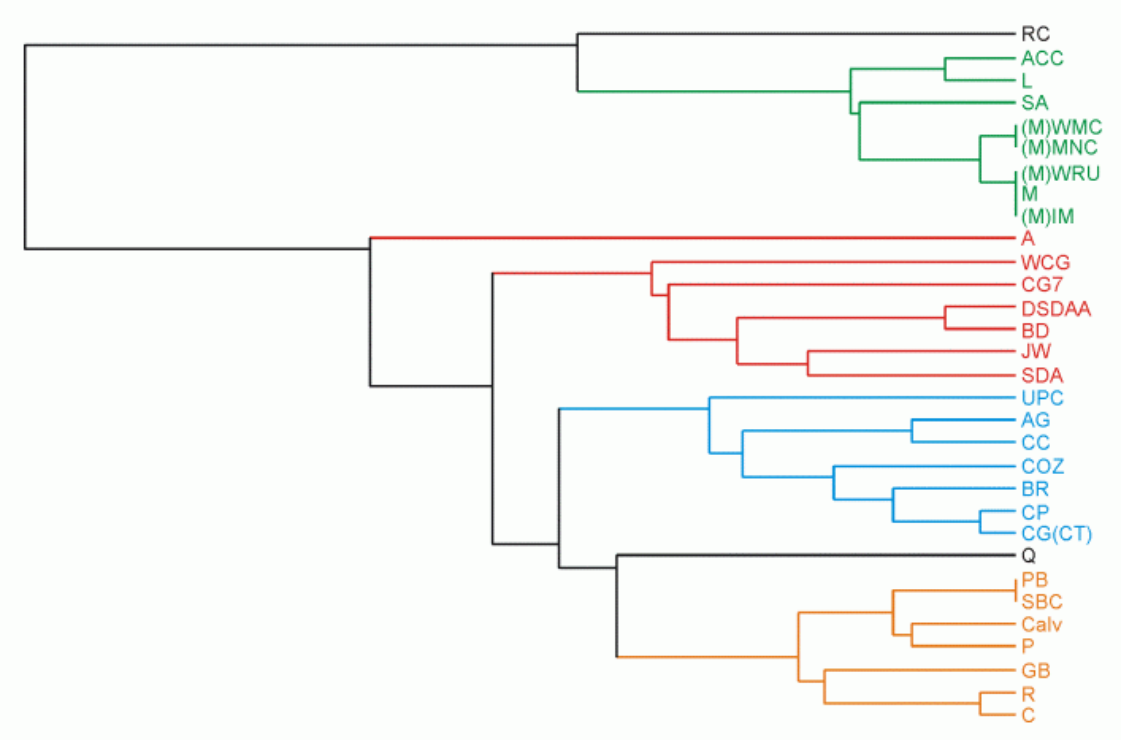


Figure 2.2.4: A phylogenetic phenogram (http://jom-emit.cfpm.org/2001/vol5/lord_a&price_i.html)

2.3. Treemaps

Treemaps is a visualization proposed by Ben Shneiderman [16] in the early 1990s for displaying tree structures. The common tree visualization displays a tree as in the following figure:

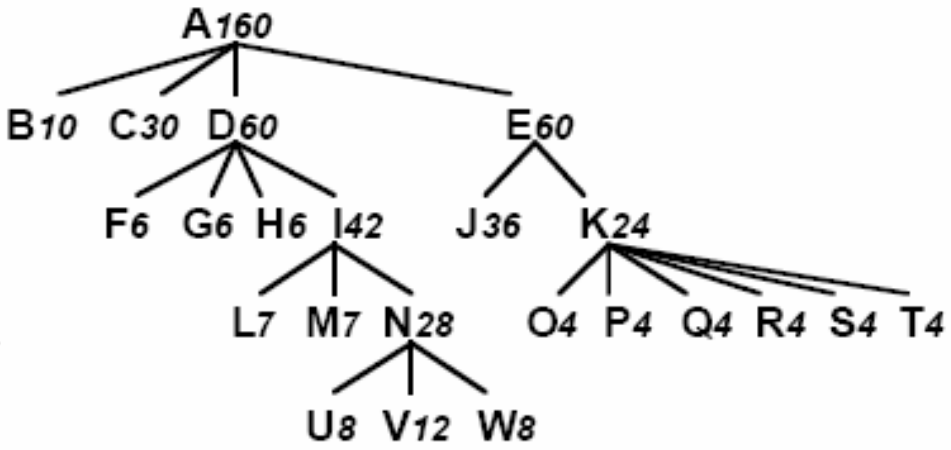


Figure 2.3.1: The classic tree visualization, the numbers on each node represent the nodes' weight

As trees grow in size this display becomes decreasingly useful due to wasted space. So in order to save space the next step was a more orthogonal approach, below we show the same tree using a Venn diagram:

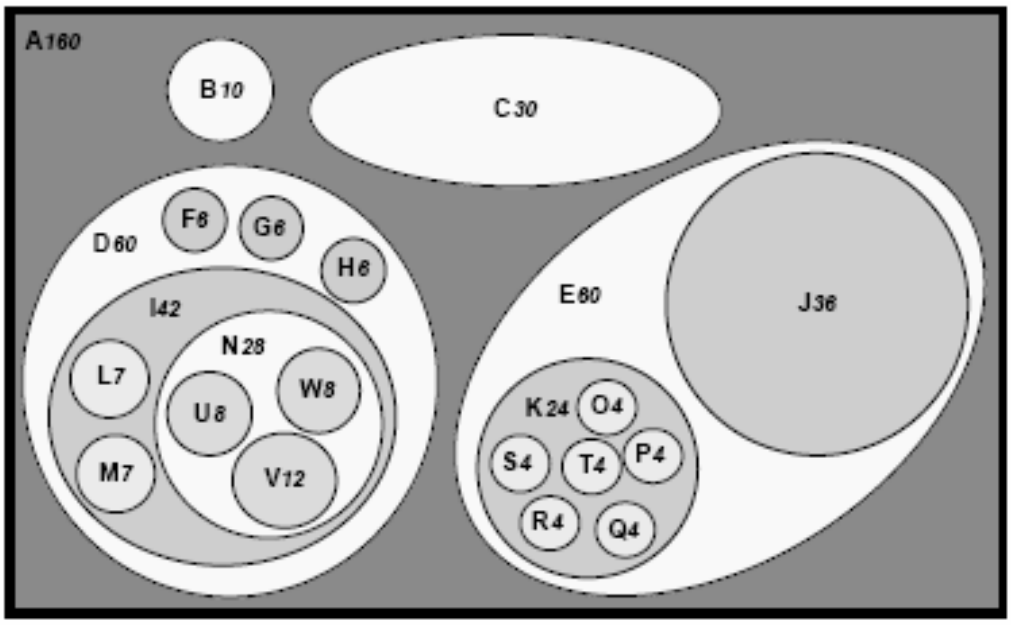


Figure 2.3.2: The same tree as in figure 2.3.1 shown as a Venn diagram

Of course Venn diagrams can do much more than just display trees, but this was the next step in Shneiderman's syllogism towards treemaps. After that followed nested treemaps, and finally treemaps themselves:

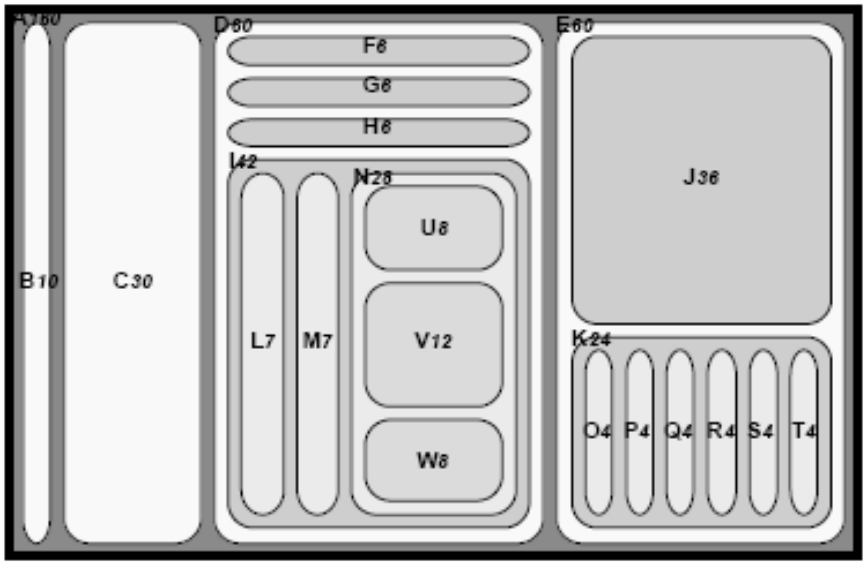


Figure 2.3.3: The same tree as in figure 2.3.1 shown as a nested treemap

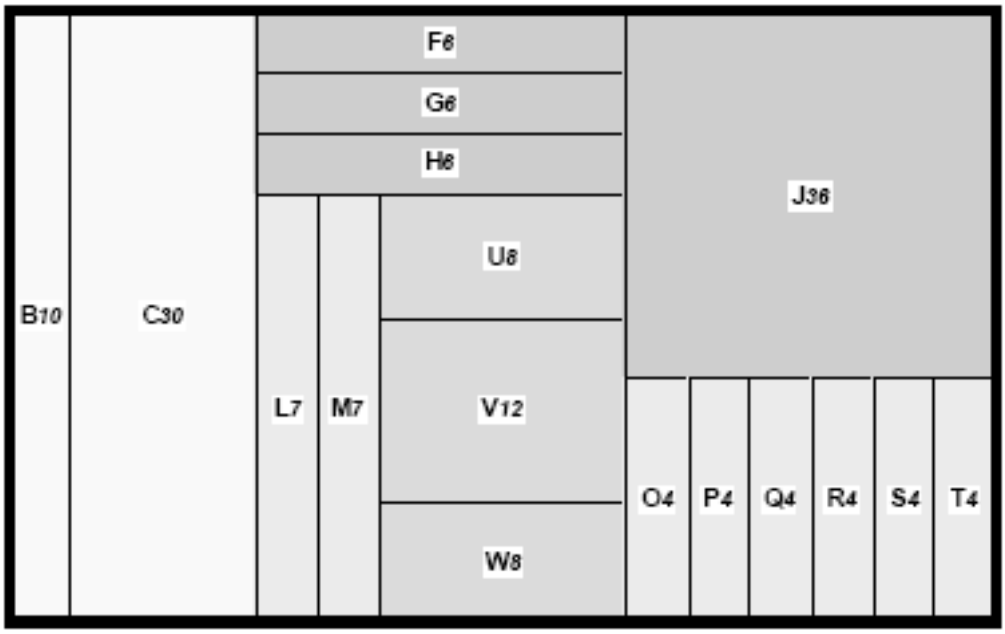


Figure 2.3.4: The same tree as in figure 2.3.1 shown as a treemap

The algorithm for drawing treemaps is a simple variation on the standard tree-drawing algorithm. The standard treemap algorithm simply takes a given area and positions a number of siblings there from left to right (or top to bottom depending on the orientation), just like the standard tree algorithm. The main difference is that treemaps use all of the available space while trees draw a symbol for the node at the centre of the

given space and connect nodes with edges. The rectangle assigned to each node is produced by the *findBounds* method shown below which simply divides the given space into equal parts. Besides simply dividing the space into as many equal-sized areas as there are children, the relative size of each area compared to its siblings' areas can depend on a *size* property of each node which is optional. The color intensity of each rectangle can also depend on a *weight* property. This method is the standard treemap-generating algorithm, named *slice & dice* because the produced rectangles are quite thin [16]. Below we can see this algorithm in pseudo code, the `orientation` variable is a simple string used to switch from horizontal to vertical orientation:

```
Algorithm TreemapAlgo(Node root, Rectangle bounds, String orientation)
{
fillRectangle(bounds);
//switch orientation
String newOrientation = (orientation == "horizontal" ? "vertical" :
"horizontal");
if (root is internal node)
       for (each child node of root)
              {
              Rectangle newBounds = findBounds(root, child-node, bounds,
              orientation);
              TreemapAlgo(child-node, newBounds, newOrientation);
              }
}
```

A variation to the slice & dice algorithm is the *squarified* algorithm [17] which seeks to produce rectangles as square as possible. Since fitting *n* squares of a given area in a rectangle of given dimensions is a difficult problem, the optimal solution takes exponential time to compute provided that the problem *can* be solved, i.e. *n* squares can fit into the given rectangle. Instead, the restrictions are relaxed and no perfect squares are required, so the problem becomes that of optimization of a heuristic (in our case, the worst aspect ratio of all rectangles put into the frame).

The algorithm for producing squarified treemaps is a bit more complex than the slice & dice. Each set of siblings to be arranged in a rectangle is first sorted based on area size because getting rid of the biggest nodes first leads to better results. Then a row is

created and nodes are added until the addition of a node leads to a worst aspect ratio of the row which is worse than without the node, so the row is frozen in place and a new one is created beside it. The algorithm is shown below:

```
Algorithm SquarifiedAlgo(Node root, Rectangle bounds)
{
List of Nodes children = root.getChildren();
List of Nodes row = new List of Nodes();
sortInOrderOfDescendingLeafCount(children);
squarifyLevel(children, row,  width(bounds));
for (each child node of root)
       SquarifedAlgo(child-node, child-node.bounds);
}

Algorithm squarifyLevel(List of Nodes children, List of Nodes row, Real w)
{
Node c = head(children);
if (worst(row, w) ≤ worst(row++[c], w)) then
       squarify(tail(children), row++[c], w)
else
       {
       layoutrow(row);
       squarify(children, [], width());
       }
}
```

The number `w` is used to compute the aspect ratio of each node by the method `worst`.

The slice & dice and squarified algorithms produce treemaps which are quite different concerning aspect ratios and preservation of order among the siblings. For example the two figures below show the same tree displayed first as a slice & dice treemap (above) and as a squarified treemap (below):

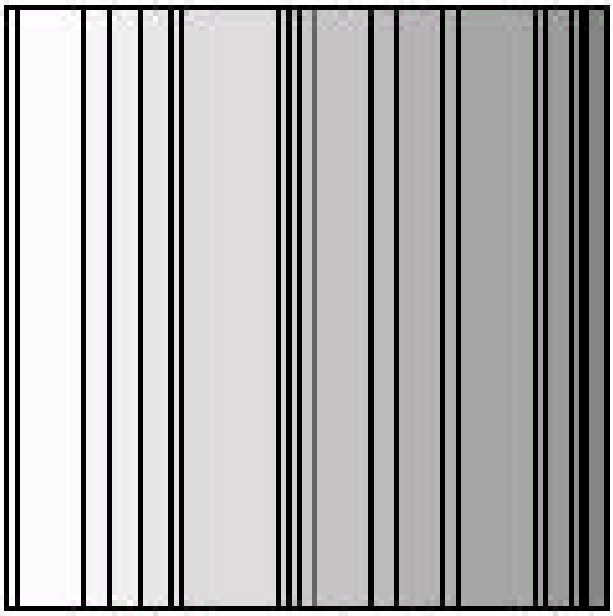


Figure 2.3.5: A tree consisting of a root and 20 children displayed as a slice & dice treemap, producing perfect preservation of order but very high aspect ratios

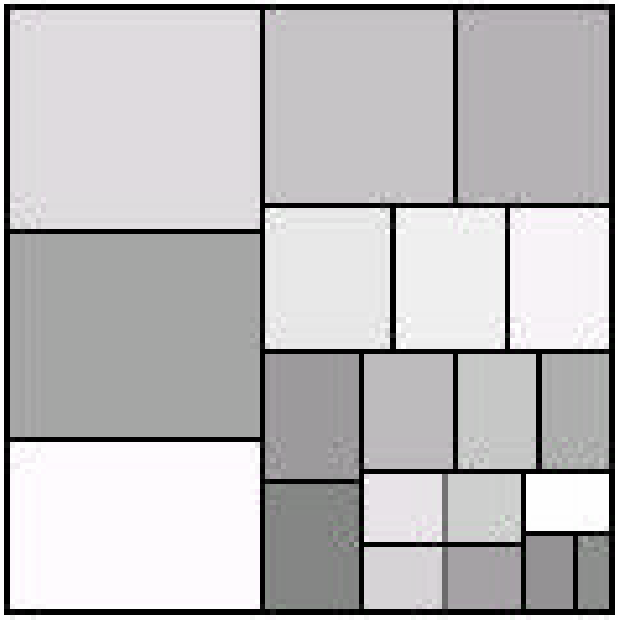


Figure 2.3.6: A tree consisting of a root and 20 children displayed as a squarified treemap, producing aspect ratios very close to 1 but complete loss of order

The above figures show a clear difference in aspect ratios as well as in preservation of the order of the children. The slice & dice and squarified algorithms lie at

the opposite ends of the aspect-ratio-versus-preservation-of-order trade-off. Other variations which lie closer to the middle are the *strip* and *ordered* algorithms [18].

Besides different algorithms which produce treemaps with different properties there are several features which can be added to treemaps in general to enhance the experience they provide.

Adding *frames* (or *borders*) [17] is the process of drawing a border around each internal node so that the internal structure is further pointed out, a treemap without borders runs the added risk of concealing color information of an internal node:

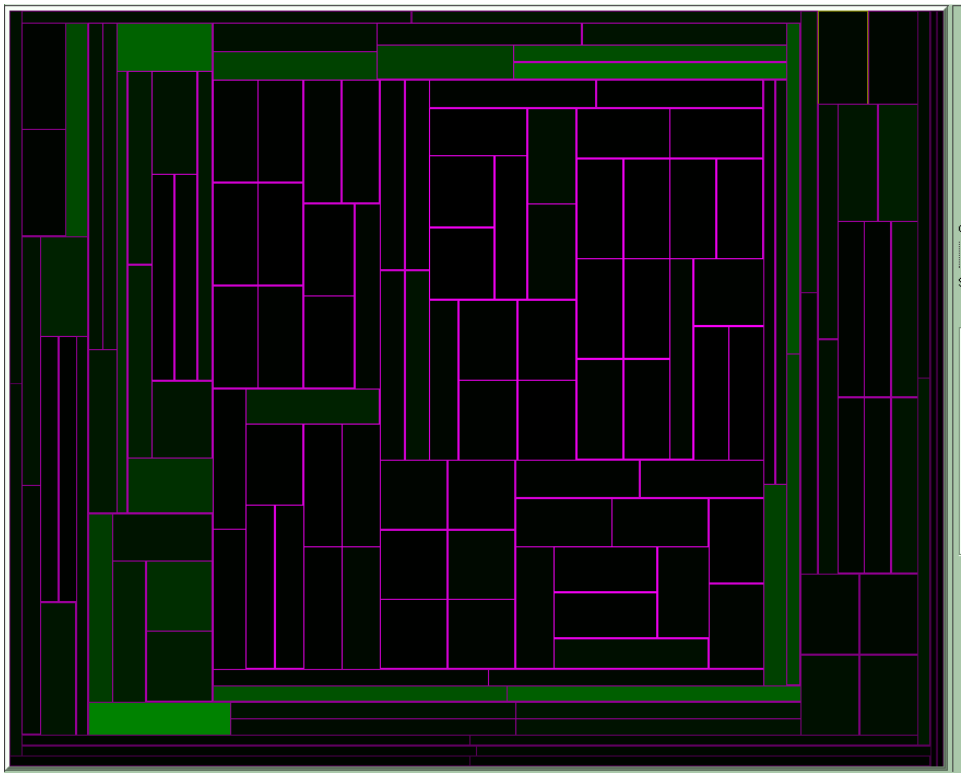


Figure 2.3.7: A treemap without borders

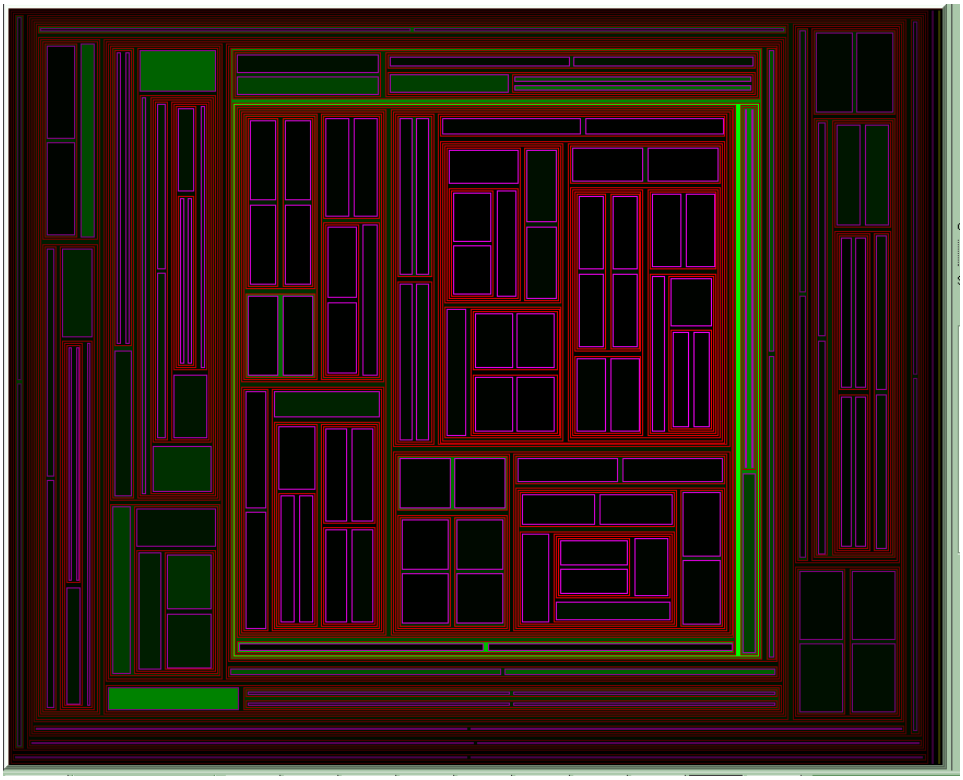


Figure 2.3.8: A treemap with a 2-pixel border around each node, internal weights are revealed

Adding *cushions* [17] is the process where instead of flat areas a pseudo-3D effect is produced where for each level a bump is added to the nodes at this level:

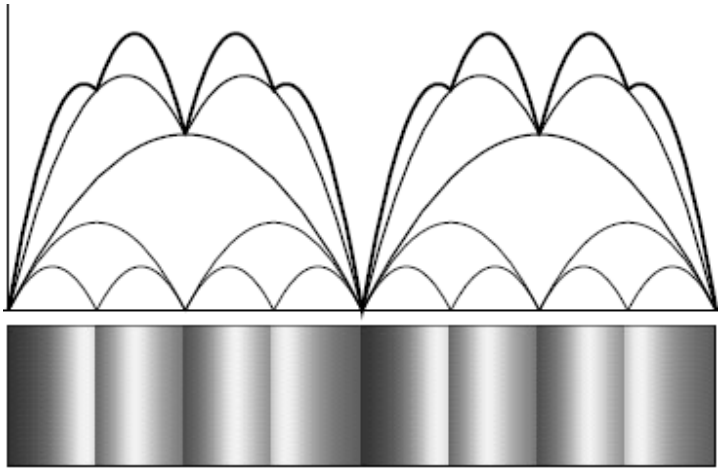


Figure 2.3.9: The addition of bumps to produce the cushion effect in one dimension

So, in another way, again the structure is pointed out:

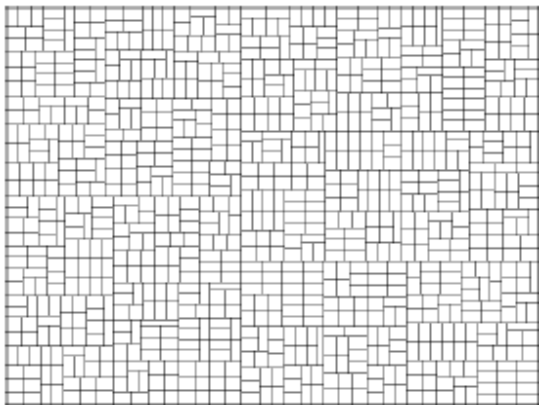


Figure 2.3.10: A treemap

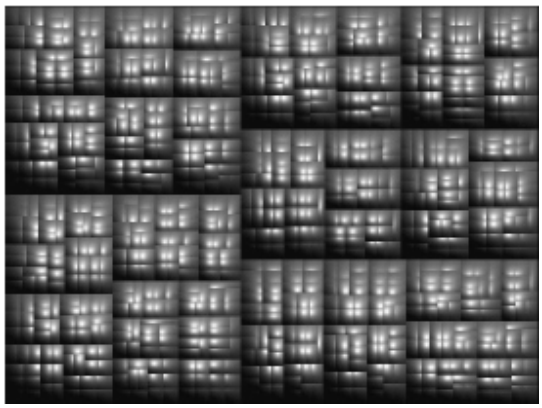


Figure 2.3.11: The treemap from figure 2.3.10 with cushions

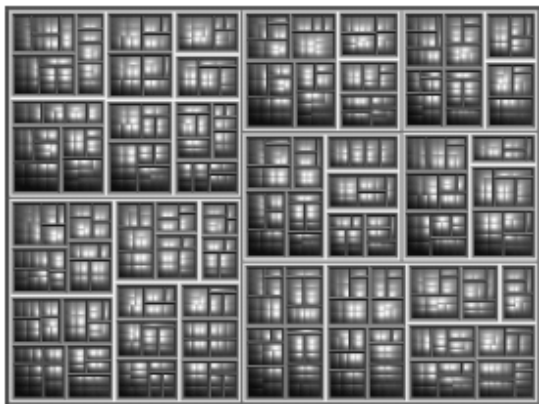


Figure 2.3.12: The treemap from figure 2.3.10 with cushions and borders

The idea of *animating treemaps* [19] is used to display dynamically changing information. Instead of just one value corresponding to each node's size, each node of the treemap holds a set of *n* values (one for each frame of the animation). The main problem is that algorithms such as the squarified do not preserve the order of siblings, so during

the animation the computation of every frame using squarified treemaps leads to jumps of a node within the animation making it very difficult to make any sense of the structure of the data. So in order to make effective use of animating treemaps the preservation of the order of siblings is of paramount importance.

3. Proposal

We propose to use the treemap visualization in order to display microarray and phylogenetic data. The fact that microarrays provide data organized in a matrix of genes versus patients translates in *number of genes* gene expressions for every patient and *number of patients* gene expressions for every gene as can be seen in the figure below:

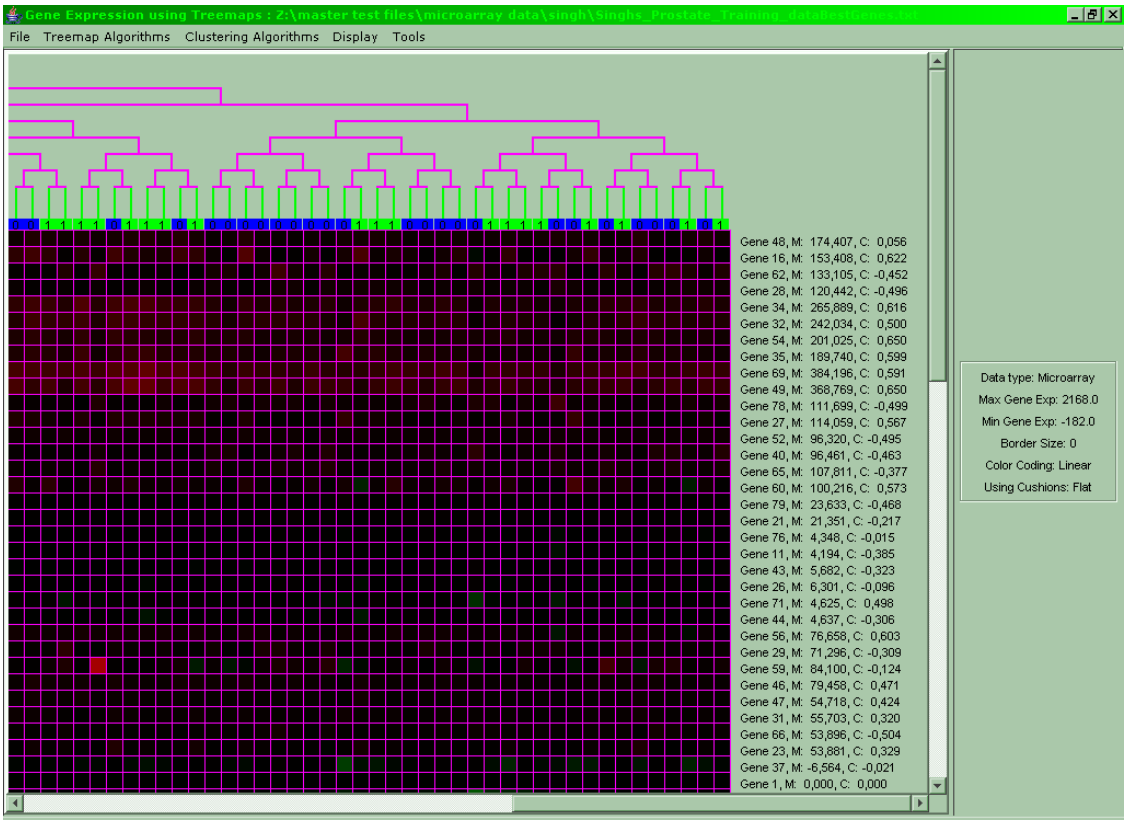


Figure 3.1: A typical clustered microarray. Every row represents one gene and every column one patient

So for microarray data the usage of animating treemaps becomes possible since, for example, for the top clustering tree we have several different trees (one for each gene). In fact, since two trees can be constructed by clustering the microarray data, two animating treemaps can be produced by one microarray. For phylogenetic trees which are a static tree structure this is not possible, but the fact that the root of the tree is uncertain leaves a margin for implementing features concerning the designation of the root.

Specifically we propose to use the slice & dice and squarified algorithms in order to display the microarray cluster trees, below we will show these algorithms in detail:

```
Algorithm TreemapAlgo(Node root, Rectangle bounds, String orientation)
{
fillRectangle(bounds);
//switch orientation
String newOrientation = (orientation == "horizontal" ? "vertical" :
"horizontal");
if (root is internal node)
       for (each child node of root)
              {
              Rectangle newBounds = findBounds(root, child-node, bounds,
              orientation);
              TreemapAlgo(child-node, newBounds, newOrientation);
              }
}
```

This algorithm becomes easier to understand if we use an example:

Let us suppose we have the following tree:

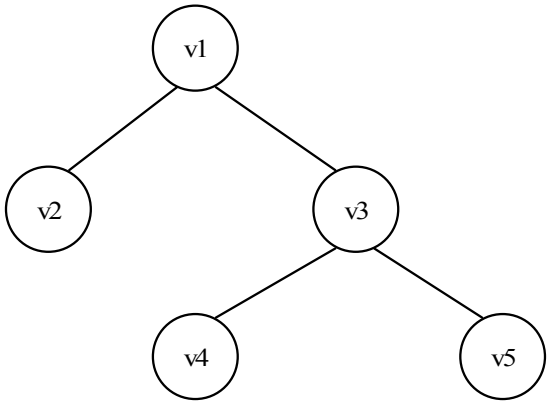


Figure 3.2: A simple tree used to demonstrate how a treemap is generated

And let us also suppose that the area to be filled is a rectangle with dimensions *w* by *h*:

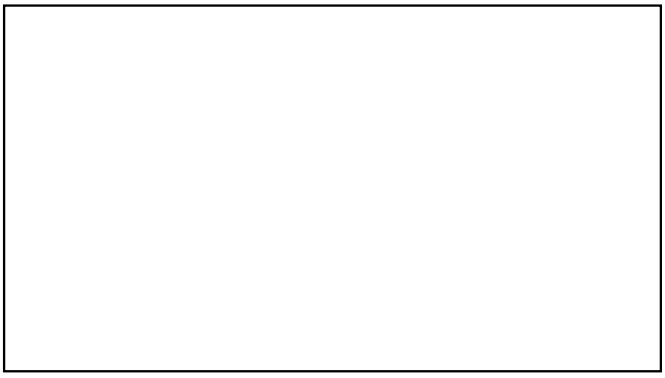


Figure 3.3: The rectangle which will hold the treemap

So the algorithm mentioned above will be called in this fashion:

```
TreemapAlgo(v1, {x, y, w, h}, "horizontal")
```

The symbol `{x, y, w, h}` denotes a rectangle with dimensions `w` and `h` for width and height respectively starting at coordinates `x` and `y`.

So firstly, the method `fillRectangle` is called which fills the initial rectangle with the color of the current vertex `root`. This color can be computed independently from the treemap algorithm, usually a linear color-coding is used (i.e. black for values of 0 and the brightest of a color for the maximum value with all other values being set in between). Because the color of each cell is irrelevant to the treemap algorithm, during this walkthrough of the algorithm no coloring will be performed.

After the `fillRectangle` method has returned, the treemap will look like this:

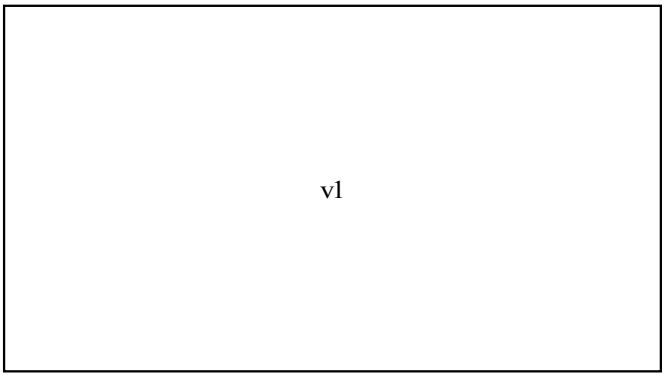


Figure 3.4: Vertex v1 is assigned to the entire rectangle since it is the root

The next line:

```
String newOrientation = (orientation == "horizontal" ? "vertical" :
"horizontal");
```

effectively switches the orientation of the layout from horizontal to vertical. So the variable `newOrientation` will hold a value of `"vertical"` to be used in the following recursive calls to `TreemapAlgo`.

The next line:

```
if (root is internal node)
```

checks whether the current vertex is internal or not, this is necessary because the algorithm is recursive and repeats itself for every child of the current vertex. For leaves the recursion terminates at this point since nothing follows this `if` statement.

The next line:

```
for (each child node of root)
```

is a `for` loop which repeats the following code for every child of the internal node `root`.

```
Rectangle newBounds = findBounds(root, child-node, bounds, orientation);
TreemapAlgo(child-node, newBounds, newOrientation);
```

This is the code that is executed for every child of the current non-leaf vertex `root`. The first line calculates a new rectangle from the old one taking into consideration the current child of node `root`: `child-node` and the current orientation. In this case the variable `child-node` would hold the vertex v2, `bounds` holds the rectangle `{x, y, w, h}` and orientation has a value of `"vertical"`.

The method `findBounds()` is a method which divides the given space (`bounds`) into *n* rectangles where *n* is the total number of children of `root` and returns the *i*-th rectangle where *i* is the position of the current child-node in the children list of its parent. But `findBounds()` does not compute equal rectangles because it takes into account the fact that the subtrees originating from the *n* siblings are not of the same size, so vertex `v2` will receive 1/3 and vertex `v3` will receive 2/3 of the allocated rectangle. This will be shown below in the pseudocode for `findBounds()`. Furthermore the way the `bounds` rectangle is divided depends on the `orientation` variable. The pseudocode for `findBounds()` follows below:

```
Rectangle findBounds(Node root, Node child, Rectangle {x, y, w, h}, String
orientation)
{
int numOfLeavesRoot = findNumOfLeavesDFS(root);
int numOfLeavesChild = findNumOfLeavesDFS(child);
int numOfLeavesUpToChild = 0;
```

```
int newX, newY, newWidth, newHeight;
for (each child node of root)
       {
       if (child-node == child)
              break;
       numOfLeavesUpToChild += findNumOfLeavesDFS(child-node);
       }
if (orientation == "horizontal")
       {
       newX = x + (int)((float)numOfLeavesUpToChild/numOfLeavesRoot * w);
       newWidth = (int)((float)numOfLeavesChild/numOfLeavesRoot * w);
       newY = y;
       newHeight = h;
       }
else
       {
       newX = x;
       newWidth = w;
       newY = y + (int)((float)numOfLeavesUpToChild/numOfLeavesRoot * h);
       newHeight = (int)((float)numOfLeavesChild/numOfLeavesRoot * h);
       }
return {newX, newY, newWidth, newHeight};
}
```

The `findNumOfLeavesDFS()` method is a simple method for traversing the tree with root the parameter given to the method and counting the leaves found. The quotient `numOfLeavesChild/numOfLeavesRoot` is the percentage of leaves this child has in the overall leaf count of its parent and determines the area which this child will occupy. The quotient `numOfLeavesUpToChild/numOfLeavesRoot` is the percentage of leaves the set of all children before this one have, this number represents the area all previous siblings have occupied and is used to determine the offset at which the current child's area will start. The below figure shows what happens for vertex v2:

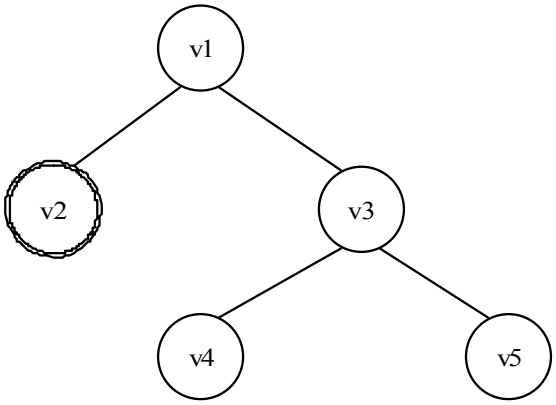


Figure 3.5: Suppose the current child is v2 and we wish to calculate the area it will occupy. It has only one leaf (shown with a double border). The quotient numOfLeavesChild / numOfLeavesRoot is 1 / 3

The quotient `numOfLeavesUpToChild/numOfLeavesRoot` is 0/3. So vertex v2 will receive 1/3 of v1's rectangle's width starting at 0/3 the width of v1's rectangle. So for v2 the rectangle `newBounds` will be `{x, y, 1/3w, h}`.

After findBounds() returns the aforementioned rectangle for vertex v2 the next line is called:

```
TreemapAlgo(child-node, newBounds, newOrientation);
```

where `child-node` is vertex `v2`, `newBounds` is `{x, y, 1/3w, h}`, and `newOrientation` is "vertical". Since v2 is a leaf, all that will happen is that the `fillRectangle()` method will be called which will assign the `{x, y, 1/3w, h}` rectangle to v2.

The below figure shows the treemap at this point:

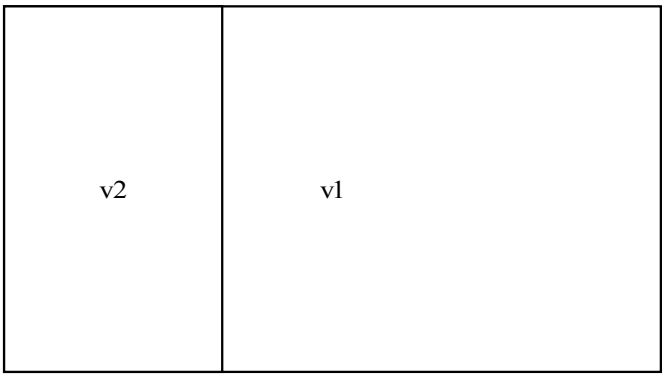


Figure 3.6: The treemap so far

Then the `TreemapAlgo()` will return to the for loop and the next child of `v1` will be processed.

The below figure shows what happens for vertex v3:

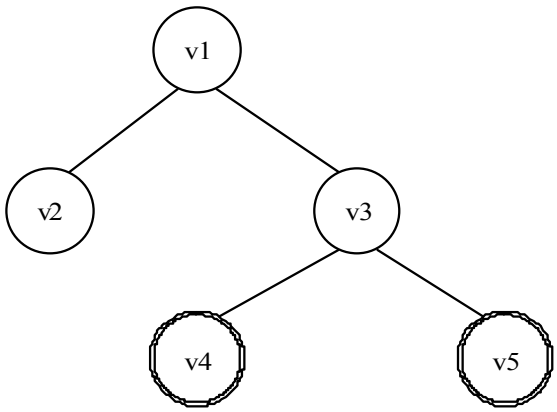


Figure 3.7: Now v3 is the current child. The quotient numOfLeavesChild / numOfLeavesRoot is 2 / 3

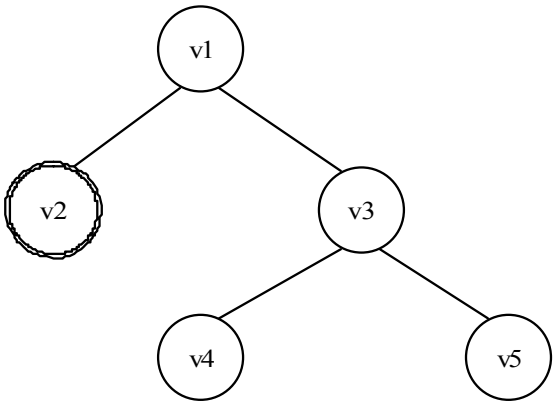


Figure 3.8: The quotient numOfLeavesUpToChild / numOfLeavesRoot which is 1 / 3 is shown in the above tree

So vertex v3 will receive 2/3 of v1's rectangle's width starting at 1/3 the width of v1's rectangle. So for v3 the rectangle `newBounds` will be `{x+2/3w, y, 2/3w, h}`. The figure below shows the treemap after rectangle v3 has been processed but before the algorithm has a chance to process vertex v4:

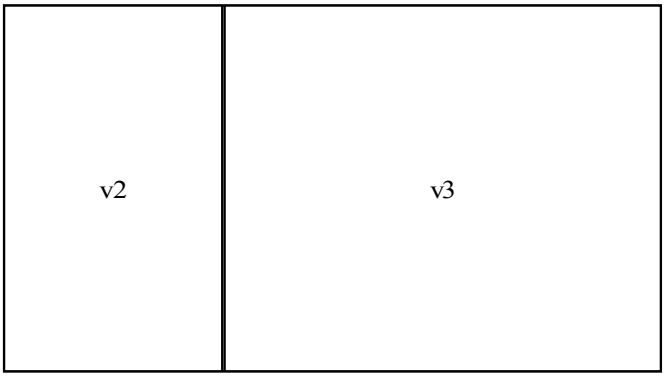


Figure 3.9: The treemap after vertices v2 and v3 have been given areas

After vertices v4 and v5 have been processed the algorithm completes and the areas given to each node are:

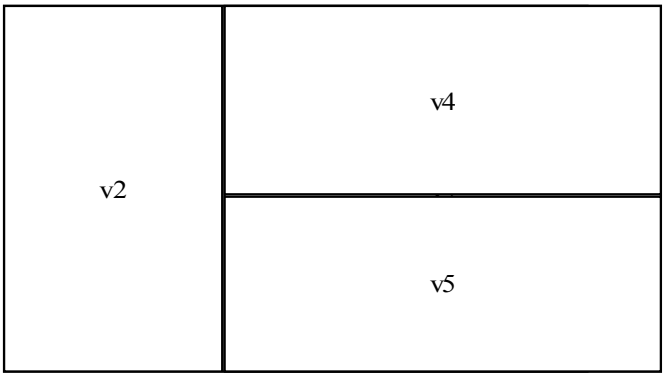


Figure 3.10: The output of the slice & dice algorithm

As can be seen above vertices v4 and v5 are processed identically to v2 and v3 except that the layout of the rectangles changes from horizontal to vertical. One can see the importance of this orientation in the following treemap where all nodes are laid out horizontally:

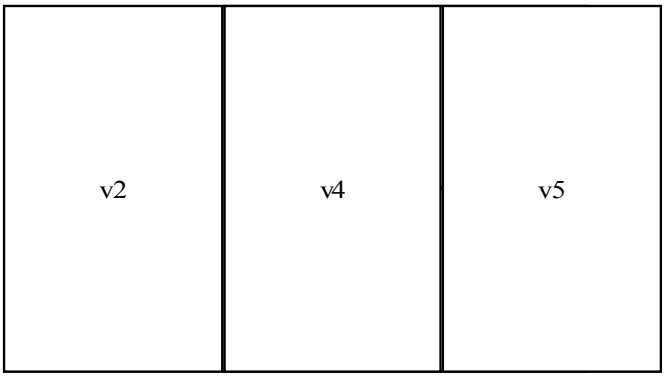


Figure 3.11: The problem the lack of orientation causes. The structure of the tree is lost

The slice & dice algorithm has the advantage that it is a relatively simple algorithm to implement and that it preserves the order of the siblings onto the treemap.

The slice & dice algorithm has the problem in that it produces nodes that are too thin when many siblings must be fitted into the same rectangle. For example:

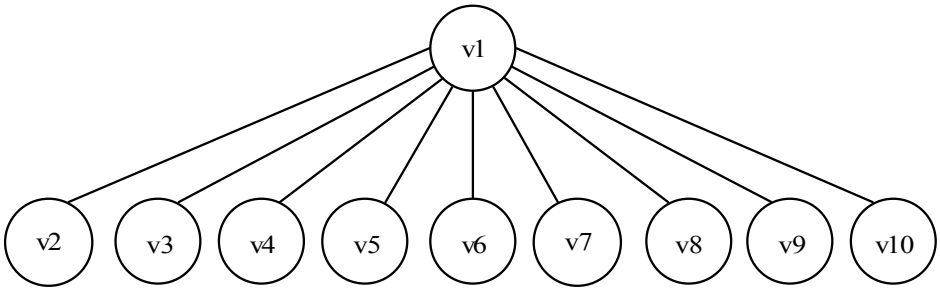


Figure 3.12: A simple tree

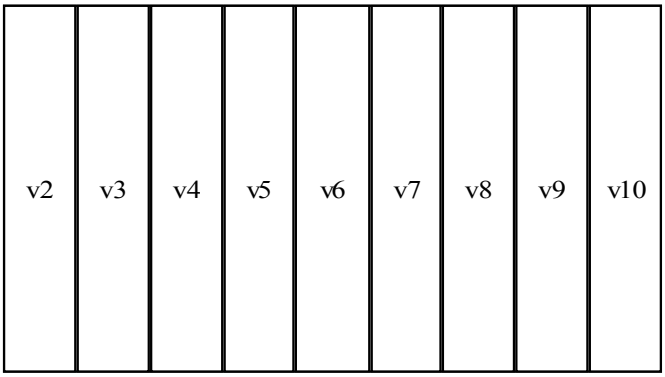


Figure 3.13: The slice & dice algorithm produces nodes that are quite thin

This disadvantage of the standard treemap algorithm is addressed with the squarified algorithm which is shown below:

```
Algorithm SquarifiedAlgo(Node root, Rectangle bounds)
{
List of Nodes children = root.getChildren();
List of Nodes row = [];
sortInOrderOfDescendingLeafCount(children);
squarifyLevel(children, row,  width(bounds), bounds);
for (each child node of root)
       SquarifedAlgo(child-node, child-node.bounds);
```

```
}

Algorithm squarifyLevel(List of Nodes children, List of Nodes row, Real w,
Rectangle bounds)
{
Node c = head(children);
if (worst(row, w) ≤ worst(row++[c], w)) then
       squarify(tail(children), row++[c], w, bounds)
else
       {
       Rectangle newBounds = layoutrow(row);
       squarify(children, [], width(newBounds), newBounds);
       }
}
```

The squarified algorithm works by storing all children of a node in a list, the method `getChildren()` is responsible for this. Afterwards the list is sorted so that the nodes with the biggest size are first, i.e. the nodes which have the most leaves originating from their subtree, the method `sortInOrderOfDescendingLeafCount()` does this. After that follows the `squarifyLevel()` method which does the main job of laying out the nodes in the `children` list onto the `bounds` rectangle so that the worst aspect ratio of the set of nodes is as close to 1 as possible. Finally this process is repeated recursively for every subtree of the original tree, identically to the slice & dice algorithm.

The `squarifyLevel()` method works by creating a row in the given rectangle along its short side. The short side is chosen so that the remaining subrectangle is as square as possible, the `width()` method does not compute the width of its parameter as its name implies, but instead it computes its shortest side. The algorithm adds nodes to the row while checking that the worst aspect ratio of the row *after* the addition does not become worse than the worst aspect ratio *before* the addition. If that is the case then instead of adding the node to the existing row, a new row is created while the old one is locked in place. The rectangle for the new row is the one that remains when the row is subtracted from the original rectangle. The `layoutrow()` method does this.

So the `SquarifiedAlgo()` is run recursively with respect to every subtree of the tree, and moreover the `squarifyLevel()` method is run recursively with regard to every row created by it.

The formula for finding the worst aspect ratio is shown below:

$$worst(R, w) = \max_{r_i \in R}(\max(\frac{w^2 r_i}{s^2}, \frac{s^2}{w^2 r_i}))$$

where R is the list of children and r_i the size of each child i, w is the parameter passed to the `squarifyLevel()` method denoting the length of the side along which the row is being laid out and s is the total size of all nodes in the row. Thus:

$$\frac{w^2 r_i}{s^2} = \frac{w^2 hd}{w^2 d^2} = \frac{h}{d}$$

which gives the aspect ratio.

This is shown visually below:

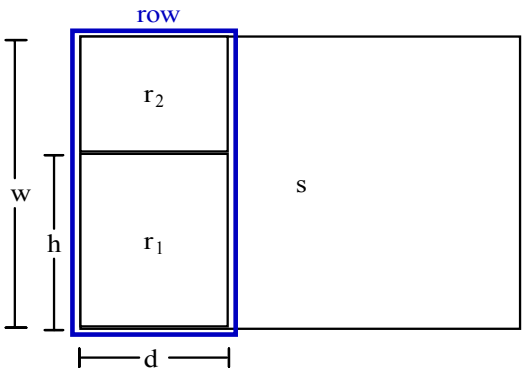


Figure 3.14: The symbols used in the computation of the worst aspect ratio shown visually

Since w^2r/s^2 increases for increasing values of r and $s^2/(w^2r)$ decreases for decreasing values of r the *worst()* formula can be written like this:

$$worst(R, w) = \max(\frac{w^2 r_{\max}}{s^2}, \frac{s^2}{w^2 r_{\min}})$$

where r_{max} and r_{min} are the maximum and minimum values of R.

Below is a graphical representation of the process of squarified layout for 7 nodes with sizes `{6, 6, 4, 3, 2, 2, 1}` into a rectangle with width 6 and height 4:

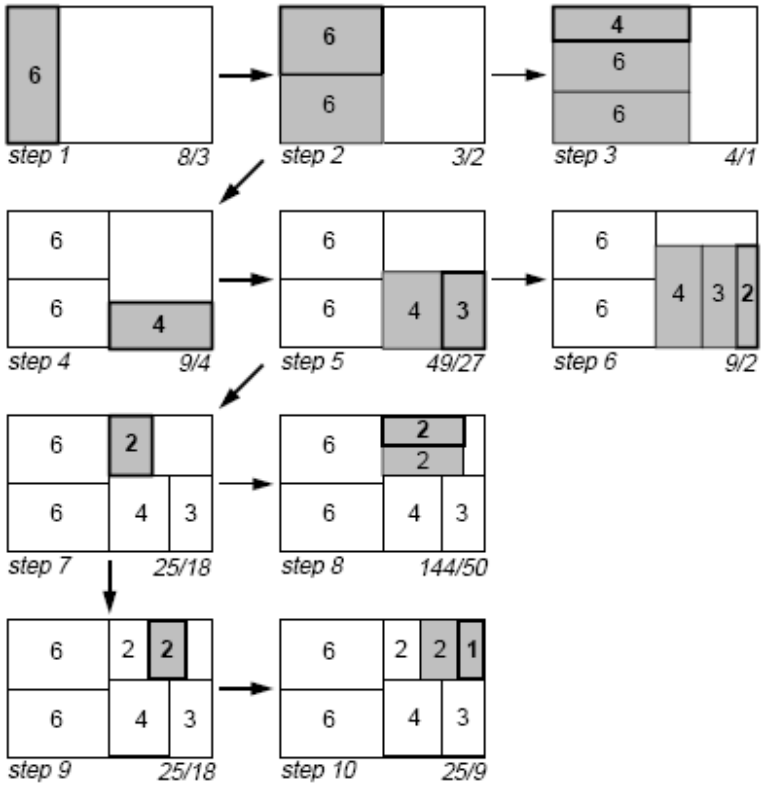


Figure 3.15: The process of creating a squarified treemap

The first row is created vertically (a column) on the left of the bounding rectangle. Since the node has a size of 6 and is fitted on a side with height 4, the other side has width 1,5. So the aspect ratio is 4/1,5 = 8/3. Afterwards the other node with size 6 is pushed on top and the worst aspect ratio is computed, which is found to be 3/2, so the row remains as is and we proceed to the next node which has size 4. After pushing it on top the column has an area of 16, so the width is 4 as is the height. The worst aspect ratio is 4/1 for the node with size 4. This solution is rejected, so the node with size 4 is used to create a new row in the remaining subrectangle with width 3 and height 4.

This time the row is horizontal because the shortest side is the horizontal and the node with size 4 is placed there. The width of the row is 3 so the node's height is 4/3, therefore the aspect ratio is 9/4. The next node in the list has a size of 3 and is pushed to the right of the row. The *w* variable has a value of 3 and the *s* has a value of 7, so the worst aspect ratio is that of the node with size 3 and is 49/27. 9/4 is 2,25 and 49/27 is less than 2, so we don't lock the row and proceed with the addition of the next node. After adding the next node with size 2 the *s* variable has become 9 and $w^2r^+/s^2 = 3^24/9^2 =$

36/81, $s^2/(w^2r^-) = 9^2/(3^22) = 81/18 = 9/2$. So the worst aspect ratio is 9/2, a quite thin node indeed and the row is reverted to its previous state while the node with size 2 is assigned to a new row in the subrectangle with width 3 and height 5/3. The rest of the computations proceed as before until the list of nodes empties. At this point the `squarifyLevel()` method returns and the `SquarifiedAlgo()` method creates a new list from the children of the next node encountered during the tree traversal.

The squarified algorithm produces better results when the biggest nodes are handled first, this is the reason why the list of nodes is sorted before the layout algorithm. Because of this rearrangement the order of siblings which are displayed in the treemap differs from the original tree, this is the main drawback of this method.

Besides the visualization of clustered microarrays as animating treemaps, a hybrid visualization combining treemaps with the standard tree display is proposed:

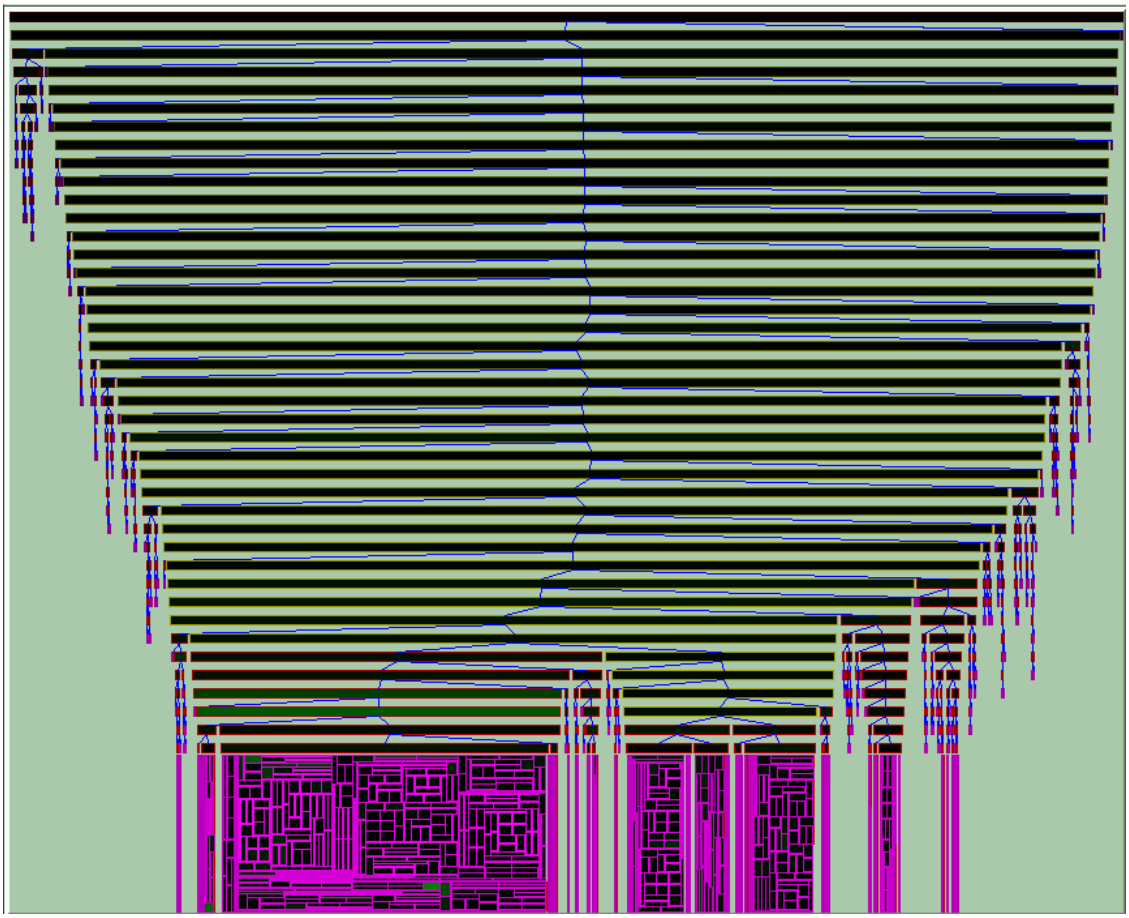


Figure 3.16: A combination of treemaps and the standard tree visualization, the width of each node is proportional to the total number of leaves originating from that node

Concerning microarrays the classification of the microarray must also be considered in order for the software tool to be effective. So various clustering methods [5] are implemented but since these algorithms take exponential time the datasets are quite small. This is a problem for future consideration.

4. Treemaps for Data Analysis

The following are useful features which apply generally to any kind of data loaded into the program. They are placed into their own chapter because of their significance and their usefulness.

4.1. Clustering

The software has the feature of applying hierarchical clustering with various cluster methods (single linkage, complete linkage, UPGMA, UPGMC, WPGMA, WPGMC) [5] and distance functions (euclidean, correlation coefficient, 50-50 between euclidean and correlation coefficient) to the data provided. Note that this feature can serve as a generic clustering tool for any kind of data which is loaded into the program as an array of *r* rows versus *c* columns. However for the rest of this document we will assume that the data loaded is a microarray. The reason for this is that:

(a) this feature was initially developed with microarrays in mind, and

(b) that regarding the data as something tangible rather than abstract has the advantage that all data components have a meaning (i.e. a cell in the array is a gene expression, a data cluster is a cluster of genes etc.)

Due to the nature of the clustering methods, during the computation of the distance between two clusters every subcluster of the first cluster is recursively compared to every subcluster of the second cluster. Therefore for every single distance computation the recursion always reaches the leaves of both clusters. Thus due to this exhaustive computation the clustering takes exponential time.

Note that the correlation coefficient used in this software is not the correlation coefficient in the strict mathematical sense. Normally for *g* genes every gene would receive *g* correlation coefficients (one by comparing it to each other gene plus correlation coefficient 0 for itself). So a lot of data would be produced and the clustering would be computationally intensive. Instead of that every gene and patient is only compared to an arbitrary *index vector* (0, 1, 2, …) which comes from the genes'/patients' sequence of gene expressions. The main problem with this is that simply by swapping two gene expressions the correlation coefficient changes (which normally it shouldn't). Although swapping gene expressions doesn't happen in this software, it shows the sensitivity of

this metric. Despite the aforementioned fact, this correlation coefficient is reliable because swapping position *i* of a gene's gene expression vector with position *j* affects all correlation coefficients in the same way since all vectors must experience a swapping of those two positions. Besides that, this metric is much faster than the typical correlation coefficient.

Initially the user selects the clustering algorithm, only hierarchical is implemented at this point:

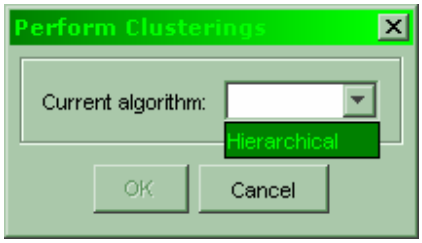


Figure 4.1.1: The clustering dialog

Once the user selects the hierarchical algorithm, its respective parameters appear and the user may choose the cluster method and the distance function:

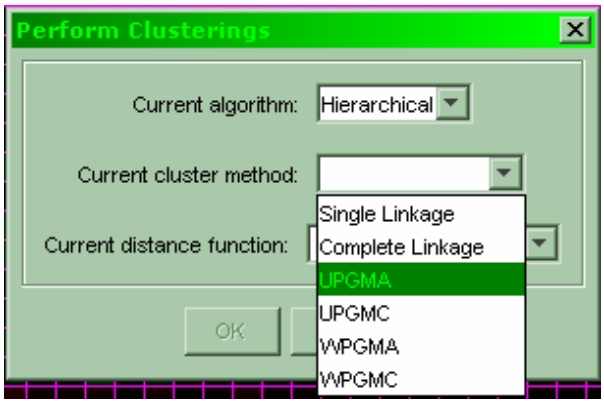


Figure 4.1.2: The cluster method selection for the clustering dialog

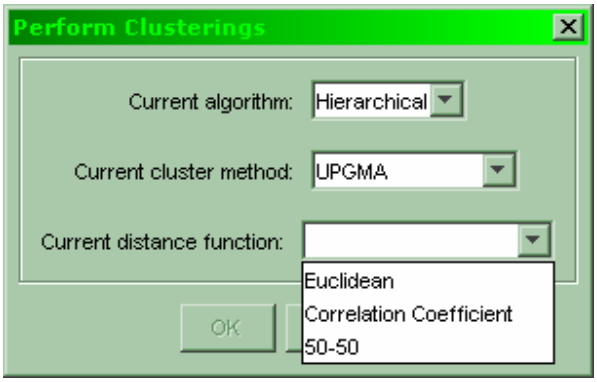


Figure 4.1.3: The distance function selection for the clusterings dialog

After the user selects the parameters the program performs the clustering, two progress windows appear and display the progress of the clustering, and each window shows one clustering:

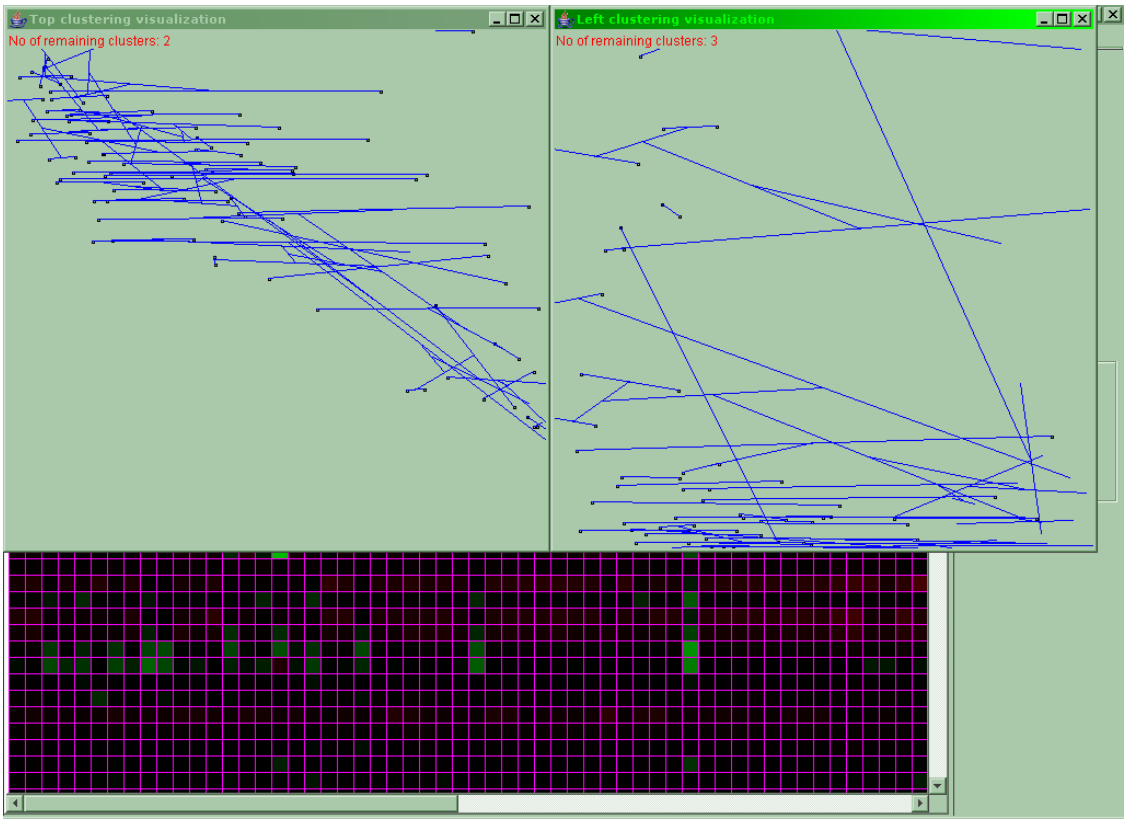


Figure 4.1.4: The monitoring windows which display the progress of the visualization

These are a rough visualization of the clustering progress by symbolizing every set of gene expressions to be clustered (either through their mean value, or their correlation coefficient, or a 50-50 between the two) with a small square on the window.

Every set which is clustered with another receives a blue link to its peer. This blue line symbolizes the cluster joining these two nodes. Should a cluster be involved with another cluster instead of a node then the blue line starts at the middle of another blue line etc. The x-axes of these visualizations represent the mean values and the y-axes the correlation coefficients. In the top left corner of every display the user may read the number of clusters remaining to be joined. When this counter drops to two the clustering is complete since for two clusters the join does not require the algorithm's calculations. It is again noted that these are rough visualizations with a main purpose to display the progress of the clustering and provide a visual aid as to when the clustering process will be completed.

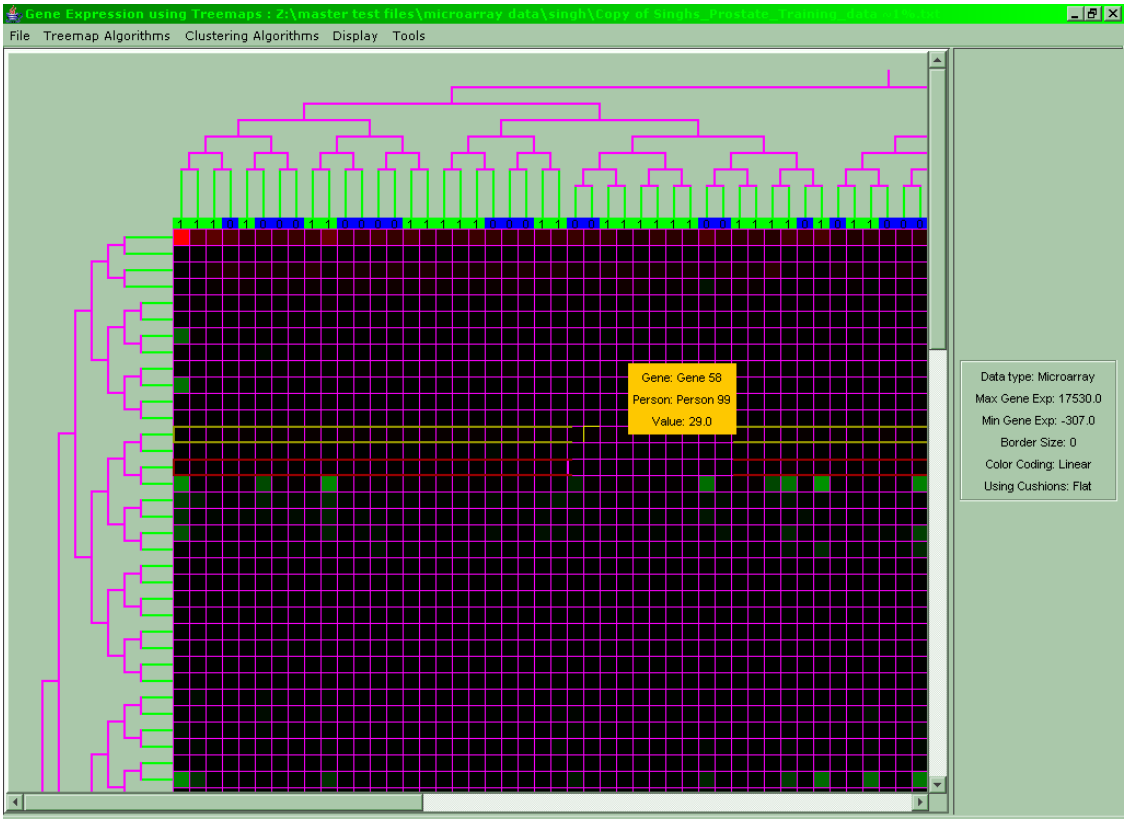


Figure 4.1.5: The microarray after the clustering has finished

4.2. Change of the color-coding

The user may change the way the color intensity corresponds to the numerical values of each node. Normally there is a linear coding from color intensity ranging from 0 (black, for a value of 0) to full intensity (for the max value among all values in the treemap currently displayed, for microarrays this is the max absolute value since there can also be negative values). But the user may –instead of a linear function- use an exponential (or logarithmic function) in order to make slight differences at the high end (or low end respectively) easier to spot due to the steepness of the exponential/logarithmic function's graph at the high/low end.

The figures below show three treemaps with different color-coding resulting in different colors for the same node:

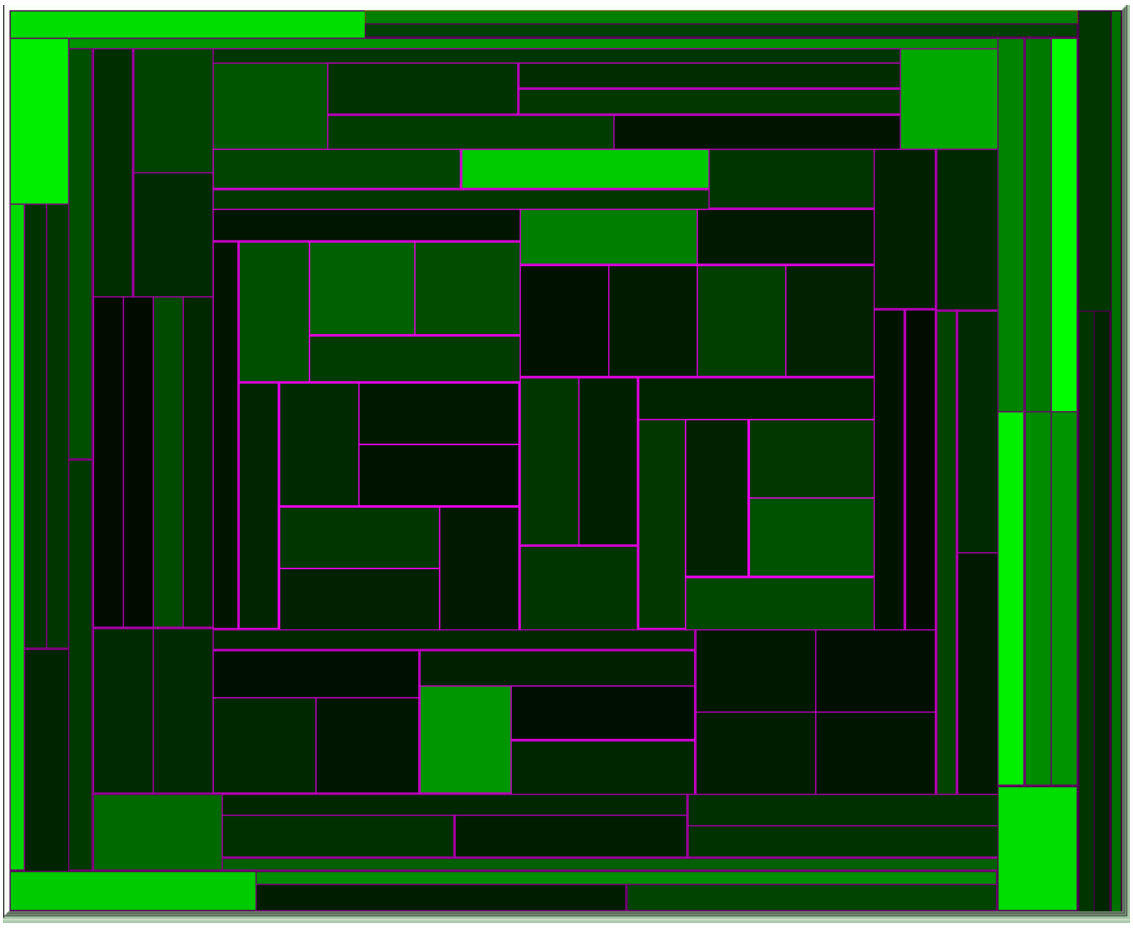


Figure 4.2.1: A treemap with linear color-coding, differences between nodes at high intensity levels are the same as at low intensity levels

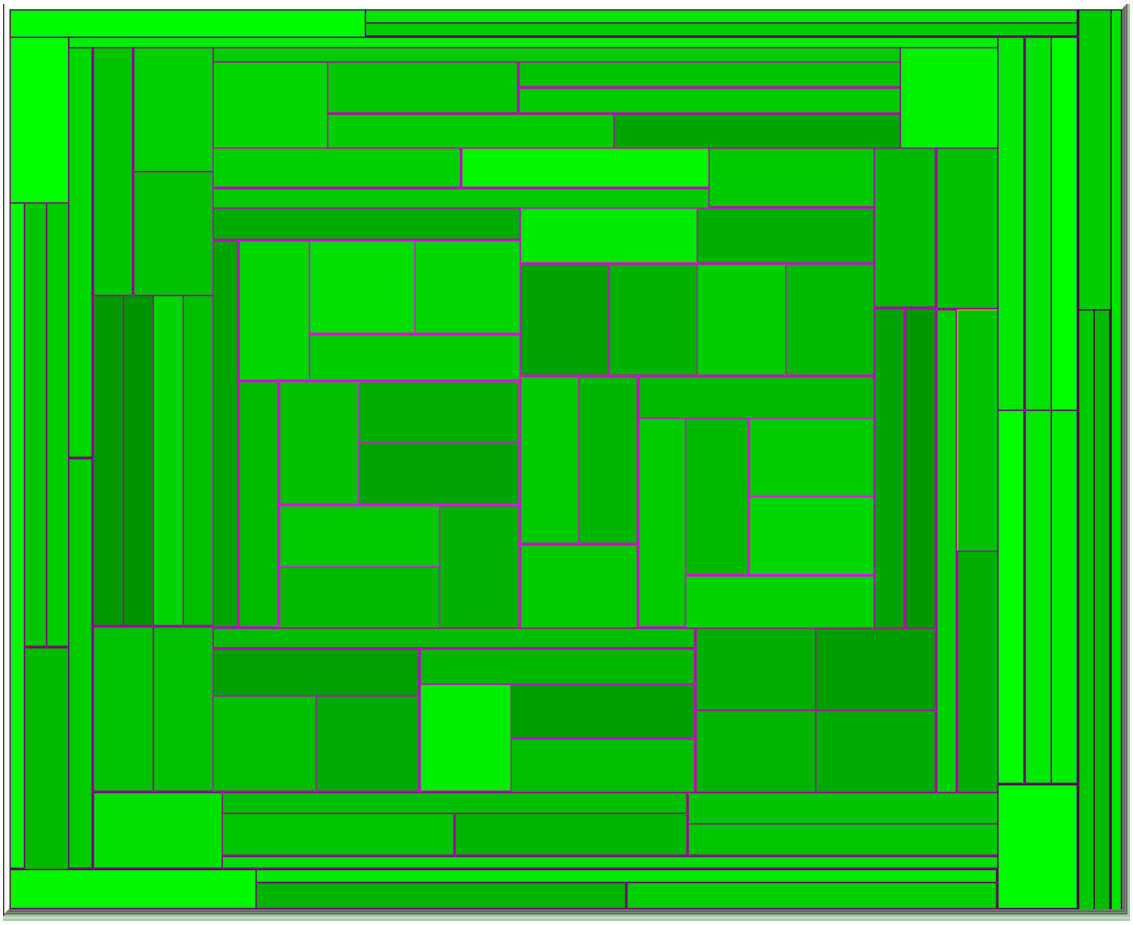


Figure 4.2.2: The treemap from figure 4.2.1 with logarithmic color-coding, differences between nodes at low intensity levels are emphasized while high intensity levels all look the same

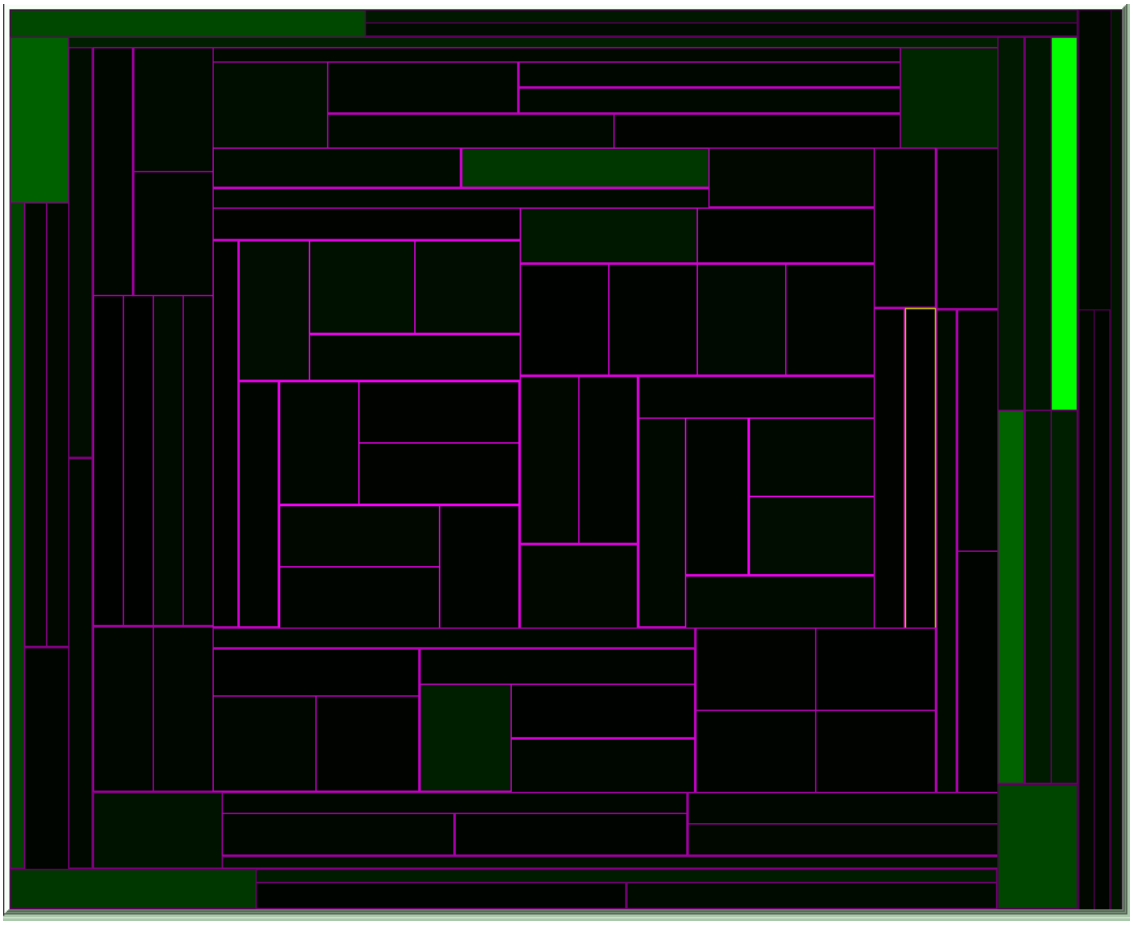


Figure 4.2.3: The treemap from figure 4.2.1 with exponential color-coding, differences between nodes at high intensity levels are emphasized while low intensity levels all look the same

The graph below shows the aforementioned functions:

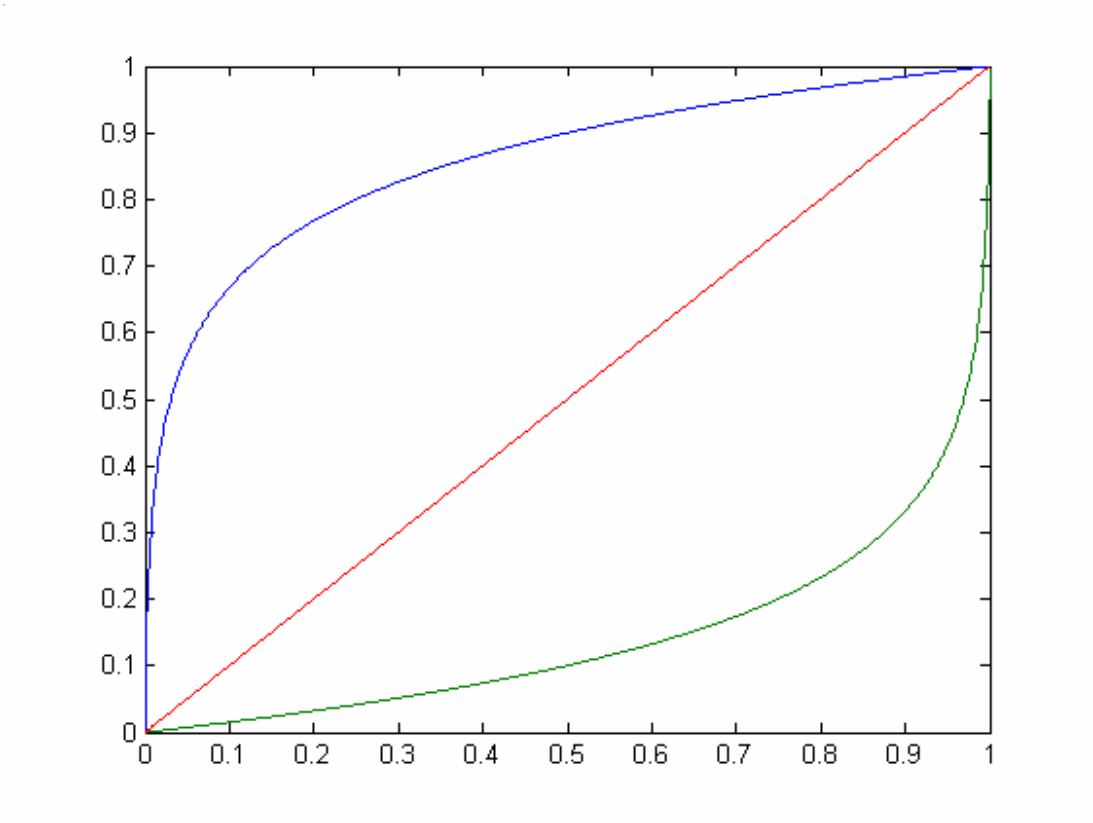


Figure 4.2.4: A plot with the x-axis being the node weight (normalized) and the y-axis being the color intensity

The functions are:

`logColor(x) =` $\log_c(x(c-1)+1)$ `(blue),`

`linearColor(x) = x (red) and`

`expColor(x) =` $\log_{1/c}(x(1/c-1)+1)$ `(green).`

As can be seen `logColor` and `expColor` are identical except for the constant `c` and `1/c`. So we use a log function with base < 1 to simulate an exp function and to retain a perfect mirror image of the original log function. The constant `c` can be in the range (1, +∞) but values over 10000 yield no significant difference, the above screenshots and graph are all with `c` = 1000. Note that the cell value is normalized to the range [0, 1] as can be seen in the x-axis of the graph. The software allows the user to select the value of `c` between 2 and 10000.

4.3. Display of a clustered microarray as an animating treemap

After a microarray has been clustered, two clustering trees have been created as can be seen in figure 4.3.1:

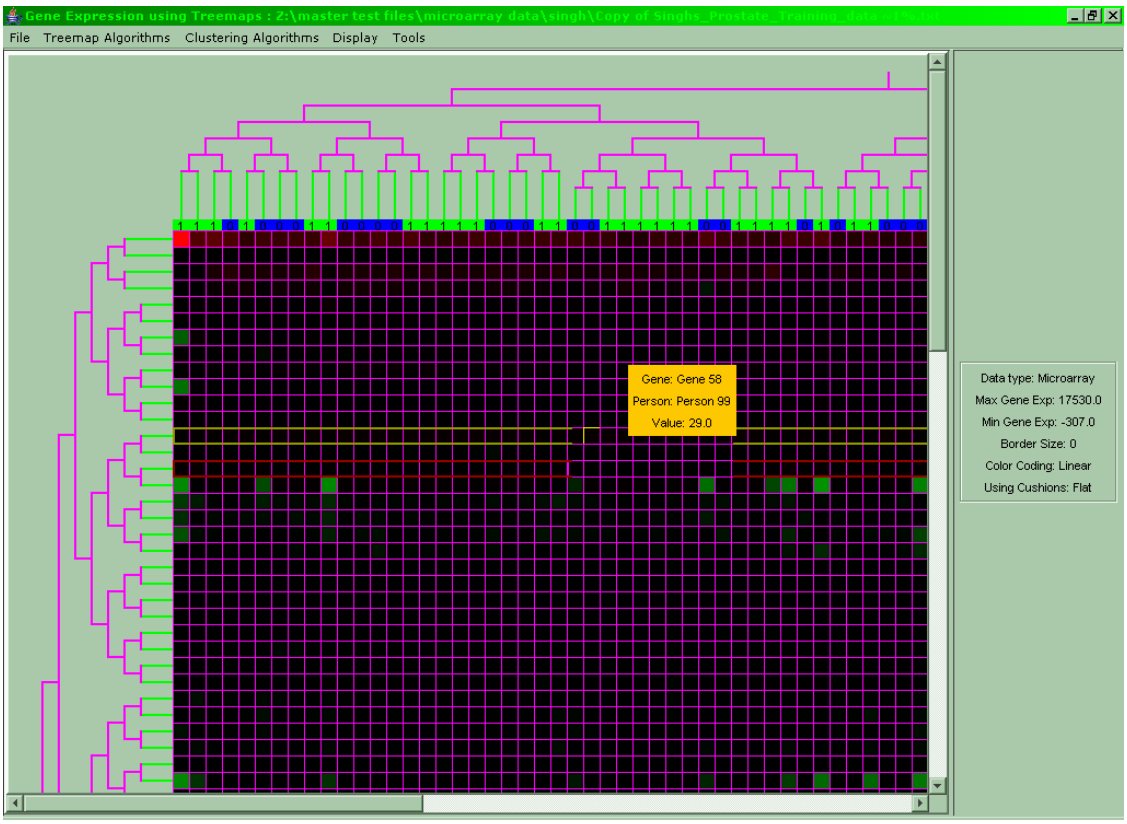


Figure 4.3.1: A clustered microarray

The user has the option of displaying these trees as animating treemaps as shown below:

Figure 4.3.2: A frame from a double animating treemap, the top treemap is the patient clustering (the top tree) and the bottom the gene clustering (the left tree), the classification can be seen in the animation bar of the gene clustering

Above each treemap is its respective animation bar with a small white cursor showing the current frame. The user may at this point control the speed of the animation by adjusting a slider. To show how the classic visualization of the clustered microarray (figure 4.3.1) leads to figure 4.3.2 let us see figure 4.3.2 in another way:

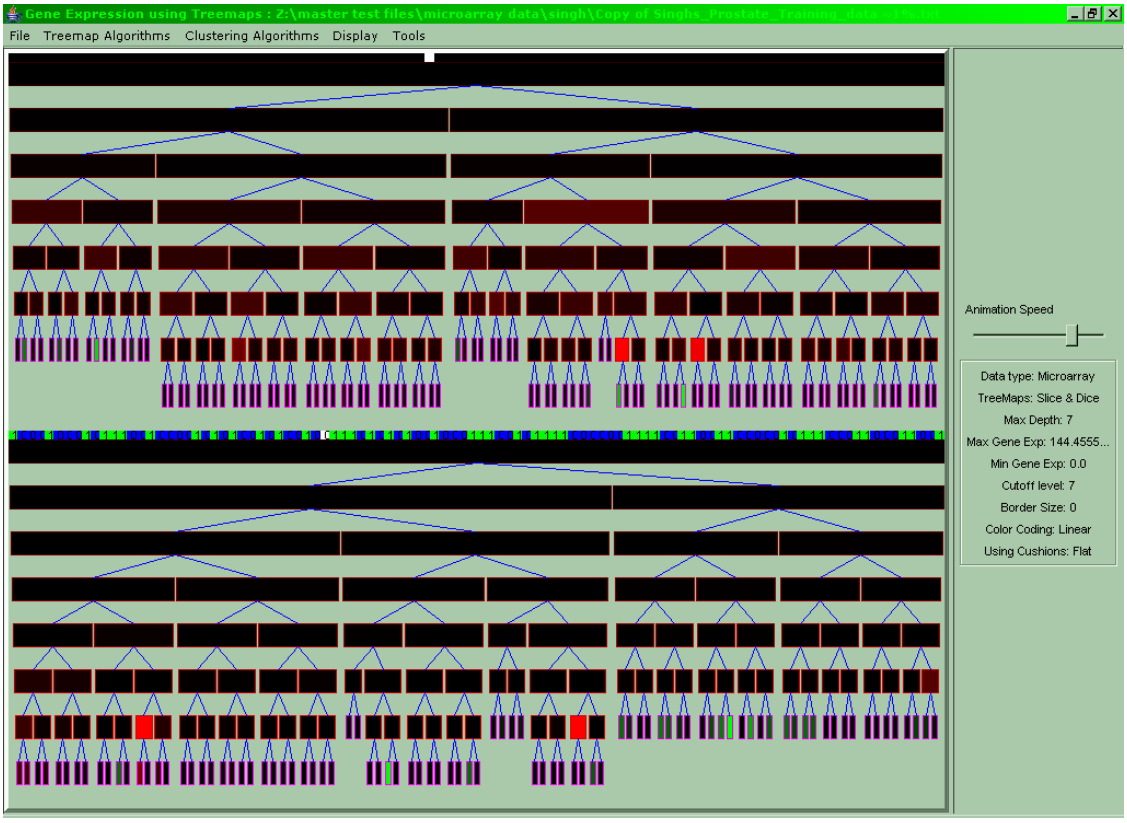


Figure 4.3.3: The animating treemaps shown as standard trees

In the figure above we can see that the above tree is the same as the top tree from figure 4.3.1 while the bottom tree is the same as the left one from 4.3.1.

The frames are sets of leaf values which produce a different tree as can be seen in the figure below:

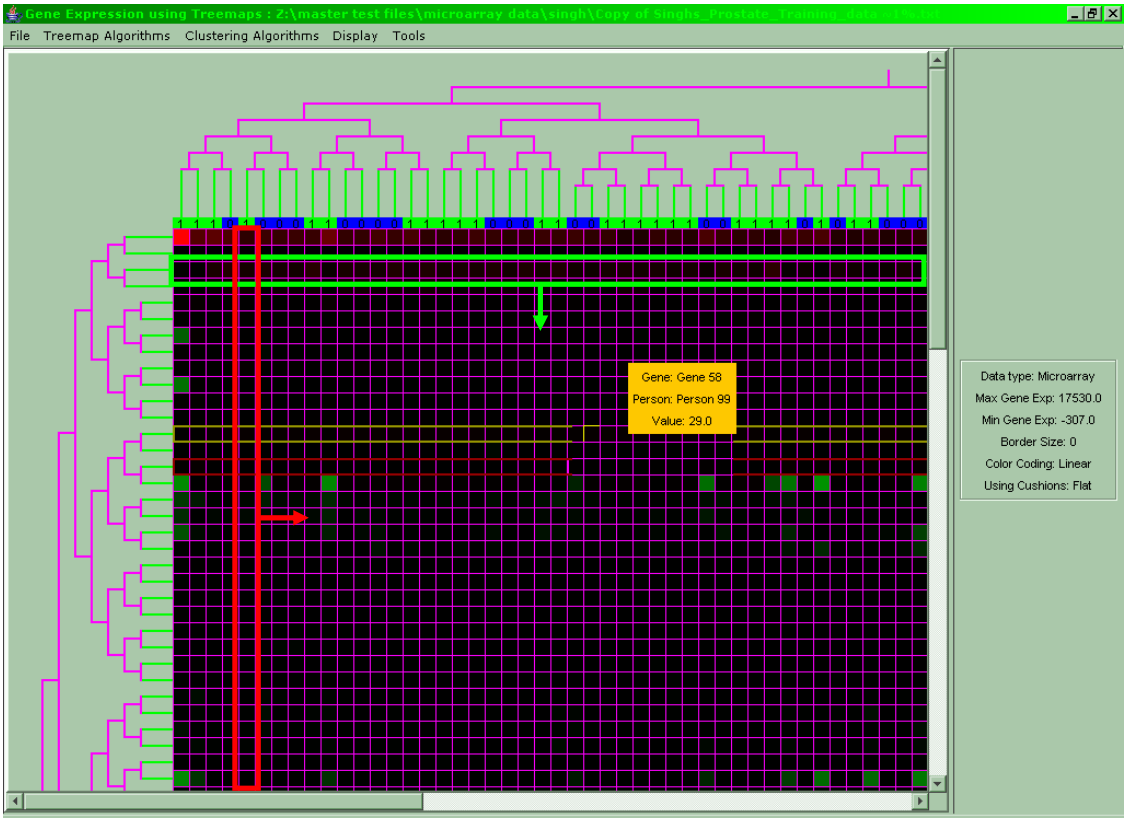


Figure 4.3.4: The green rectangle denotes frame #3 of the top treemap while the red rectangle denotes frame #5 of the bottom treemap. The arrows point to how the frames progress

5. Utilizing Treemaps in Microarray Data Analysis

5.1. Gene-Selection via Treemap Animation

The following figure shows the *patterns* of selected genes after inspection of the animated treemaps. The respective clusters were identified by their *color-difference* that reflects their *'long' distance* from other sub-clusters in the hierarchical treemap view. Inspection was done by letting the program run over all patient samples and identifying different-color (red or green) treemap-cells. We name this gene-selection methodology '*treemap gene-selection methodology*'.

As it can be observed from this figure and the respective patterns the selected genes show a *clear difference in their patterns* with respect to ALL and AML leukemia types. In other words, with the described treemap methodology (inspection of animated treemaps) *we are able to identify genes that discriminate well between the two classes* (ALL vs. AML leukemia types).

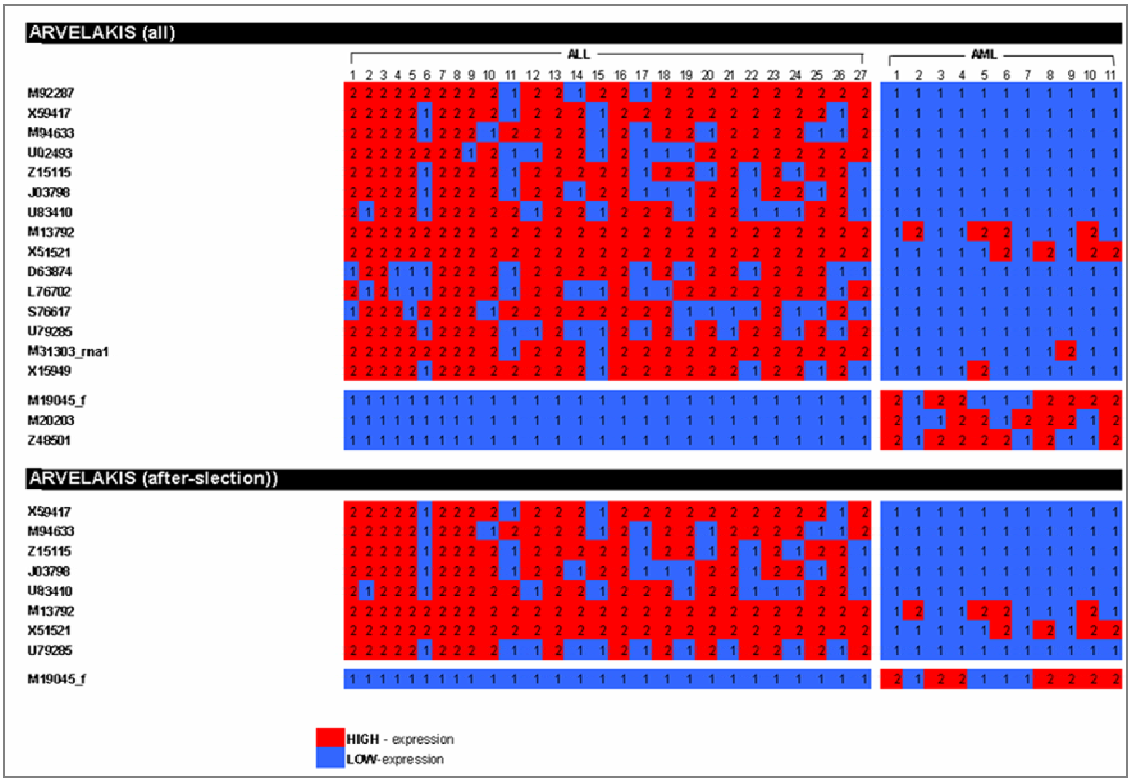


Figure 5.1.1: Patterns of selected-genes (red: high expression, blue: low expression)—genes are selected after treemap animation (running over all patient samples) and identification of 'long' distant sub-clusters of genes (i.e., red/green-colored subclusters)

Accuracy assessment

To validate the results we also performed an accuracy assessment. Table 1, below, shows the accuracy assessment results between the treemap selected genes and the genes that were selected in the original study (`STUDY – Leukemia-Golub`) (see ref. [20]).

In this assessment we compare the *fitness* accuracy (i.e., selected genes from training data and prediction results on the same training data), and the *LOOCV* (Leave One Out Cross Validation, i.e., selected genes from all training data excluding '1sample' and prediction of the class of the left-out '1sample' – then accuracy is assessed on the basis of the prediction over all left-out'1samples').

For both fitness and LOOCV we assessed accuracy results when: (a) *all* selected genes are used (for the treemap gene selection methodology and for the genes selected in the original study, i.e., 50 genes are reported in the study), and (b) after applying a gene-selection methodology on all (treemap selected genes) and on all 50 reported study genes. For this we used a specially devised gene-selection algorithm and tool (named '*MineGene*', see references [21], [22]).

From these results we could make the following observations:

i. When all treemap selected genes are used (18 genes) the study selected genes (50 genes) exhibit better performance -- 97.4% vs. 96.8% for fitness, and 73.7% vs. 60.5% for LOOCV.
ii. When we used MineGene and used the respective (treemap vs. study) selected genes (18 vs. 50), from which to select the most discriminant ones, the treemap gene-selection methodology is clearly superior to the respective study selected genes – 92.1% vs. 81.6% for fitness, and 84.2% vs. 65.8% for LOOCV. With MineGene we were able to select 9 genes (from the 18 treemap selected) and 13 (from the 50 study genes). This means, that the treemap gene-selection methodology presents a valid and reliable approach for the identification of discriminant genes, at least as a 'starting point'.

In the light of the above observations, the treemap gene-selection methodology could be utilized as a reliable *pre-filtering* gene-selection approach.

ARVELAKIS (all) vs. STUDY (all) ... alll genes identified by **treemap-animation** and '***identification of distant clusters***' are used

	#Genes	Fitness		LOOCV	
ARVELAKIS	18	86.8%	33/38 (ALL: 26/27; AML: 7/11)	60.5%	33/38 (ALL: 22/27; AML: 1/11)
STUDY (Leukemia-Golub)	50	97.4%	37/38 (ALL: 26/27; AML: 11/11)	73.7%	28/38 (ALL: 23/27; AML: 5/11)

ARVELAKIS (after-selection) vs. STUDY (after-selection) ... the same as above but ***gene-selection*** is applied on the identified genes

	#Genes	Fitness		LOOCV	
ARVELAKIS	9	92.1%	35/38 (ALL: 25/27; AML: 10/11)	84.2%	32/38 (ALL: 24/27; AML: 8/11)
STUDY (Leukemia-Golub)	13	81.6%	31/38 (ALL: 27/27; AML: 4/11)	65.8%	25/38 (ALL: 25/27; AML: 0/11)

Table 5.1.1 Accuracy assessment results – treemap gene-selection methodology vs. study selected genes.

5.2. Identification of Clinical Classification Irregularities via Gene-Expression Data: A Treemap Animation Approach

Suppose that the given samples are assigned to some clinical classification (e.g., disease or, disease type, as in the case of AML vs. ALL leukaemia types). It is interesting if we can identify samples that seems 'not' to fit the representative gene-expression patterns of these clinical classes.

In the case of the leukemia data we start from a set of 'best selected' genes. We visualise these data with treemaps and we animate the treemaps running over all patient samples. The hypothesis to confirm is the following: could we identify samples or, clusters of samples that deviate from all the others. In treemap terms this means to identify different colours of treemap-cells (e.g., green or, red in contrast to black).

- Following this methodology we were able to identify a sample, patient 20, the gene-expression of which *significantly deviates* from the other patients' gene-expression patterns. In particular, and based on a set of 220 best-selected genes, the average distribution of high-, and low-expressed genes is 56, and 162 (in 220), respectively. The respective distribution for the sample of patient 20 is 144, and 74, and it is the only sample with such big deviation.

So, we may conclude that with the inspection of animated treemaps we may be able to *identify peculiarities in the clinical classification* of patients, based on their gene-

expression profiles. This may help to filter-out such cases that may obscure the identification of discriminate genes.

Moreover, identification of such samples may guide to the re-classification of the targeted disease. In particular, for the leukemia study, there are gene-expression analysis studies (not published) that guide to the identification of such samples and propose the presence of a potential another leukemia type, besides the ALL and AML.

6. Working with Treemaps: Special Features

The following are features included in the software which don't have the gravity of the ones described in chapter 4 but are nevertheless important.

6.1. Supported file types

This visualization tool has several features. Firstly there are three different file types that can be loaded:

- Weighted adjacency lists can be loaded as long as the graph stored in them is a tree.
- Microarray files in either text or excel format (excel files can contain gene and patient names, and text files can contain a classification line on top consisting of 0s and 1s) can be loaded.
- Phylogenetic tree files in *newick* tree format. Furthermore the user is given the option to load an *order* file of the previously loaded phylogenetic species so that the orders of the species can be displayed. This order file is a simple text file where each line contains the species name followed by a comma and then the name of the species' order.

Microarrays are initially shown in their standard matrix display, so the size of each node is fixed. The color of each node represents the expression of each gene, the standard coloring (red meaning positive, green negative and grey a value that is not available) is used. For phylogenetic trees the color of each node represents the distance from that node to its parent since in phylogenetic research the distance between nodes is an important element. For adjacency lists the color used in nodes is green.

6.2. The popup window

This window appears whenever the mouse pointer is above a node and displays useful information about this node with regard to the current context (simple, microarray, phylogenetic). Below are various figures from all kinds of popup windows:

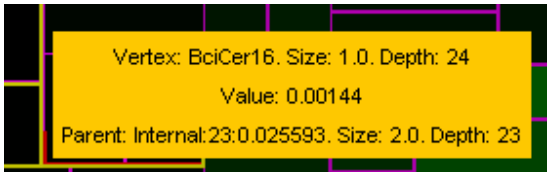


Figure 6.2.1: The popup window of a phylogenetic node in compact mode, size denotes the number of leaves originating from this node and value (weight) the distance from this node to its parent

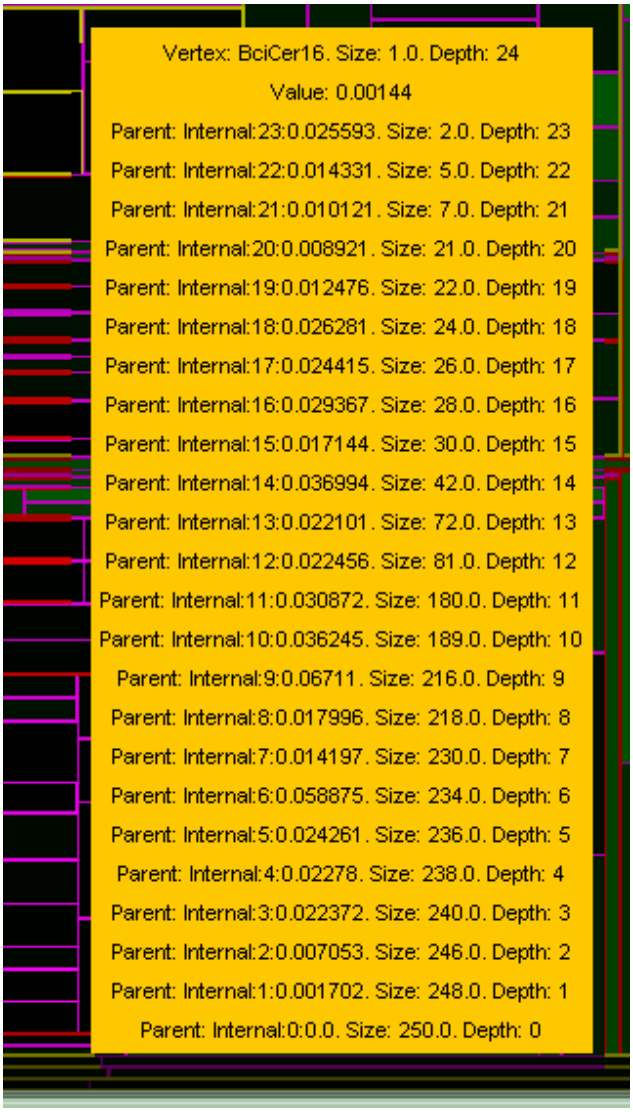


Figure 6.2.2: The popup window of a phylogenetic node with the full node path to the root visible

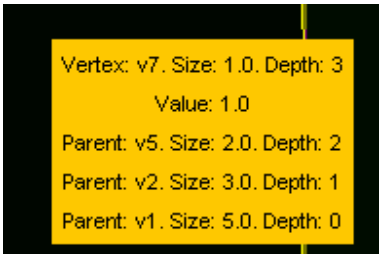


Figure 6.2.3: For simple trees the popup window is identical with that of phylogenetic trees

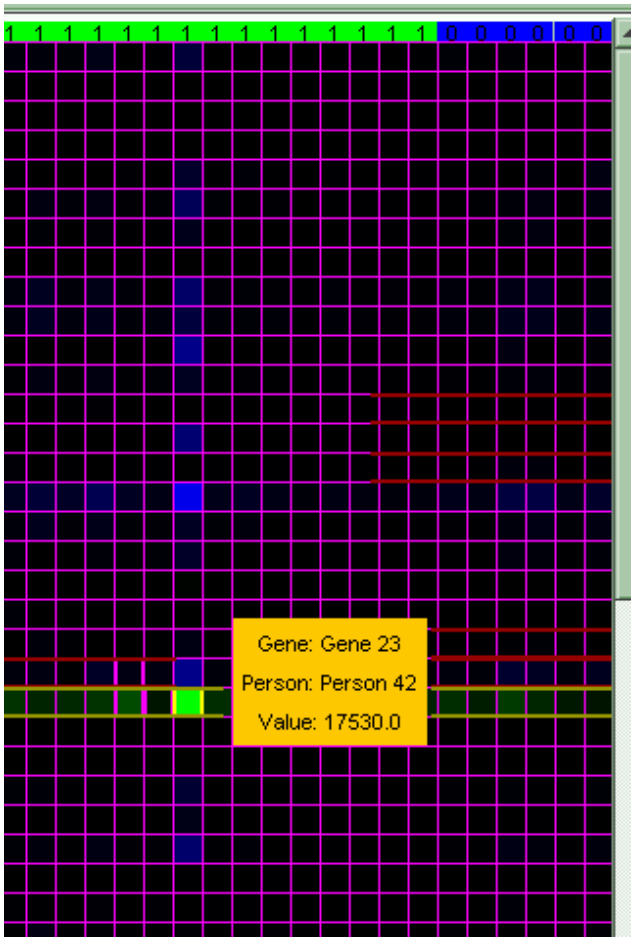


Figure 6.2.4: The popup window for a microarray file, the data here is different, value is the current gene expression

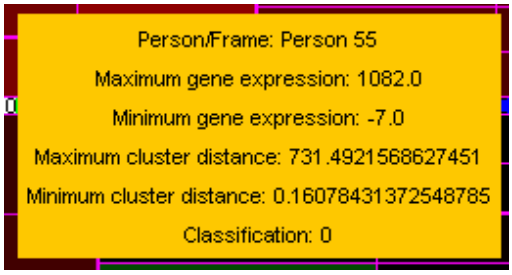


Figure 6.2.5: The popup window for animating microarrays when the pointer is over a classified frame

6.3. Features which apply generally to treemaps

6.3.1 Supported treemap algorithms

As long as treemaps are displayed on screen several features are available:

Firstly, one can select the display algorithm for the treemaps. Two algorithms are implemented: the standard slice & dice and the squarified algorithm. The reason for those two is that they represent the two edges of the aspect-ratio-versus-preservation-of-order spectrum.

The figures below show the same phylogenetic tree displayed as a slice & dice and as a squarified treemap:

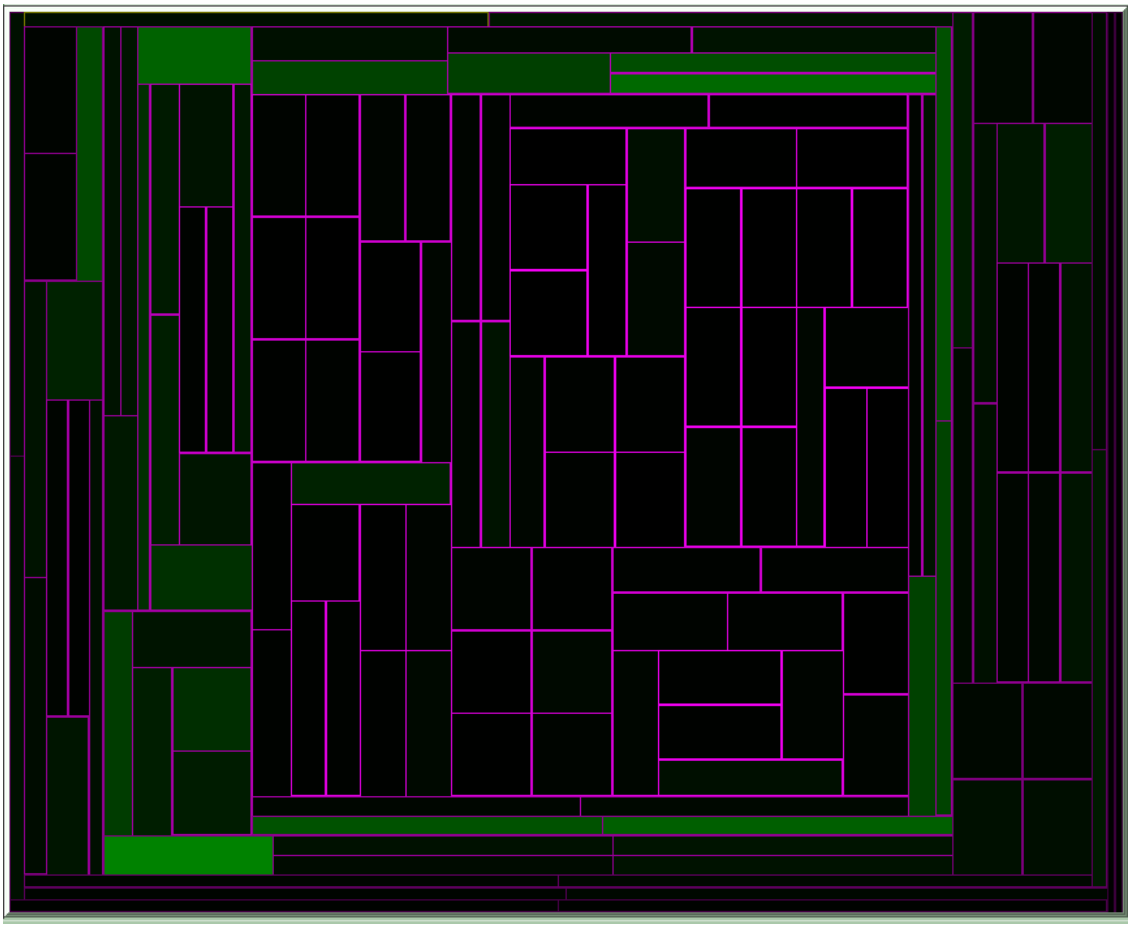


Figure 6.3.1.1: A treemap using the slice & dice algorithm

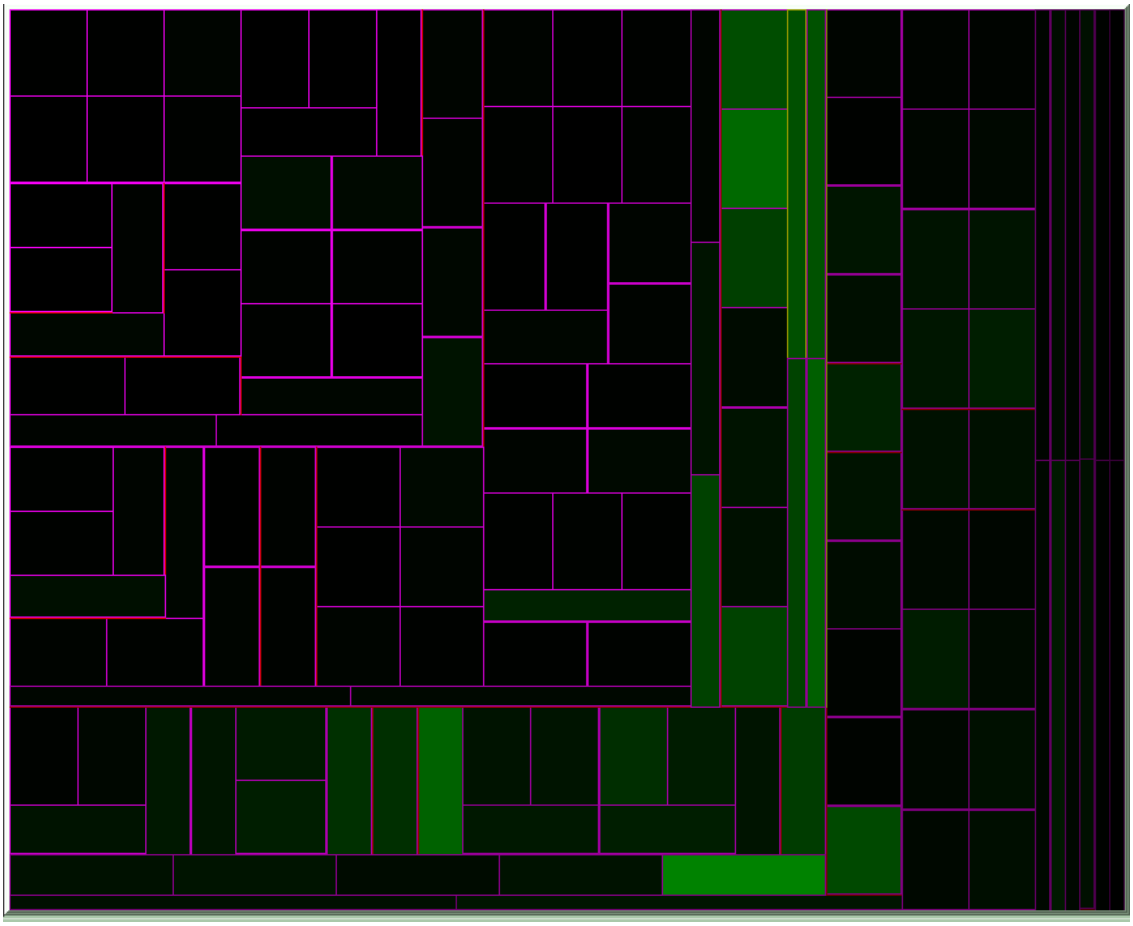


Figure 6.3.1.2: The treemap from figure 6.3.1.1 using the squarified algorithm

6.3.2. Addition of borders

Another option the user has is to adjust the border size of the treemap from 0 pixels which means completely overlapping treemaps with the internal nodes completely invisible, up to 4 pixels which produces a treemap with highly visible internal nodes but very small leaves. It should be noted that with large trees where node rectangles are quite small the activation of a border may make nodes at the top of the treemap (the deepest nodes of the tree) disappear due to lack of display space. Utilizing this feature was trickier than it seems at first because the approach of pushing the rectangle to be used as a treemap by *b* pixels to the right and to the bottom and reducing the width and height by 2*b* pixels (where *b* the border value) does not give good results for siblings because between the siblings the distance becomes 2*b* instead of *b*, this is easy to remedy for slice

& dice treemaps where the number of siblings placed side by side is fixed, but not for squarified treemaps where other -more specific- methods have to be used.

The figures below show the same phylogenetic treemap with border values 0, 2 and 4, note the decrease in the number of nodes with magenta borders (the leaves) during the three figures:

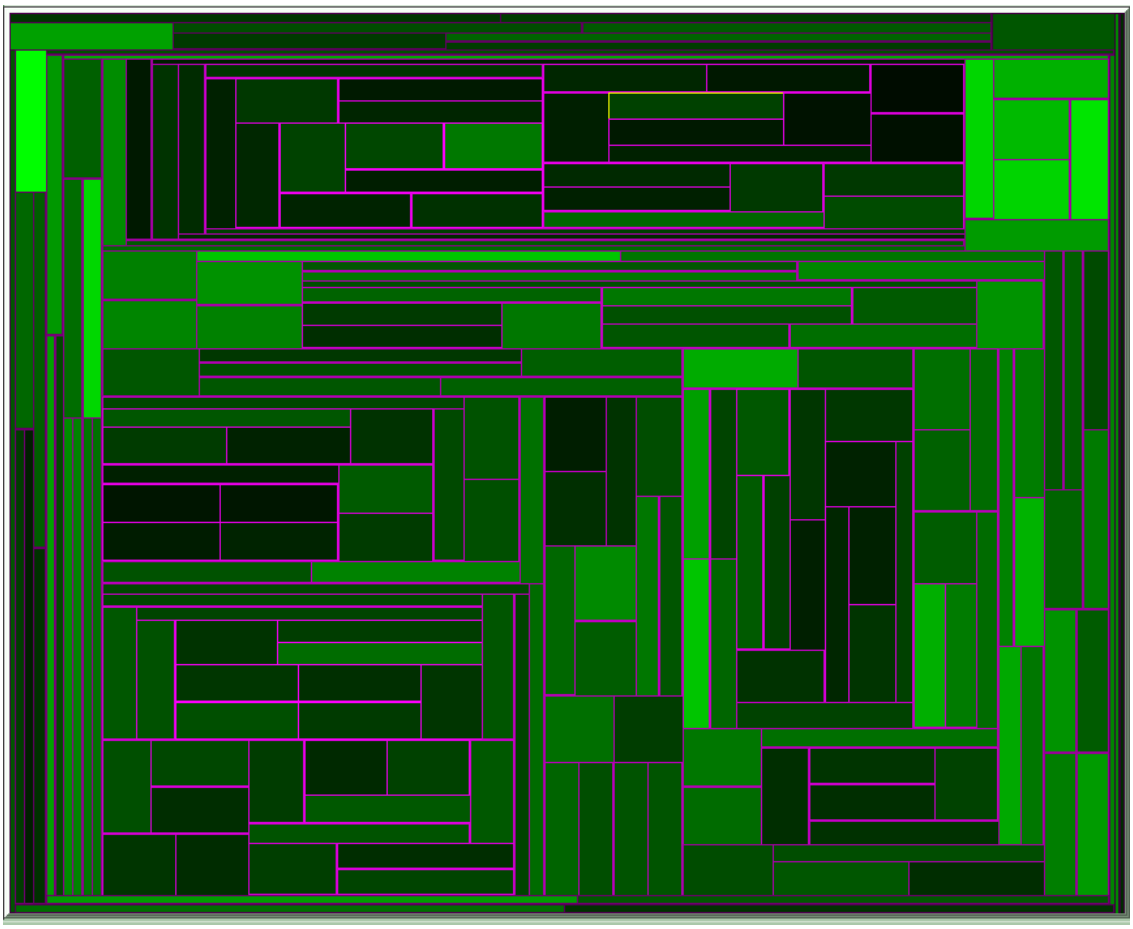


Figure 6.3.2.1: A treemap with border 0 (deactivated)

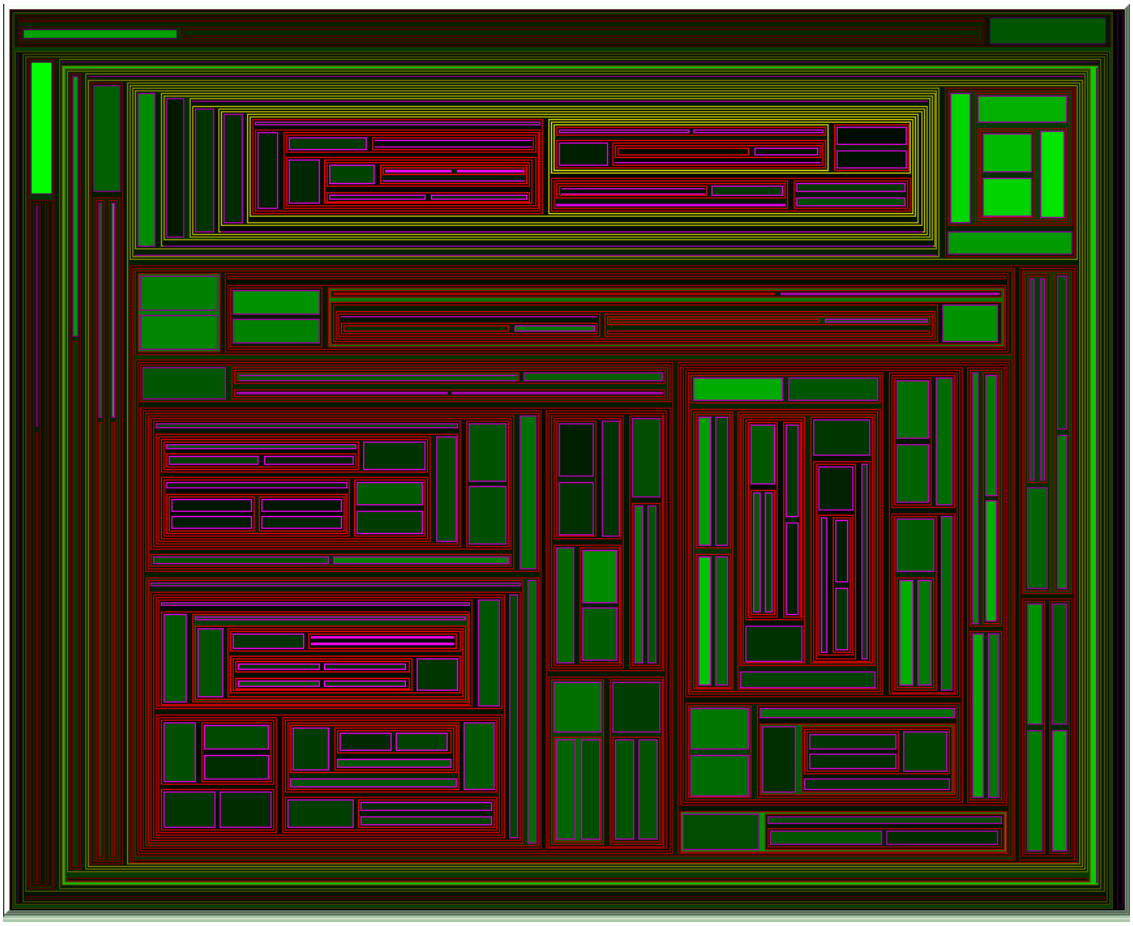


Figure 6.3.2.2: The treemap from figure 6.3.2.1 with border 2, internal nodes are shown but many leaves and even some internal nodes cannot be shown

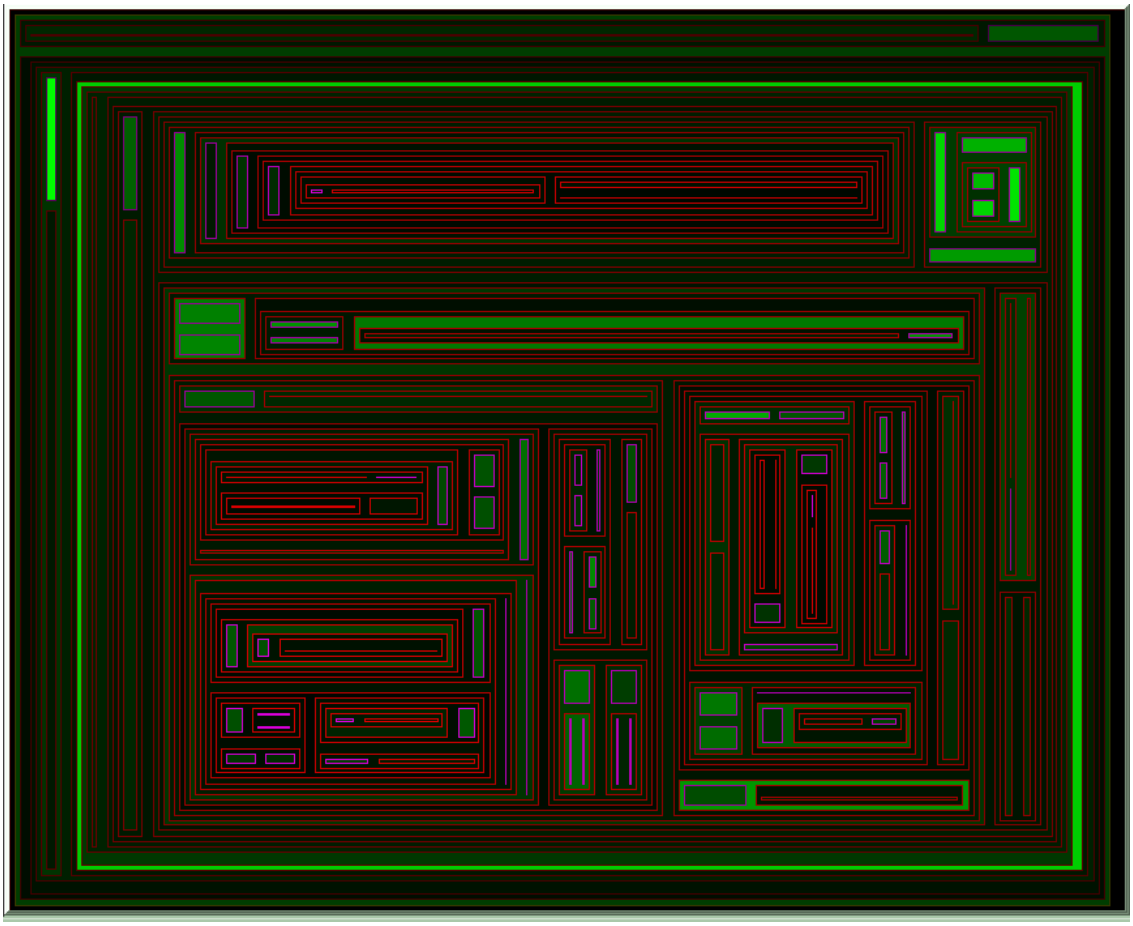


Figure 6.3.2.3: The treemap from figure 6.3.2.1 with border 4, internal nodes are shown even better but many leaves and internal nodes cannot be displayed

The figures above show clearly that a greater border size gives a better perspective of depth making the treemap more pyramid shaped. There is a drawback however which concerns large tree files, in that case the allocation of 1 pixel to a border around every internal node consumes space which cannot be given later on to the leaves, so the leaves are practically invisible since there is no more space left to display them. The figures below show a tree with 10000 leaves and border set to 0 and then set to 1, note that magenta rectangles can hardly be seen:

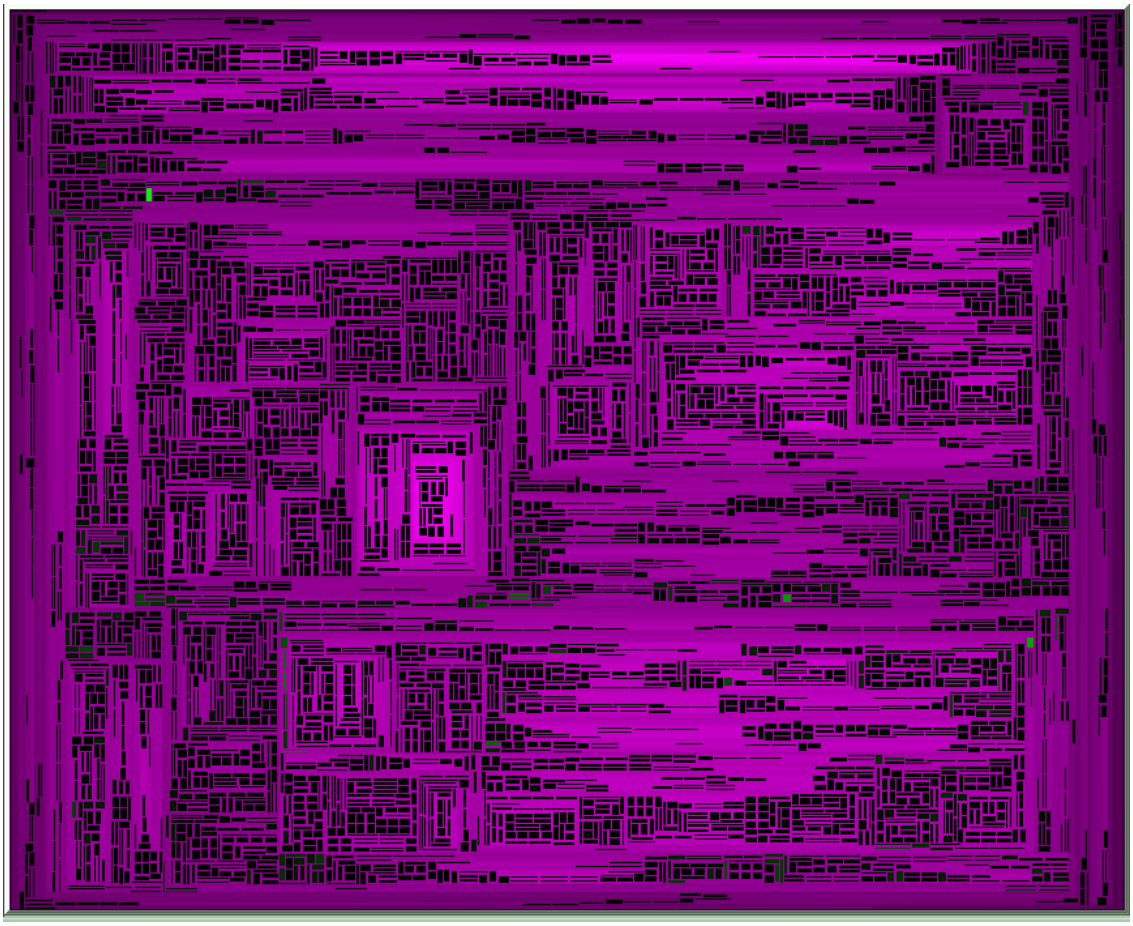


Figure 6.3.2.4: A treemap with 10000 leaves and border set to 0

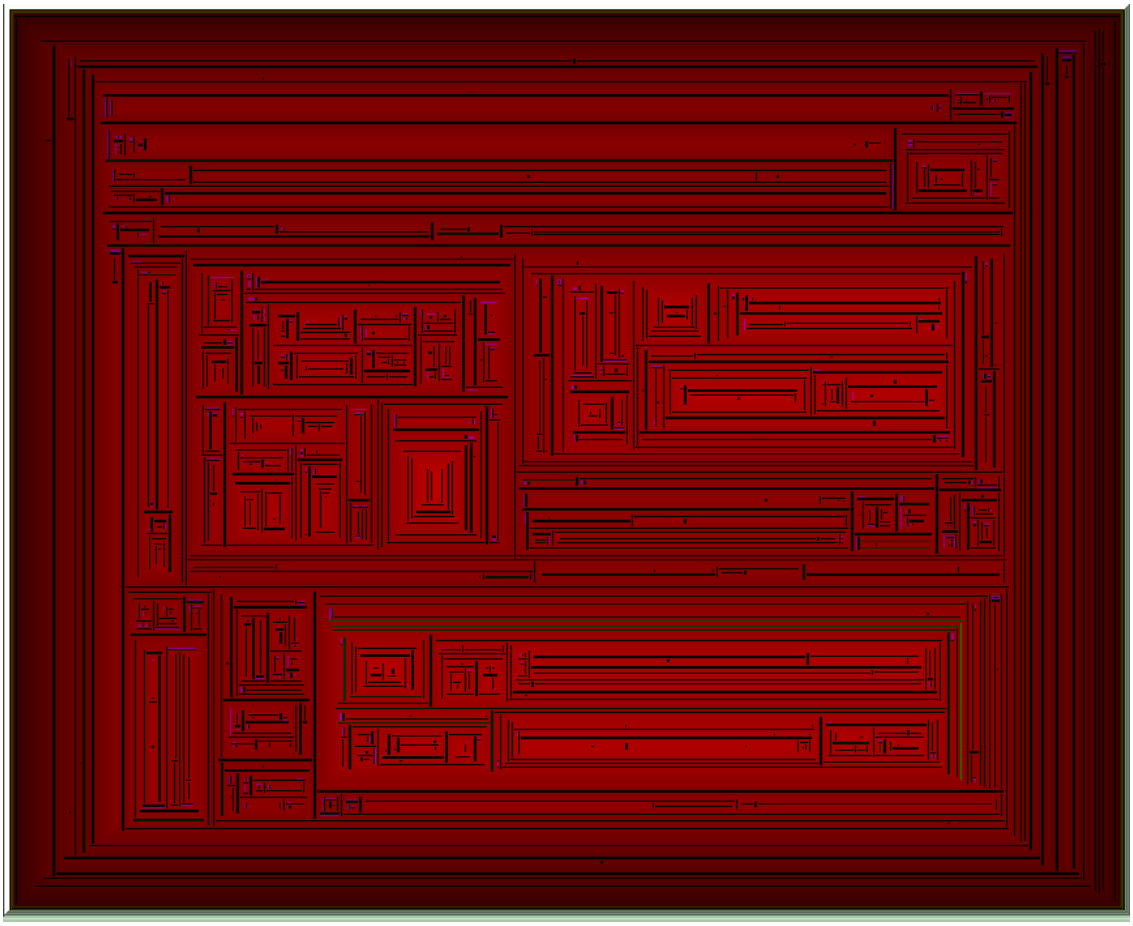


Figure 6.3.2.5: The treemap from figure 6.3.2.4 with border set to 1, hardly any leaf can be seen

6.3.3. Influence of the weight of a node on its size

The treemap algorithms described so far consider the size of a node to depend solely on the number of leaves of the subtree defined by this node. Specifically, since *l* leaves must fill the entire screen each is given an area of *screen-area/l*. The easiest way for doing this is to give each leaf a size of 1 and each parent the sum of the sizes of its children. This way the size denotes the number of leaves contained in the subtree originating from this node and every leaf's area will be exactly the same.

But the user has the option to change this: Instead of giving a size of 1 to each leaf the software may assign the weight associated with each node as its size. Note that for the sake of keeping the integrity of the tree the size of the internal nodes still remains the sum

of the sizes of its children. So while every leaf's size is equal to its weight this does not hold for internal nodes.

The implementation has a slider for the user to adjust the balance between using "size = 1" and "size = weight". Below are some figures with the percentage set to 0%, 50% and 100%:

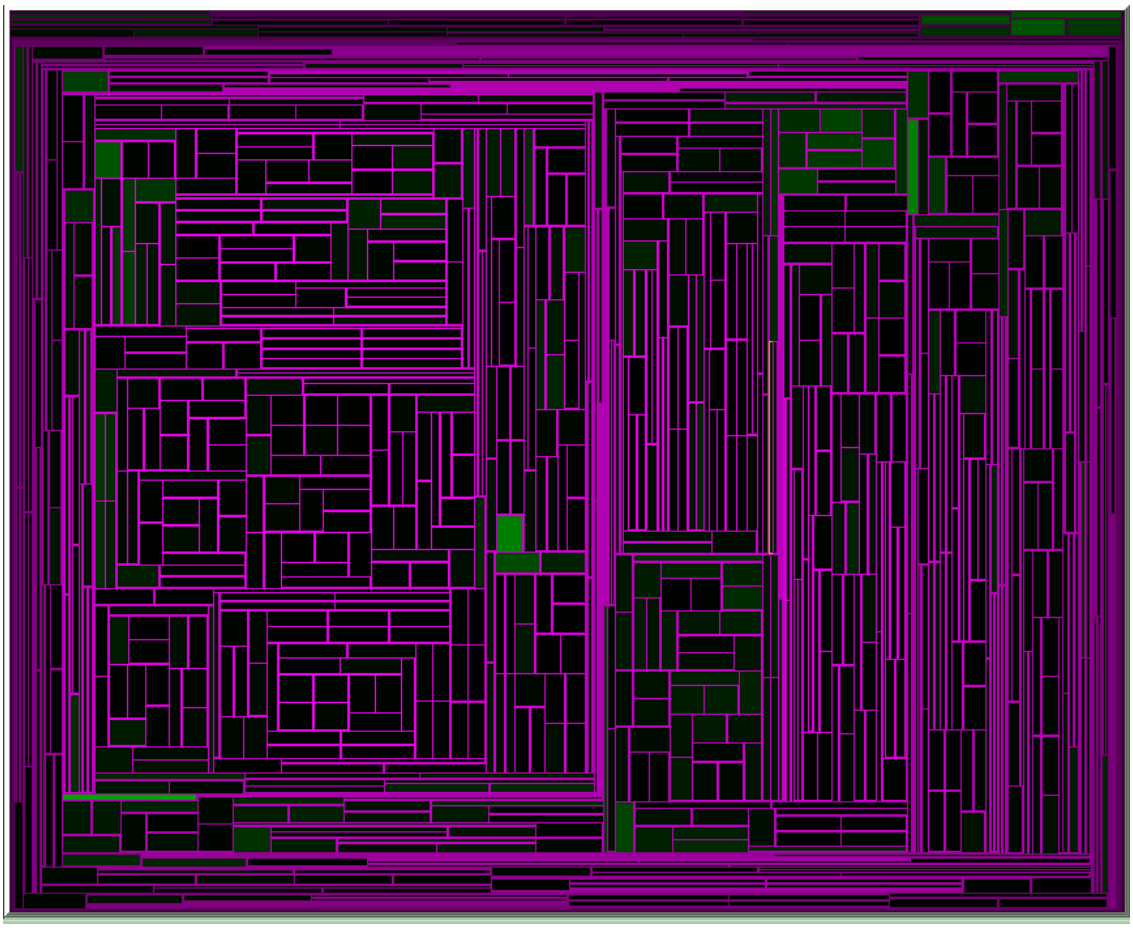


Figure 6.3.3.1: A treemap with 1000 leaves and all leaves sizes set to 1

Figure 6.3.3.2: A treemap with 1000 leaves and all leaves sizes set to 50% of 1 and 50% of their normalized weight

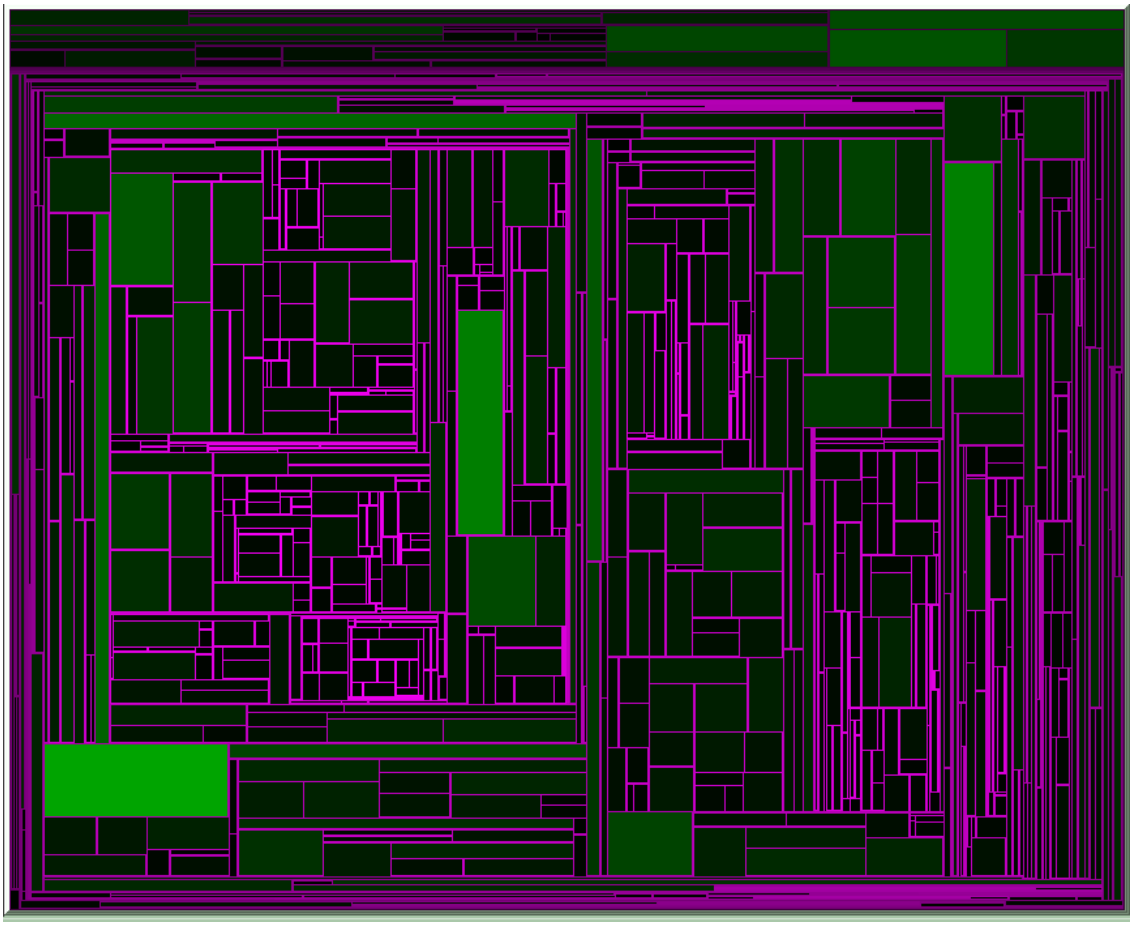


Figure 6.3.3.3: A treemap with 1000 leaves and all leaves sizes set to their normalized weight, the brightest nodes are also the biggest

This function is especially useful in animating treemaps which display clustered microarray data as will be shown later. This is because the spotting of a series of intense gene expressions is a useful tool in the specific case.

6.3.4. Addition of emphasis to thin nodes

In order to make the thin nodes -which when placed alongside much bigger siblings disappear- visible there is an option to make them thicker according to the user's preference. The software detects nodes which have a width (or height) smaller than a threshold (including borders this threshold is 4 pixels) and marks them as "thin". These thin nodes benefit from this feature by getting "free pixels" added to their thin dimension which are taken from their biggest sibling.

The following figure shows a phylogenetic tree with 500 leaves, meaning there is plenty of space for each node to be visible, still the node with the max value (and consequently the brightest green) cannot be seen:

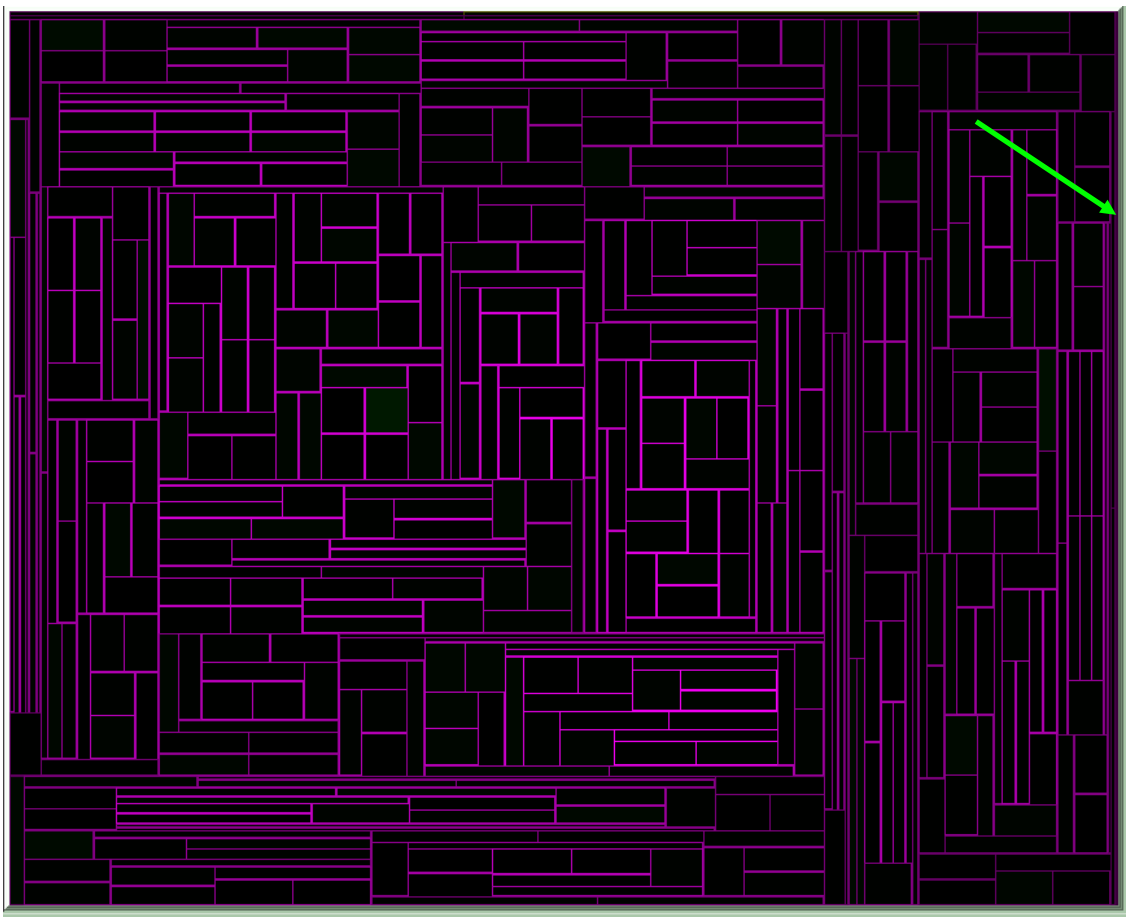


Figure 6.3.4.1: A treemap with 500 leaves

In reality the highest-valued node is a thin vertical strip on the right of the display panel where the green arrow point to. But this highest-valued node is a leaf at depth 1 and has only one sibling which obviously contains the 499 remaining leaves, so 499/500 = 99,8% space goes to the sibling and only 0,02% to the highest-valued node, moreover this 0,02% is given as a thin strip rather than as a square.

The user has the option to add up to 8 pixels to those thin nodes, as can be shown in the below figure:

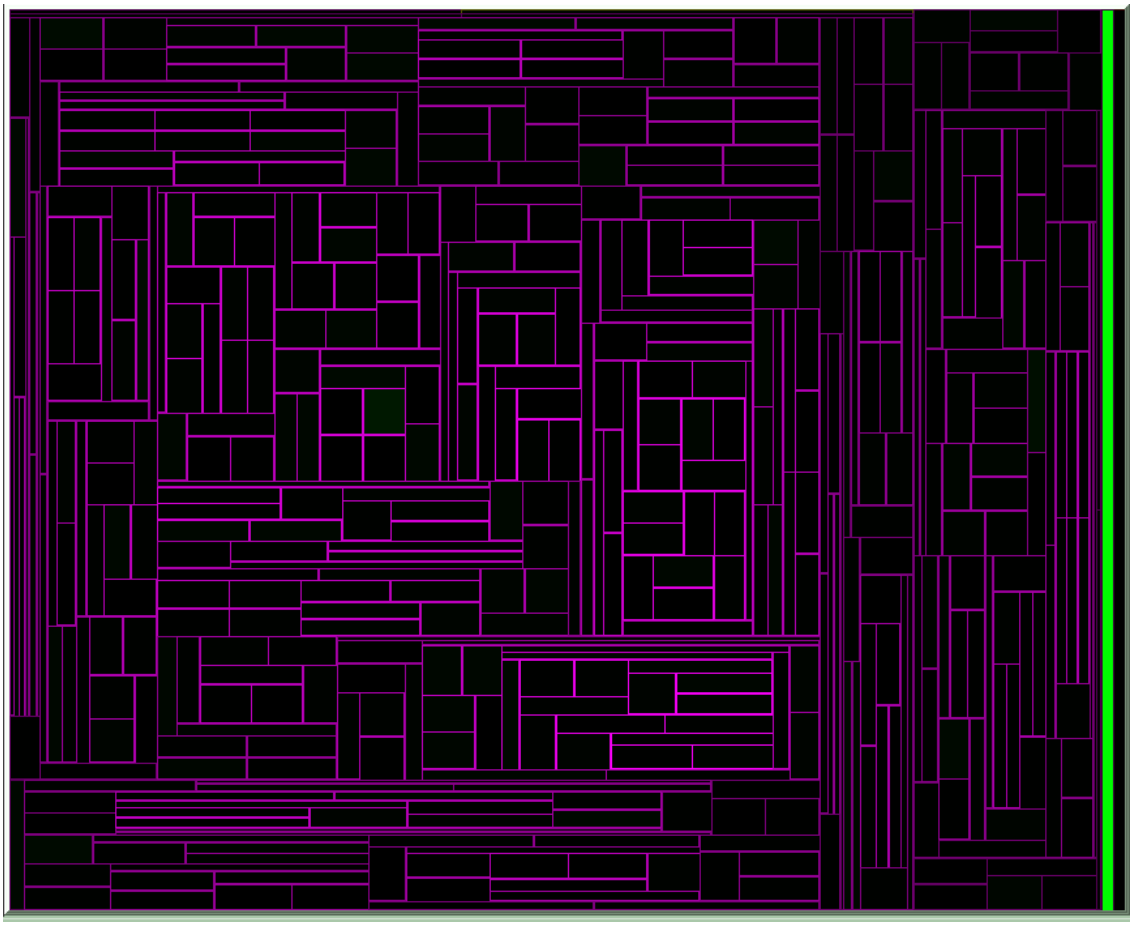


Figure 6.3.4.2: The treemap from figure 6.3.4.1 with added pixels to the thin nodes

In large trees space is very valuable and the percentage of nodes characterized as "thin" reaches 95% of the total node count, so the use of this feature is not going to do any good by taking pixels from an already thin node in order to add to another thin node. It is rather intended as a useful aid for small trees or zoomed portions of large trees as will be explained later on.

6.3.5. Addition of cushions

This is a feature based on [17] where the flat coloring of each leaf is replaced by an illusionary colored "cushion" which gives the treemap a further impression of height. Note that the cushions implemented here are simpler compared to the work of [17]. The reason for this is that the calculations necessary to produce accurate cushions in [17] are

very time-consuming and render the software very slow even for small datasets. The user will notice a decrease in speed even for the cushions implemented here.

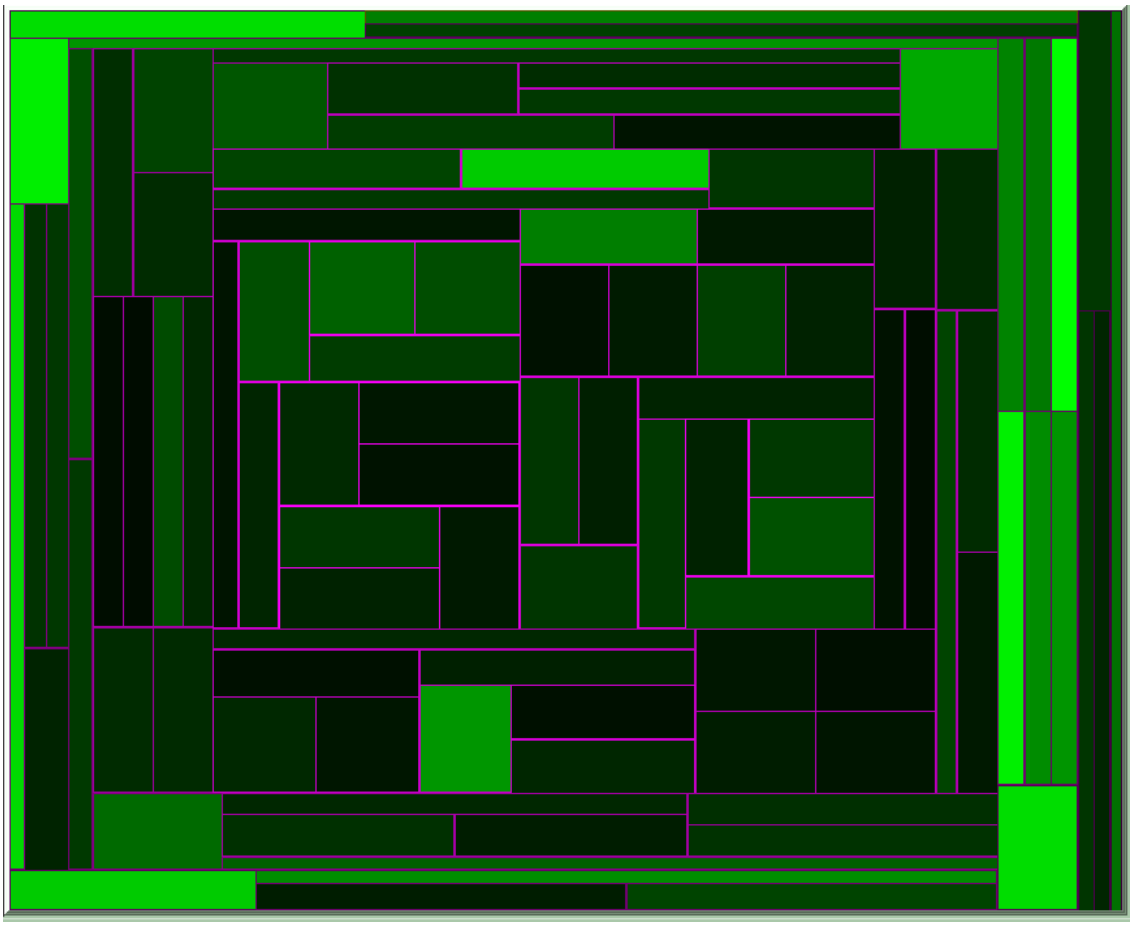


Figure 6.3.5.1: A treemap with 101 leaves and flat display

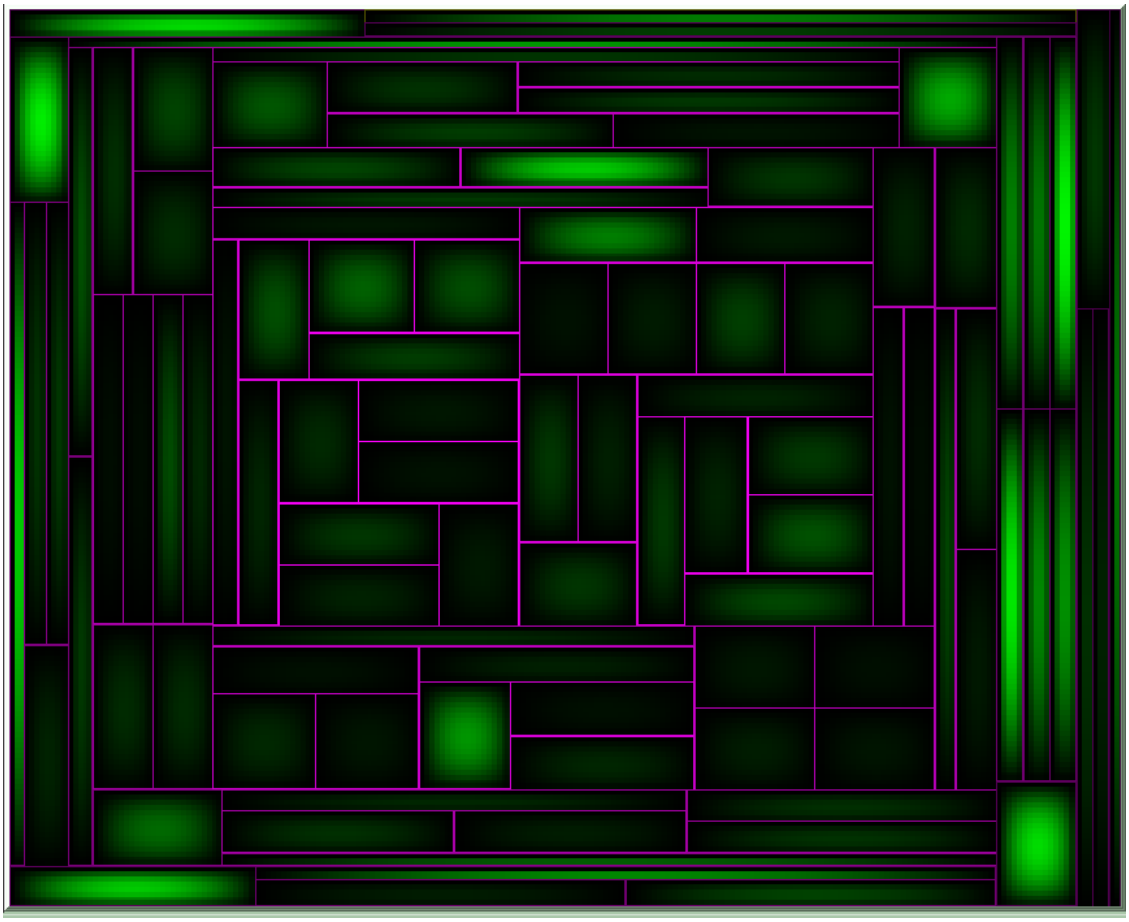


Figure 6.3.5.2: The treemap from 4.3.6.1 with cushion display

6.3.6. Cut-off of levels of nodes

This is a feature which enables the user to display only nodes from a certain depth and below towards the root. So the clogging of many small internal nodes and leaf nodes can be cleared up for the sake of the big picture. This has been attained by placing a "mark" at the offset-level for the cut-off and instructing the treemap algorithm to ignore all levels above the offset.

Below we have a figure of a tree with 1000 leaves:

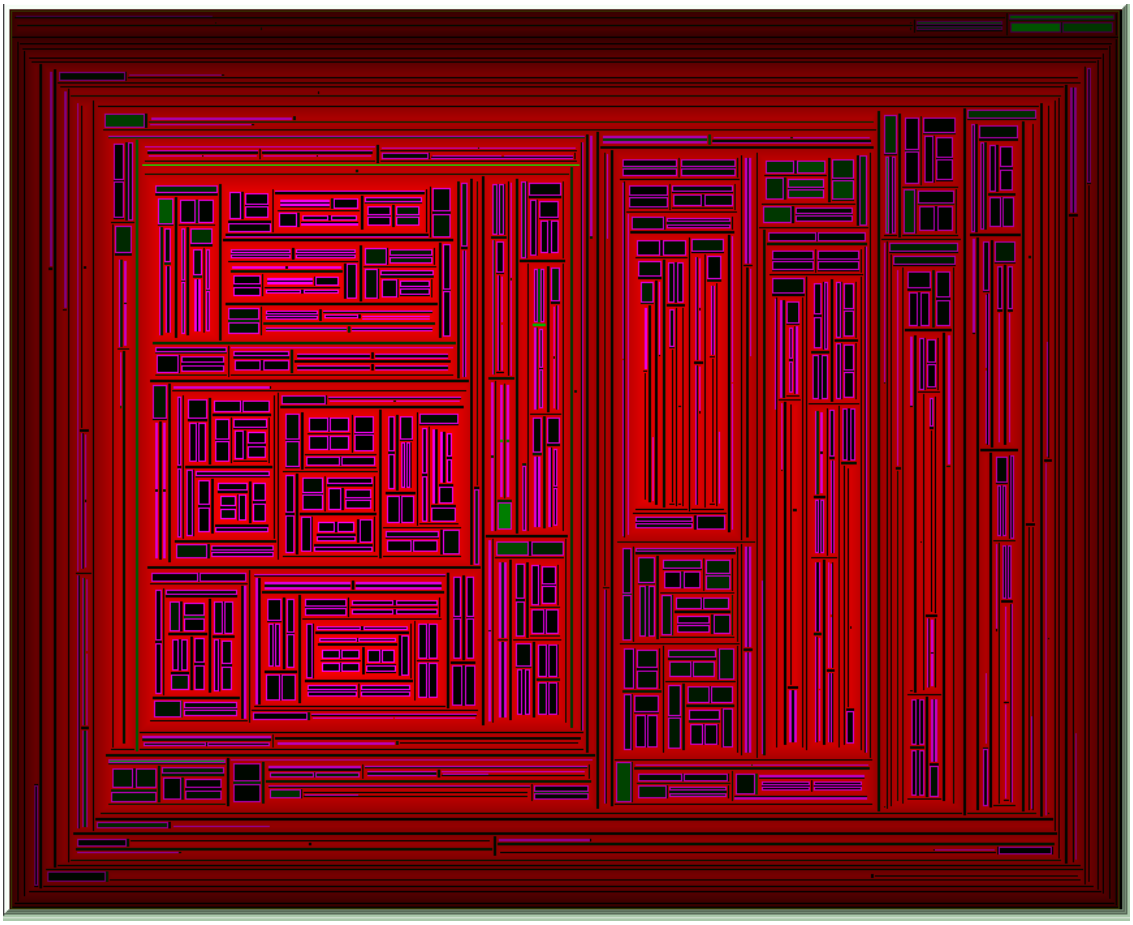


Figure 6.3.6.1: A treemap with 1000 leaves (the max depth of the tree is 63)

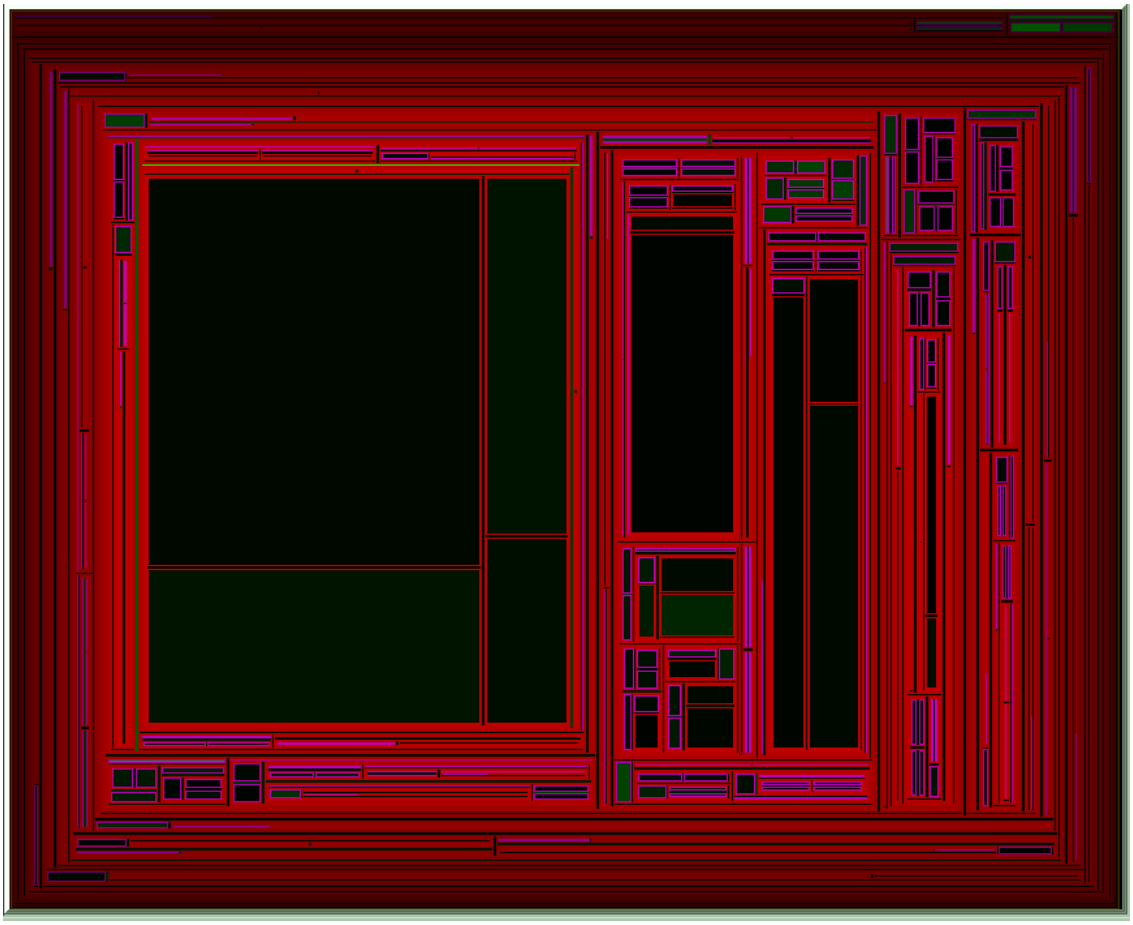


Figure 6.3.6.2: The treemap from figure 6.3.6.1 with all nodes above level 46 cut off

6.3.7. Display of part of the treemap as a standard tree

This feature gives the user the option to display a portion of the tree from the root and upwards as a standard tree. As can be seen in previous figures of treemaps the first levels of the tree are practically rectangles around the treemap which are *border value* pixels thick. Below is a figure of a treemap with 1000 leaves and below it is the same treemap with its first 40 levels displayed as a tree:

Figure 6.3.7.1: A treemap with 1000 leaves

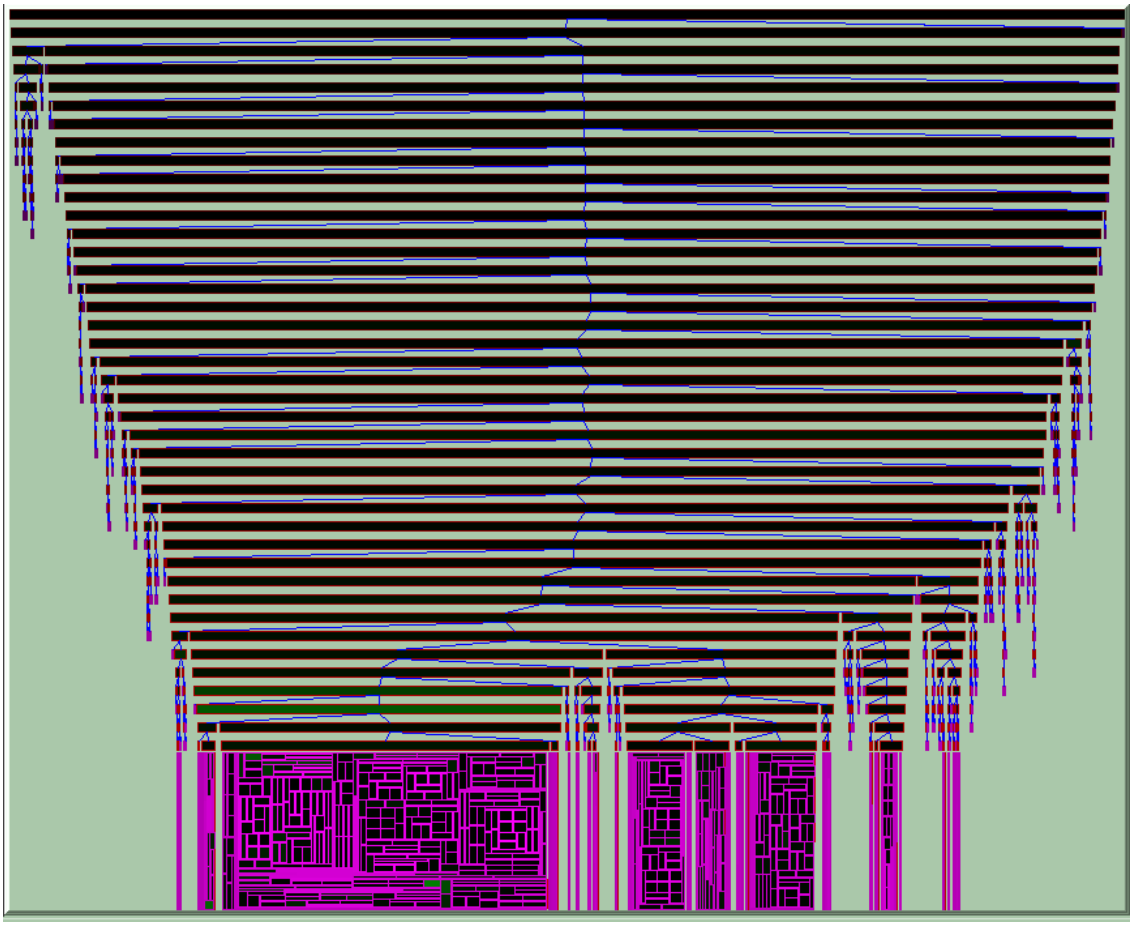


Figure 6.3.7.2: The treemap from figure 6.3.7.1 with its first 40 levels displayed as a tree

The software keeps internally a fixed *node-height* and *distance-between-nodes* for nodes displayed as a standard tree, then it calculates how many nodes fit on the height of the display-screen by finding the maximum *number* of nodes which can be displayed as a standard tree and which leave enough height so that the rest of the tree can be displayed satisfactorily as a treemap. The user may select a number between 0 (deactivating the tree-view) and the *number* described above to be the number of levels displayed as a tree. Then the software during its display algorithm checks the level of the current node being processed and decides whether to display it as a standard tree node or a treemap. Note that the width of every standard tree node reflects the number of leaves which emanate from it for a better organizational view.

6.3.8. Export to a newick file

The user has the option to export the data being displayed on screen to a newick file on disk for storage and retrieval. This option is enabled whenever the data shown on screen is a treemap, namely only if the data on screen is a standard tree, a newick file or an animating treemap.

The first two cases present some interest since the treemap which is exported may be structurally different than the one originally loaded, meaning that the user can load a file, change its root and then export the altered file. Or the user may load a file, zoom into a node and then export it, hence saving a smaller subtree of the original tree on a new file.

In the case of animating treemaps the software saves *num of genes + num of patients* different files, one for each frame, to the directory specified by the user. The user types a name into a save dialog and the program creates files named *nameGene 1.nw, nameGene 2.nw, etc.*

6.4. Features exclusively for microarrays

6.4.1. Filtering of a microarray

Microarrays are initially displayed as matrices of *g* genes by *p* patients which are quite big, but if the microarray contains a classification on the first line, the user is given the option of applying a filtering algorithm [23] in order to throw away a number of genes which contribute the least to the microarray. This is a significant way to reduce the size of the microarray to be displayed because with microarrays with more than 200 genes the software has difficulties in displaying them fast enough and the clustering time becomes quite long.

6.4.2. Location of a specific gene expression in the other treemap

Another useful option the user has is to locate a specific gene expression in the other treemap during the treemap animation. Since researchers are interested in locating *areas* of the clustered microarray with similar gene expressions, just looking at the siblings of a node displays the neighboring gene expressions on the same column or row as the targeted gene expression.

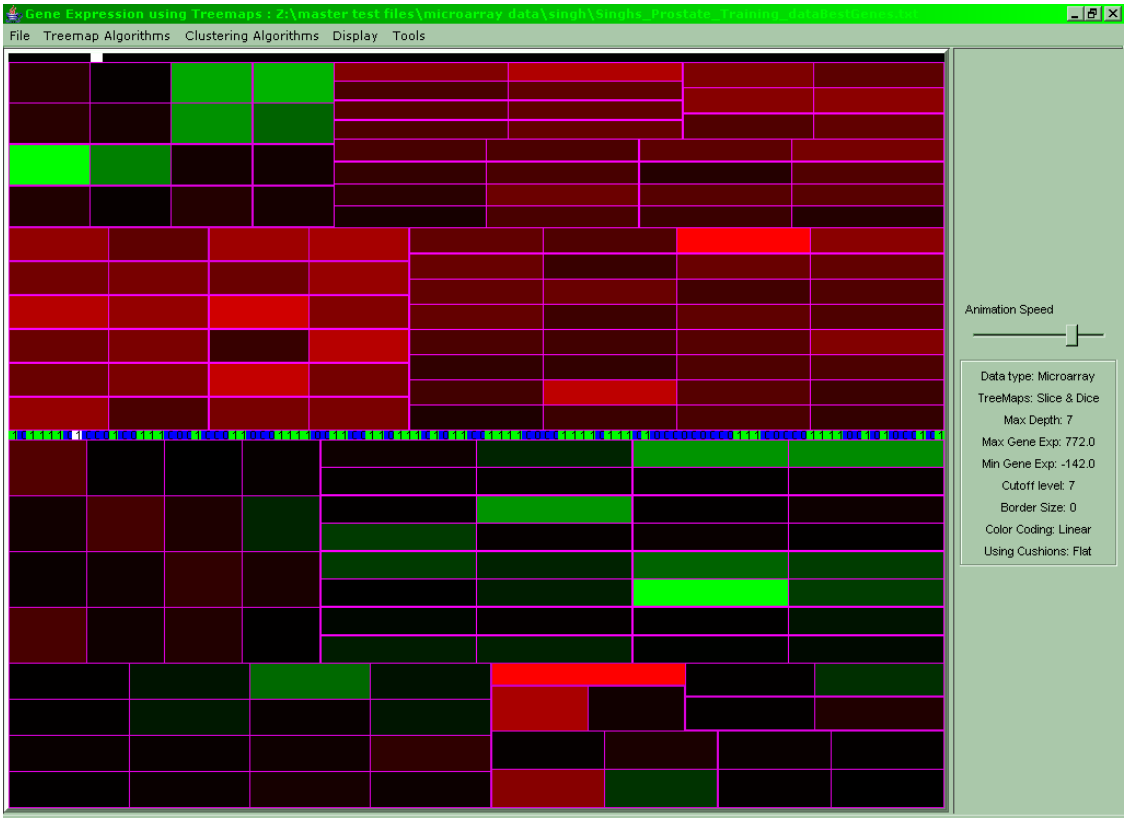


Figure 6.4.2.1: Each frame of the treemap animation corresponds to one row (and column) of the microarray, so locating areas can become difficult

The user may, after pausing the animation, use this function to locate the same gene expression on the other treemap whose siblings are on the same row or column (respectively to the first treemap).

Figure 6.4.2.2: The gene expression selected in one of the treemaps has just been located in the other

6.4.3. Display of the classification instead of the gene expressions

If a microarray file is loaded which contains a classification line, the user may choose to display the genes of every patient color-coded according to whether they belong to the one or the other classification. This enables the user to make visual estimations about the accordance of the classification to the clustering.

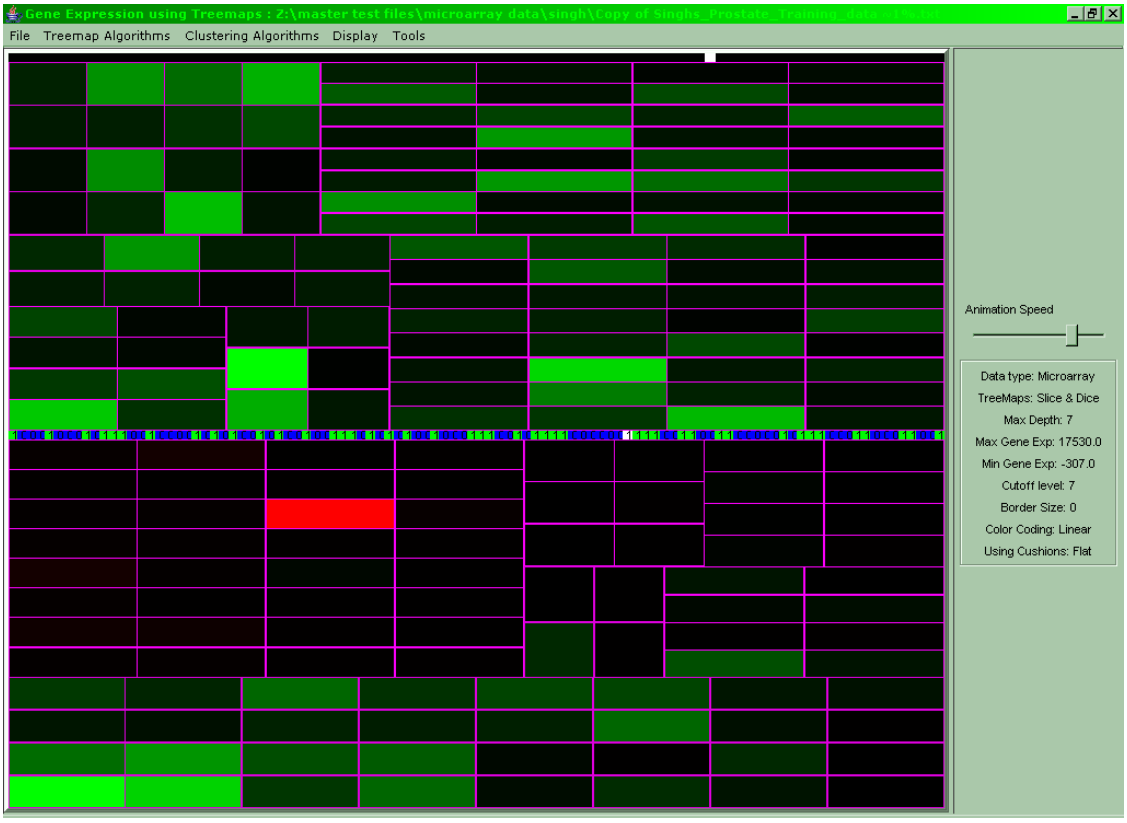


Figure 6.4.3.1: A double clustered animating treemap

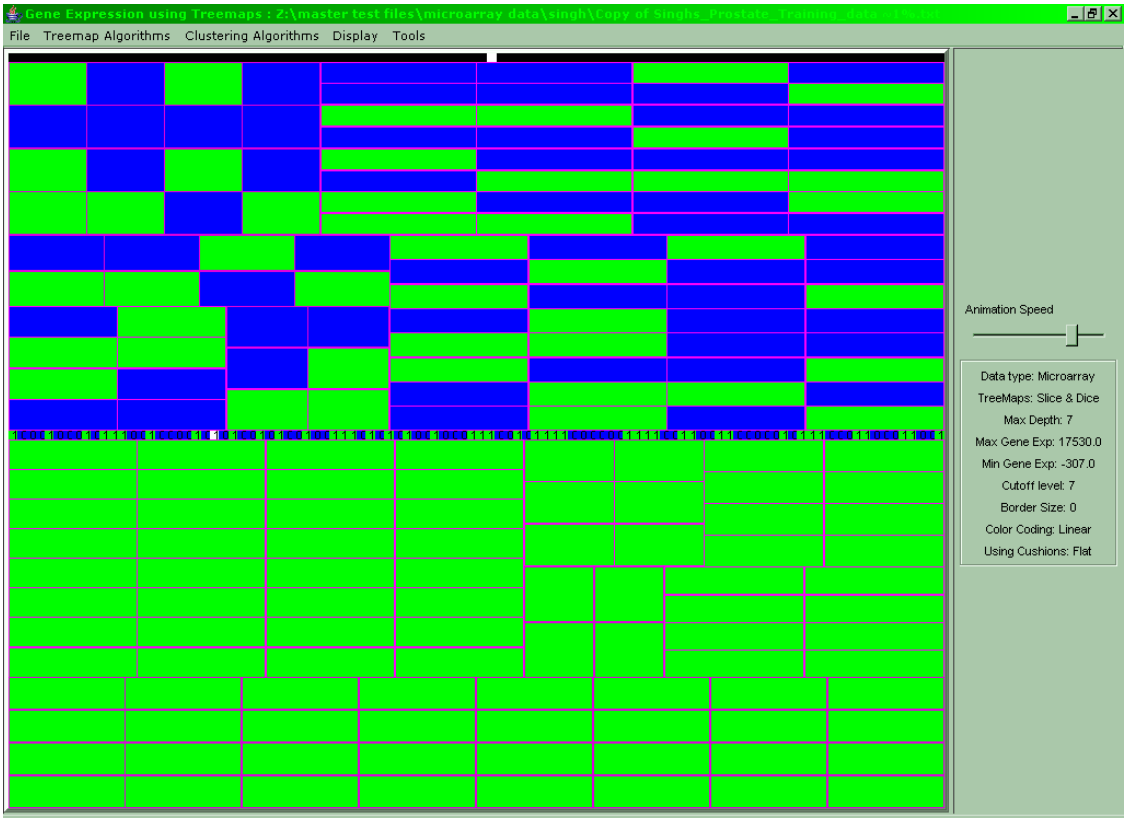


Figure 6.4.3.2: The treemap from figure 6.4.3.1 with classification color-coding instead of gene expression

Note that while the leaves of the above treemap are differently classified, the bottom treemap frames belong either completely to classification 0 or classification 1. The explanation for this is visible in the classic clustered microarray visualization which can be seen below in figure 6.4.3.3:

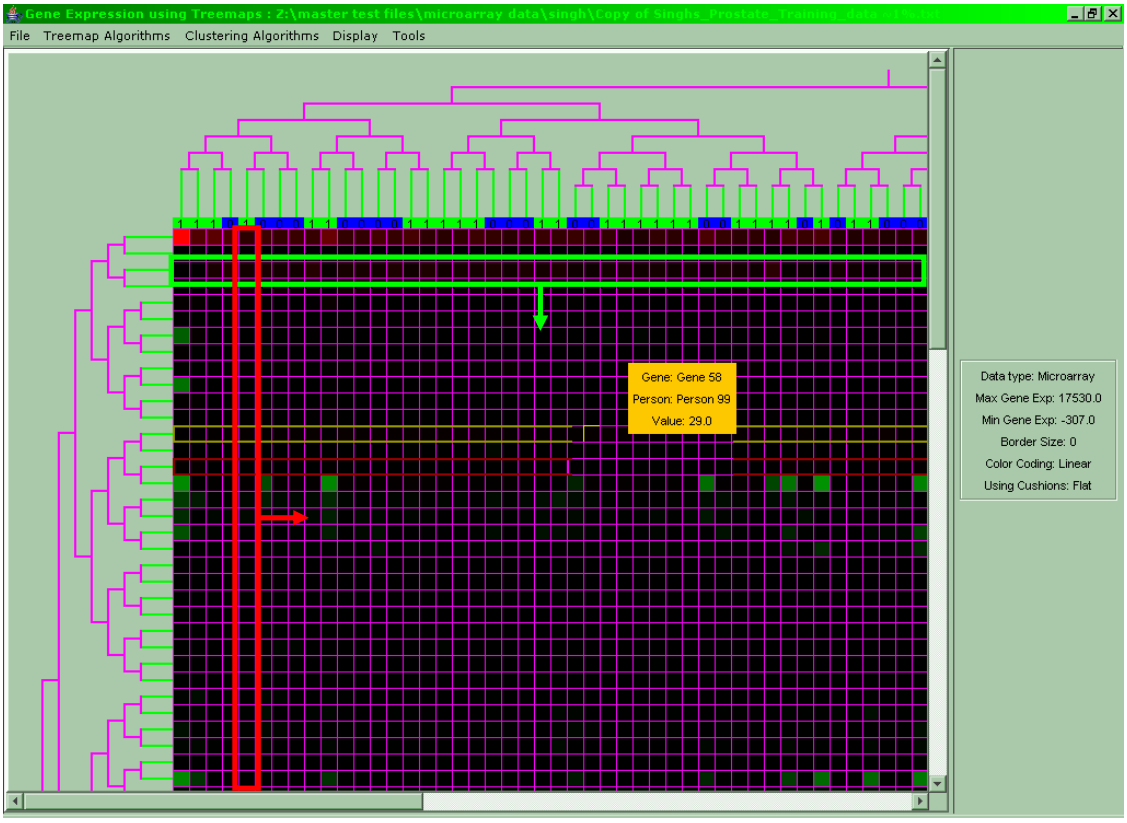


Figure 6.4.3.3: The progressing frames when seen in context with the classification show that the top treemap has leaves of varying classification per frame while the bottom treemap has just one classification per frame

6.5. Features exclusively for phylogenetic and standard trees

6.5.1. Change in the tree center

For phylogenetic files where the center of the tree is only formally defined, changing the tree center is an interesting feature. The user may select a node to be the new root of the tree and so to redesign the tree completely:

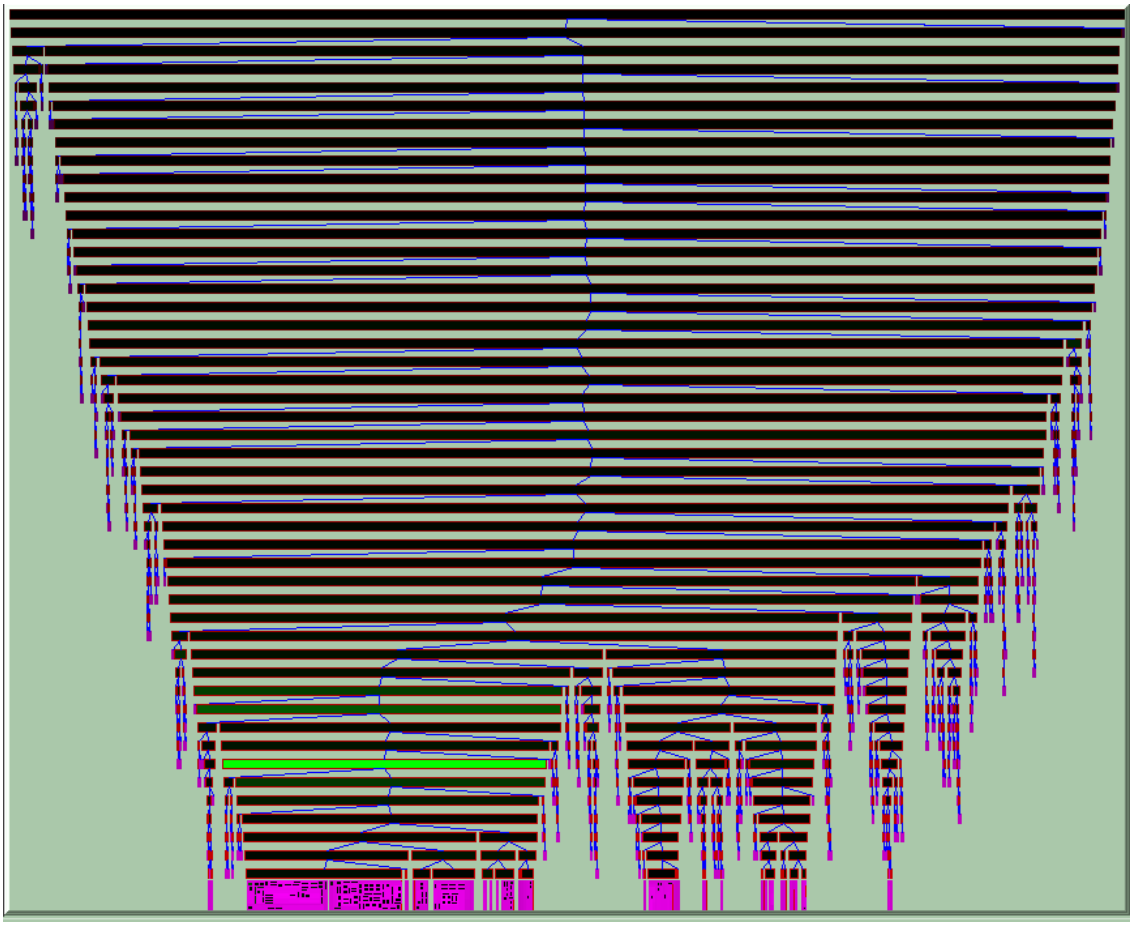


Figure 6.5.1.1: A phylogenetic treemap with 1000 leaves displayed in maximum tree-view, its maximum depth is 63 and the tree-view reached until level 47

By selecting the bright green node at the lower left of the display and making it the current root the following tree occurs:

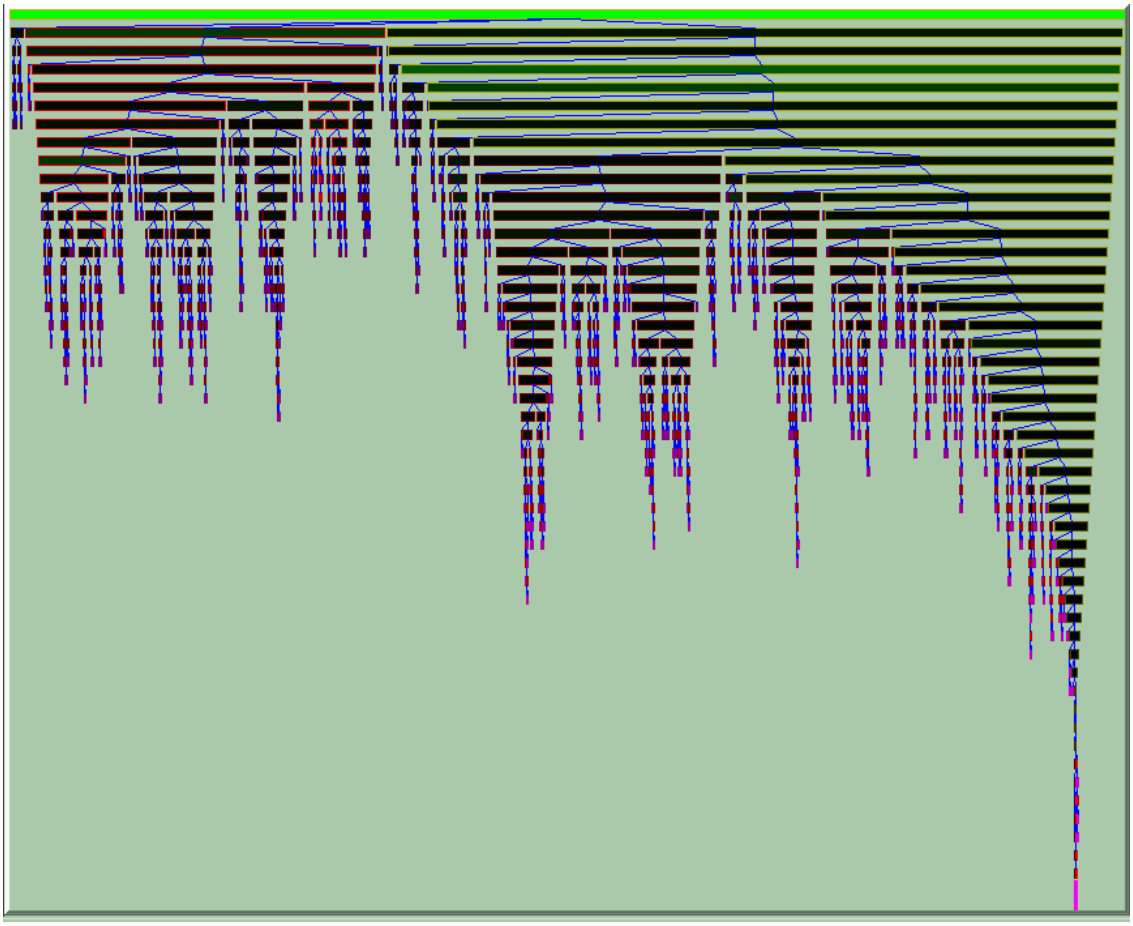


Figure 6.5.1.2: The tree from figure 6.5.1.1 after a new root has been chosen, its maximum depth is currently 51

The process of selecting a new root is quite easy for unrooted trees, but for rooted ones (as almost all are in data structures) the process is a recursive one with progressive swapping of children with parents etc. so as not to lose a connection between a node and its parent and children.

There is also the feature of applying the algorithm for finding the tree center and setting the center of the tree as its root. This algorithm works by taking the given tree and performing a full tree traversal during which every leaf is cut off from the tree. For example we have the following tree:

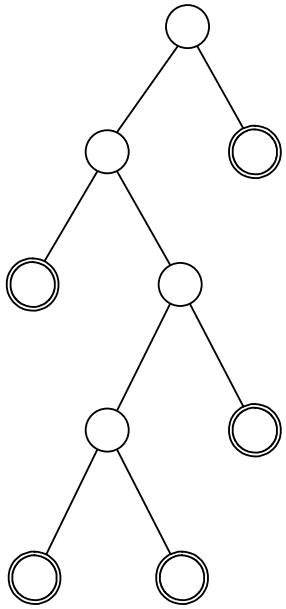


Figure 6.5.1.3: A tree. Leaves are marked with a double border

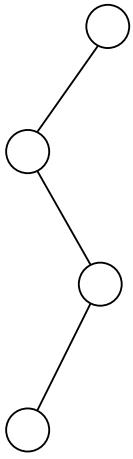


Figure 6.5.1.4: The tree from the previous figure after all leaves have been removed

After that another traversal is performed during which all nodes with degree one are marked as leaves, note that even the root may be marked and later deleted through this process.

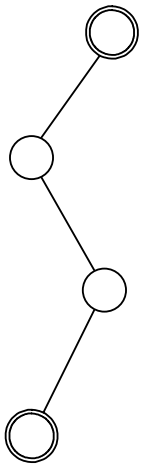


Figure 6.5.1.5: The tree has been traversed and all nodes with only one edge are marked as leaves

This process continues recursively producing continuously smaller tree. When the total number of nodes becomes three or less the center of the tree becomes obvious. This node is marked as the center of the tree and the tree is loaded again with this node as its root.

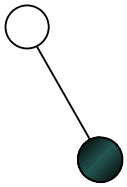


Figure 6.5.1.6: When three or less nodes remain the center becomes obvious

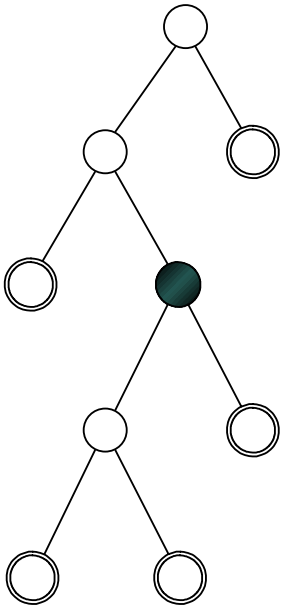


Figure 6.5.1.7: The original tree with the center marked

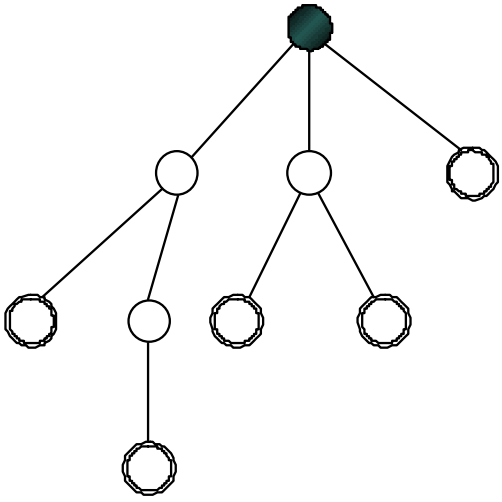


Figure 6.5.1.8: The tree after being redesigned. The maximum depth has become minimum

This produces results seen best using the maximum tree-view to show the obvious decrease in maximum depth of the tree:

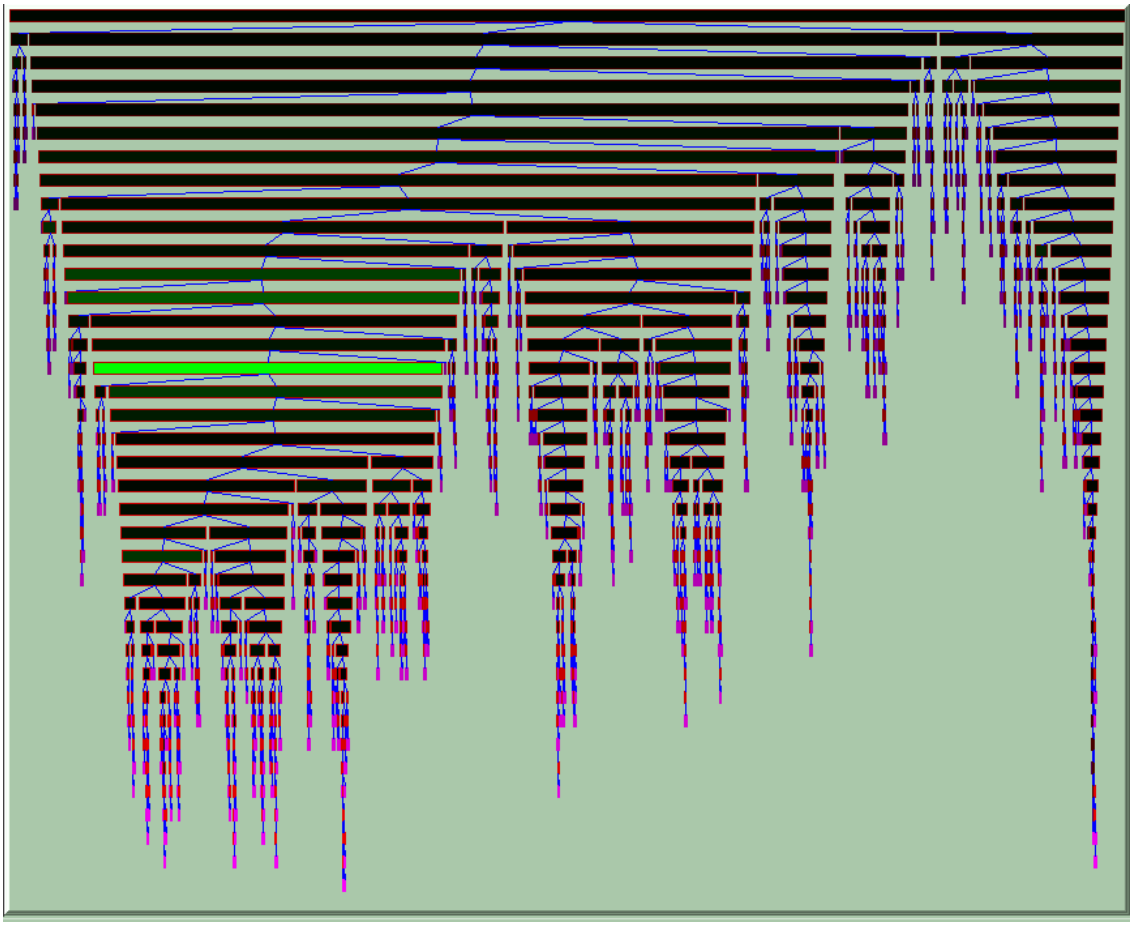


Figure 6.5.1.9: The tree from figure 6.5.1.1 after the tree-center algorithm has been applied, the maximum depth has been reduced to 37

Note that by automatically changing the root the names of the nodes do not change, the old root will still be called “Internal 0:0:0” but it will no longer be at the top of the display.

6.5.2. Zoom into a node

Another feature is that of zooming into a node. In fact it is just a recentering of the entire tree with the selected node being made the root while all nodes towards the old root are not displayed as well as the new roots siblings. This is easy to implement, it just requires storing the old root somewhere in order to keep track of the entire tree in case the user wants to return. Other than that it is just the replacement of the root object with the selected node.

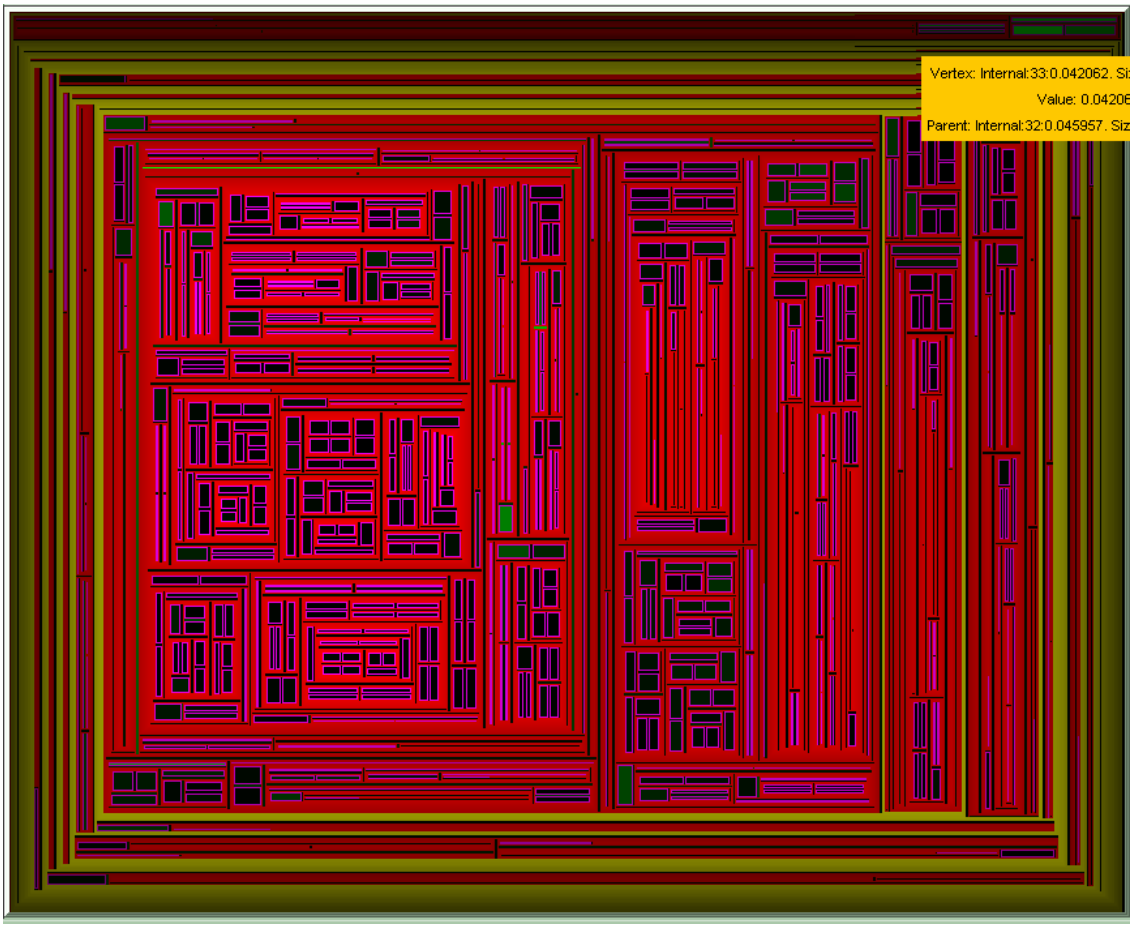


Figure 6.5.2.1: A phylogenetic treemap with 1000 leaves, the selected internal node (smallest / brightest yellow border) will be zoomed into

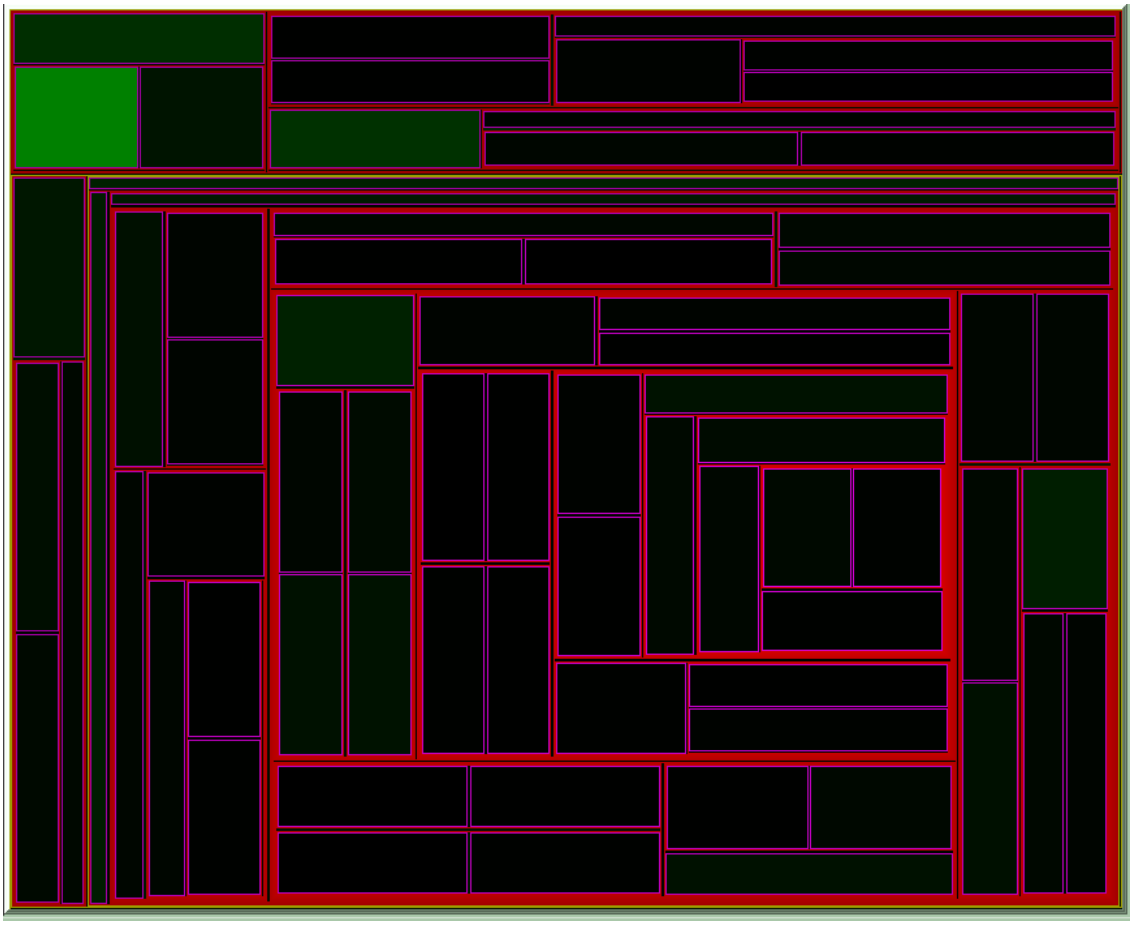


Figure 6.5.2.2: The treemap from figure 6.5.2.1 zoomed into the node previously selected

This function clearly makes local nodes much easier to view. In order to return to the original tree there is a function for going up one level.

6.5.3. Display of a list of leaves originating from a specific node

While displaying phylogenetic or standard trees the user may view the entire set of leaves originating from a specific node (either internal or a leaf node, in the latter case the leaf set contains just this one leaf) along with the total distance from the clicked node to each leaf. The information is displayed in a list below the information panel to the right of the treemap:

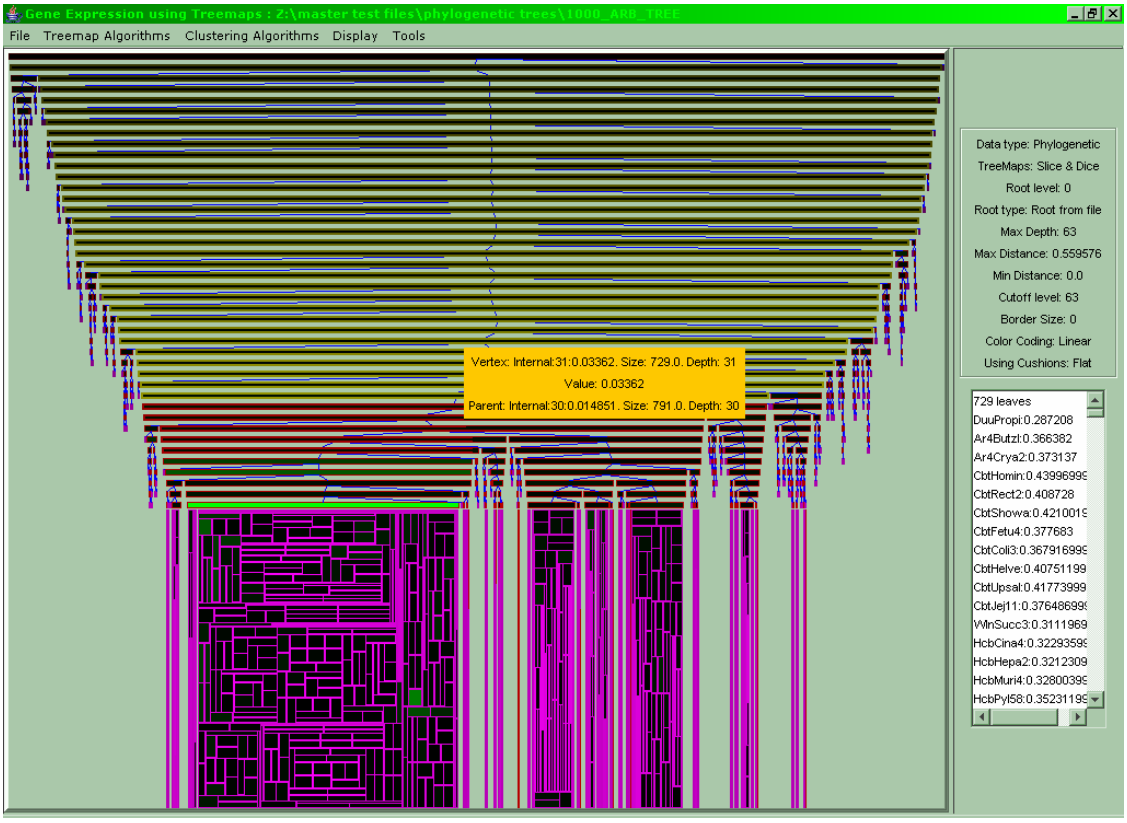


Figure 6.5.3.1: The list on the right of the display shows the leaves originating from the selected node

Furthermore when the user selects a node from the list on the right the selected node is highlighted with a border and a popup window with its info:

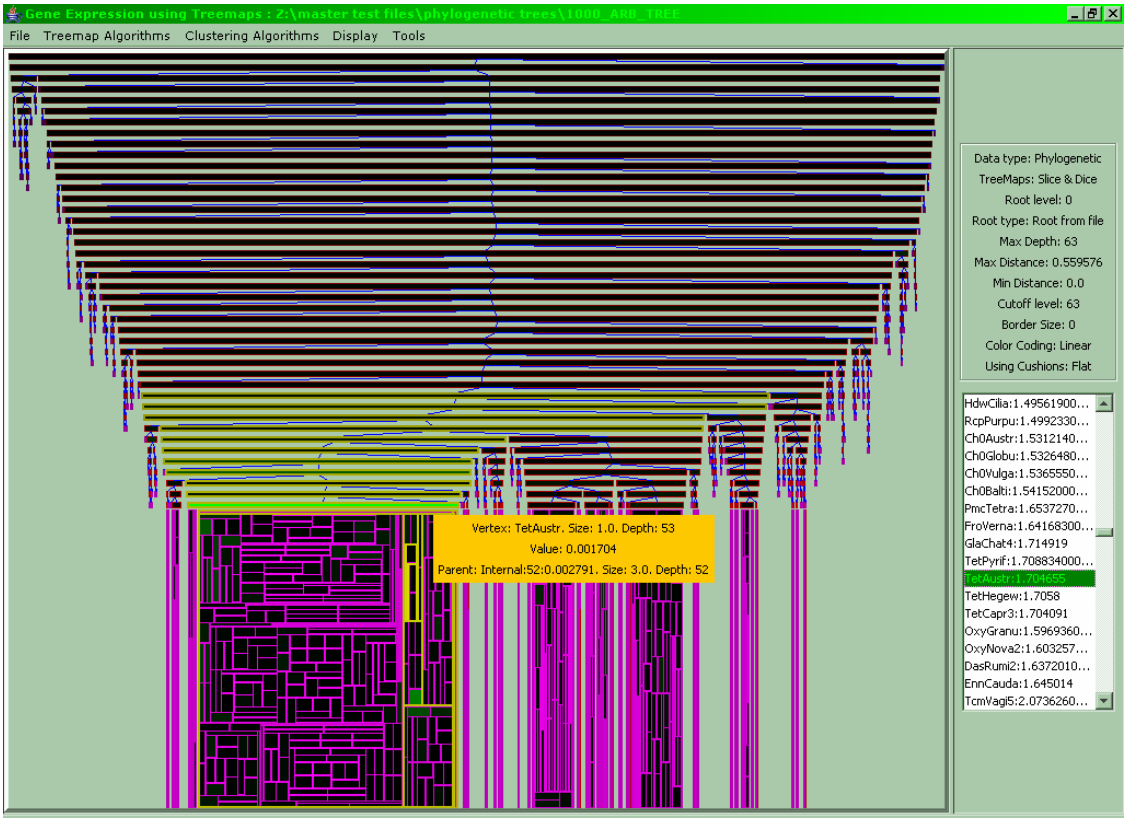


Figure 6.5.3.2: Whenever the user selects a node from the list on the right the corresponding node is located, selected and information about it displayed

6.6. Display of the orders of the species of a phylogenetic tree

This feature is mentioned separately from the others because it applies only to phylogenetic trees which have designated the orders for their leaf nodes by loading an order file.

Whenever the user loads a phylogenetic tree a dialog appears and asks whether an order file should be loaded as well:

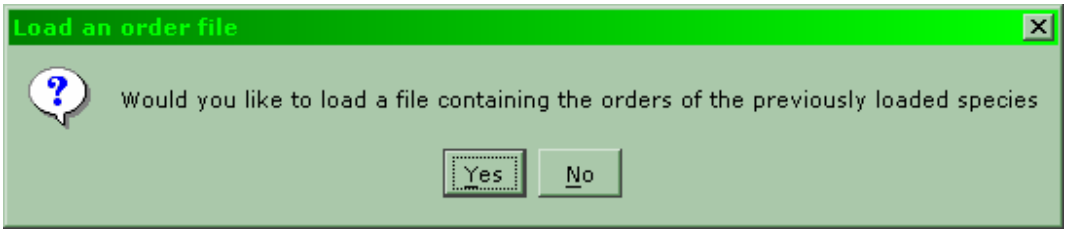


Figure 6.6.1: The dialog asking for the order file

If the order file is valid, meaning that every leaf node in the newick file appears exactly once in the order file, then the phylogenetic file appears normally:

Figure 6.6.2: Initially the orders of the species are not visible, only the popup window shows the order of the currently selected node

The user may choose to have the nodes colored according to either their distance as can be seen in figure 6.6.2 above, or according to their order:

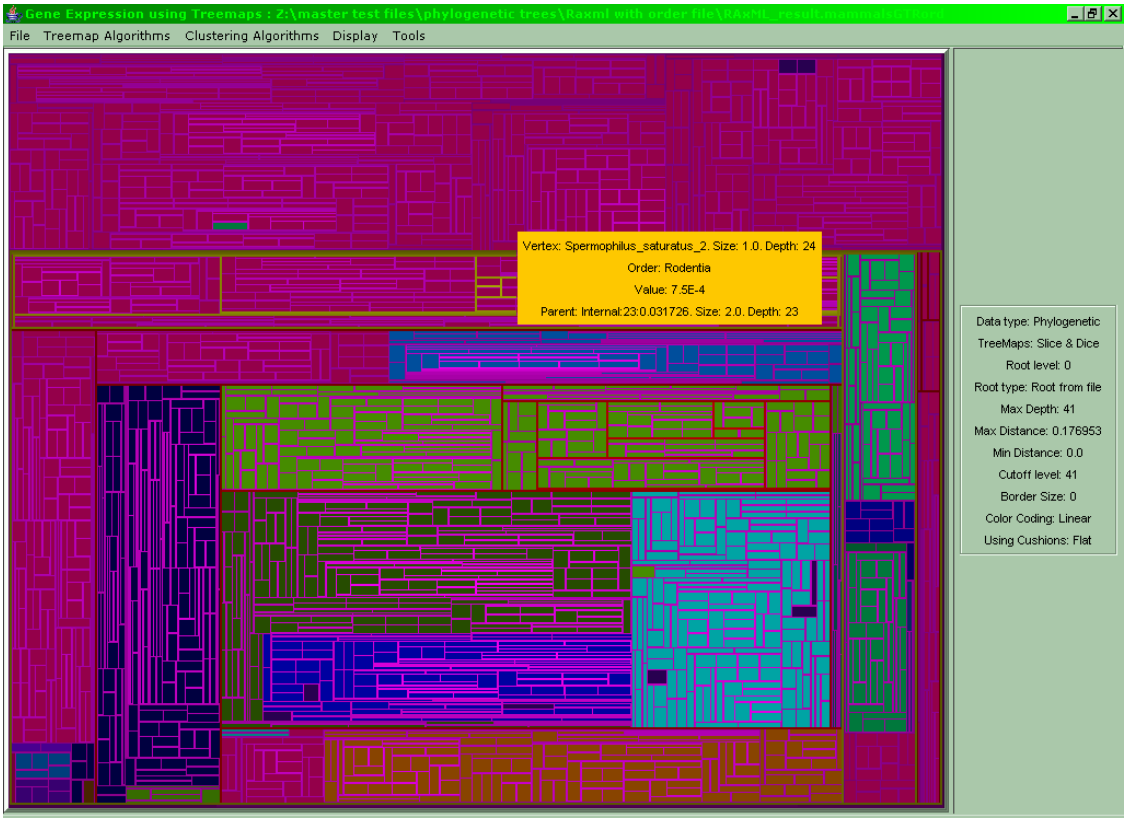


Figure 6.6.3: The color coding has changed to reflect the orders of the species

The user may also combine distance and order for every node where the left half of the node displays the distance and the right half the order:

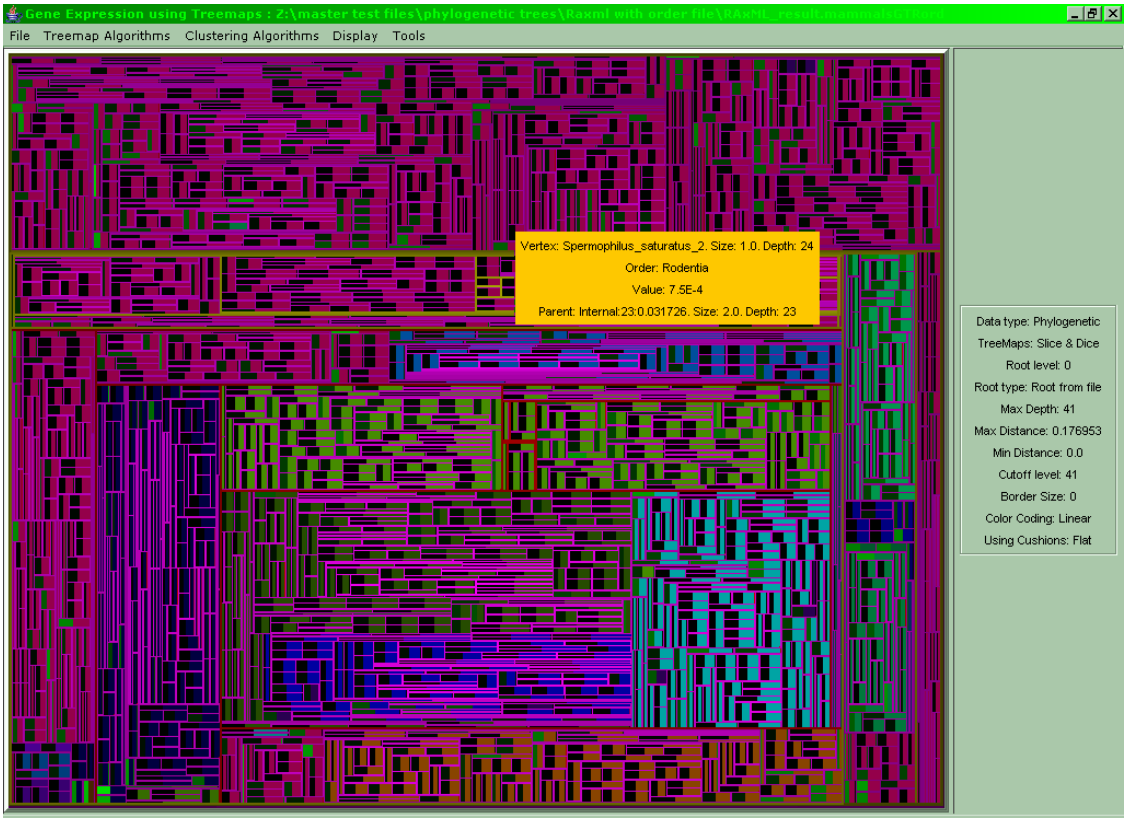


Figure 6.6.4: The left part of each node corresponds to its distance from its parent while the right part is the node's order

While the order file states only leaf nodes, the software calculates whether an internal node's children are all of the same order and, if so, colors it accordingly. This is done by performing a DFS through the tree and during the return of the last child of an internal node to its parent checking if this child and every one of its siblings are of the same order. Below is a screenshot of the phylogenetic tree from figure 6.6.3 with its internal structure shown:

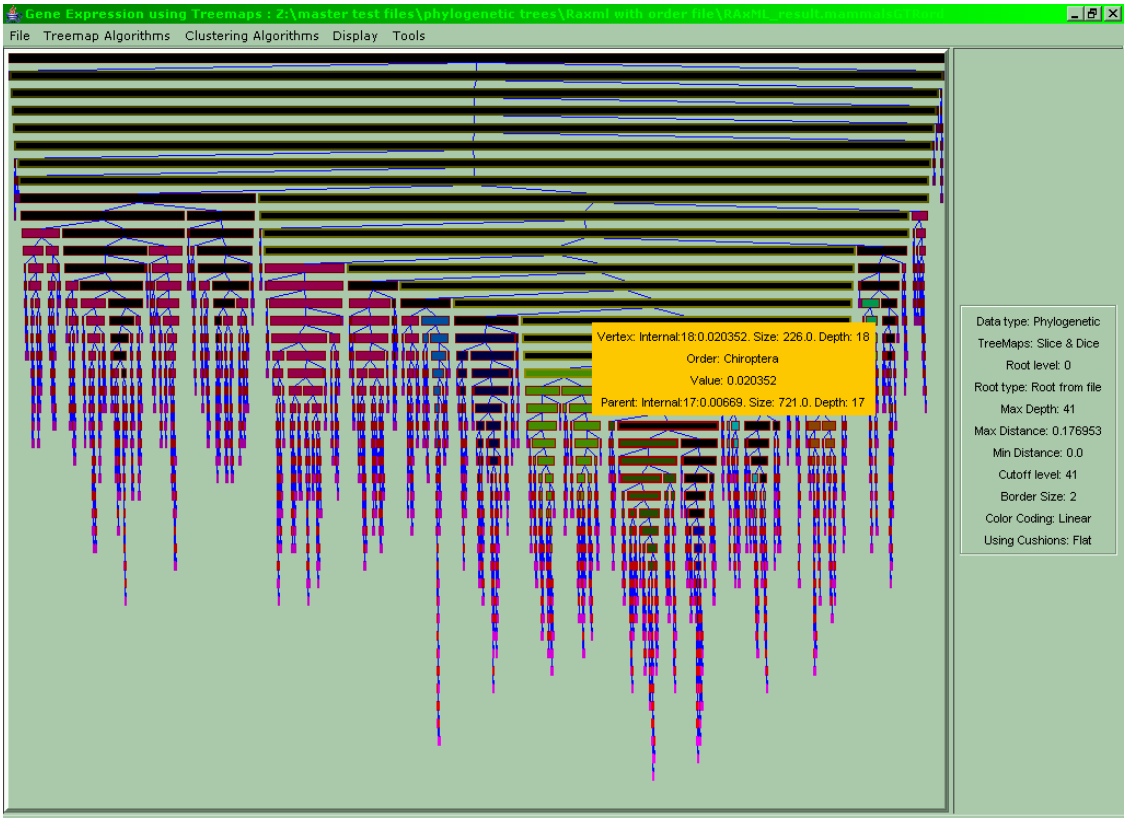


Figure 6.6.5: It can be seen that several internal nodes were assigned orders based on their children's orders

7. Conclusions and Future Work

Treemaps is an effective visualization used in many aspects of hierarchical representations such as the stock market, file systems, statistical and business analysis data, but not that much in genome research. In this thesis we present a novel way for representing clustering data along with an implementation which utilizes several added features for enhancing the treemap experience in order to provide an effective tool. Furthermore another application is implemented: using treemaps to display phylogenetic data. This fresh look at phylogenetic trees using treemaps seeks to take advantage of the treemaps' capability for displaying large trees which are common in phylogenetic trees.

But there are more things which can be added to complement the animating cluster treemaps shown here. For example an interface to the gene ontology database would be quite helpful for looking up the gene currently in view. As well as a general improvement of the various features added to the implementation so far. Also, the number of genes/patients the software can handle is quite low (around 200) due to the complexity of the clustering algorithm. Since common microarrays contain data of several thousand genes versus dozens of patients, this is a high priority issue for future development. Furthermore, more clustering algorithms and more parameters would be a useful addition.

As far as the phylogenetic aspect is concerned, several improvements can be made. Variations to the present visualization to emphasize the phylogenetic tree's unrooted nature as well as a general improvement of the various features added to the implementation so far are also high priority issues.

8. References

[1] http://www.microarrays.org/protocols.html, March 2005

[2] http://en.wikipedia.org/wiki/DNA_microarray, March 2005

[3] M. Gabig, G. Wegrzyn, “An Introduction to DNA Chips: Principles, Technology, Applications and Analysis”. Acta Biochimical Polonica vol. 48 No. 3/2001 615–622.

[4] Quackenbush: “Computational Analysis of Microarray Data”, Nature Reviews Genetics, vol. 2 June 2001, pp. 418–427.

[5] Herrero, J., Al-Shahrour, F., Díaz-Uriarte, R., Mateos, Á., Vaquerizas, J.M., Santoyo, J. & Dopazo, J. (2003). “GEPAS, a web-based resource for microarray gene expression data analysis”. Nucleic Acids Research 31(13), 3461-3467. (http://gepas.bioinfo.cnio.es)

[6] DJ Lockhart and EA Winzeler. “Genomics, gene expression and DNA arrays”. Nature, 2000, 405(6788):827-836.

[7] Acuity 4.0 software, http://www.axon.com/gn_Acuity.html

[8] Eric H Baehrecke, Niem Dang, Ketan Babaria and Ben Shneiderman, “Visualization and analysis of microarray and gene ontology data with treemaps”

[9] http://en.wikipedia.org/wiki/Phylogenetic_tree, March 2005

[10] Daniel R. Brooks and Deborah A. McLennan. 1991. “Phylogeny, Ecology, and Behaviour”, University of Chicago Press, Chicago, USA

[11] Sokal, R. R. and F. J. Rohlf. 1981. “Biometry, 2nd. Ed.” W.H. Freeman & Co., San Francisco

[12] Alexandros Stamatakis, Thomas Ludwig, and Harald Meier: ``RAxML-III: A Fast Program for Maximum Likelihood-based Inference of Large Phylogenetic Trees". To be published in Bioinformatics.

[13] Huelsenbeck J.P., Ronquist F. (2001b) “MrBayes: Bayesian Inference of Phylogenetic Trees”, Bioinformatics 17(8), 754-755.

[14] Skiena, S. S. "Nearest Neighbor Search." §8.6.5 in *The Algorithm Design Manual.* New York: Springer-Verlag, pp. 361-363, 1997.

[15] http://en.wikipedia.org/wiki/Maximum_parsimony, March 2005

[16] B. Johnson and B. Shneiderman, “Treemaps: a space-filling approach to the visualization of hierarchical information structures”. In *Proc. of the 2nd International IEEE Visualization Conference*, pages 284–291, October 1991

[17] D. M. Bruls, C. Huizing, J.J. van Wijk, "Squarified Treemaps", Proceedings of the joint Eurographics and IEEE TVCG Symposium on Visualization, 2000, Springer, pp 33-42

[18] Benjamin B. Berderson, Ben Shneiderman, and Martin Wattenberg, “Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies”. In ACM Transactions on Computer Graphics. 2002

[19] Ghoniem, M. and Fekete, J.-D., “Animating Treemaps”, in Proc. 18th HCIL Symposium - Workshop on Treemap Implementations and Applications 2001, University of Maryland, College Park, Maryland, USA, May 31, 2001

[20] Golub TR, Slonim DK, Tamayo P, Huard C, Gaasenbeek M, Mesirov JP, Coller H, Loh ML, Downing JR, Caligiuri MA, Bloomfield CD, Lander ES. (1999). Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *Science* 286(5439):531-537.

[21] Potamias G., Koumakis L., and Moustakis V. (2004). Gene Selection via Discretized Gene-Expression Profiles and Greedy Feature-Elimination. *LECT NOTES ARTIF INT – LNAI* 3025, 256-266.

[22] Potamias G., Kanterakis A., and Moustakis V. (2005). Integrated Clinico-Genonics and Gene Selection. *Computers in Biology and Medicine: an Intrernational Journal* (special issue: *Intelligent Technologies in Bioinformatics and Medicine*) (submitted).

[23] Pavlidis P. and Poirazi P. "Combining Individualized Molecular Signatures for Cancer Classification", to be submitted

Appendix A – Software Manual

This software is provided as a set of three jar files (`classes.jar`: contains the core classes of the program, `excelaccessor.jar`: contains the auxiliary classes for accessing excel files, `jbcl.jar`: contains other useful classes mostly for the proper display of the program on screen) as well as a bat file which launches the program (`run.bat`).

The only requirement for this program is a sufficiently recent version of the Java Runtime Environment. Any version equal to or above 1.4.0 will be adequate.

Upon executing the bat file `run.bat` the following application will launch:

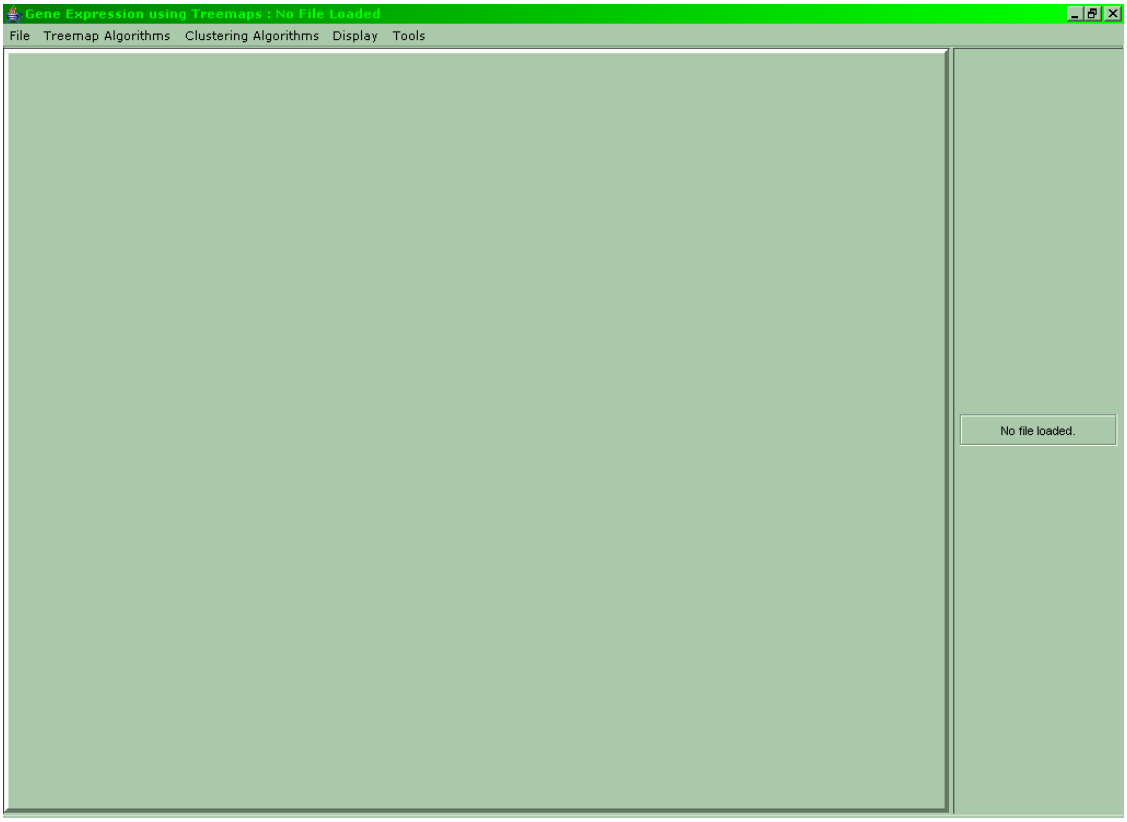


Figure A.1: The initial screen of the application

From here the user may load a file, this is the only course of action for now besides exiting:

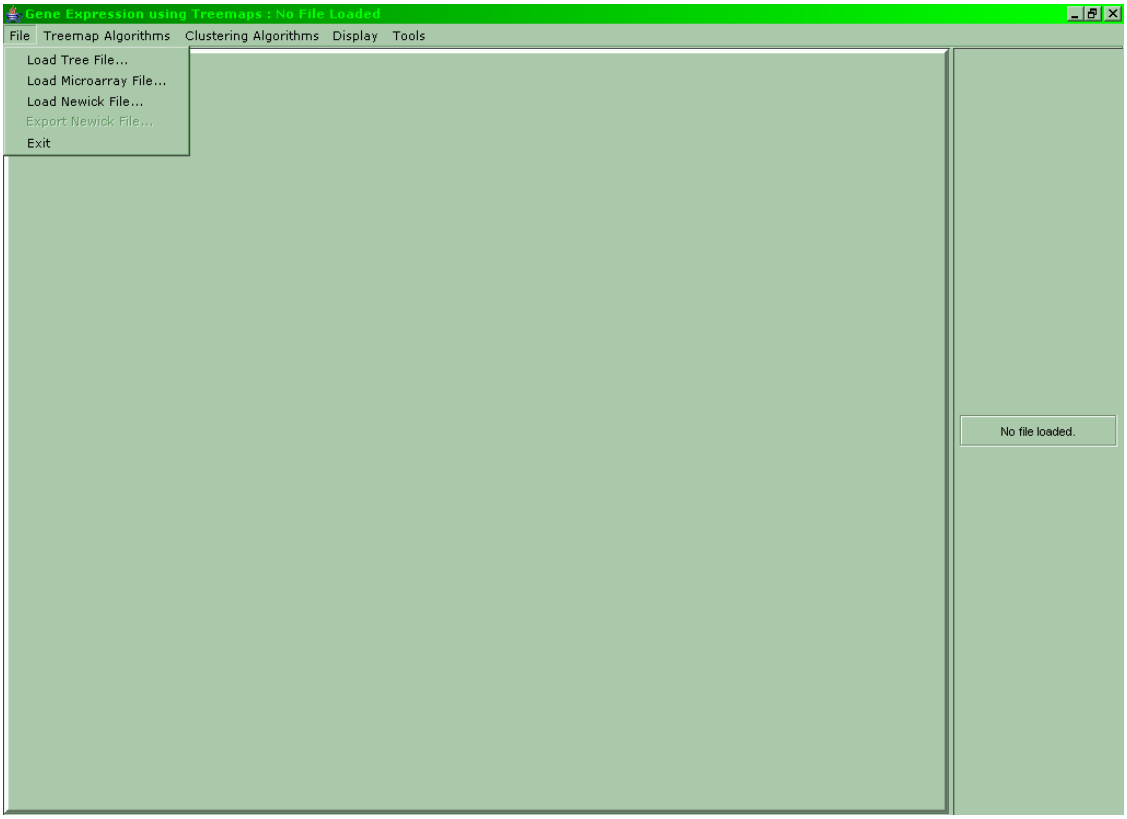


Figure A.2: The menu for loading a file

Firstly let us load a simple tree file, these files are simple text representations of trees:

```
v1 41 v2 33 v3 2 v4 6
v2 33 v5 13 v4 20
v5 13 v6 12 v7 1
```

Vertex v1 with weight 41 has three children: vertex v2 with weight 16, v3 with weight 1 and v4 with weight 3. The weights need not be necessarily positive integers, any value will be displayed. The line below that is the same except it concerns vertex v2 and its children. After selecting to load a tree file the file browser appears:

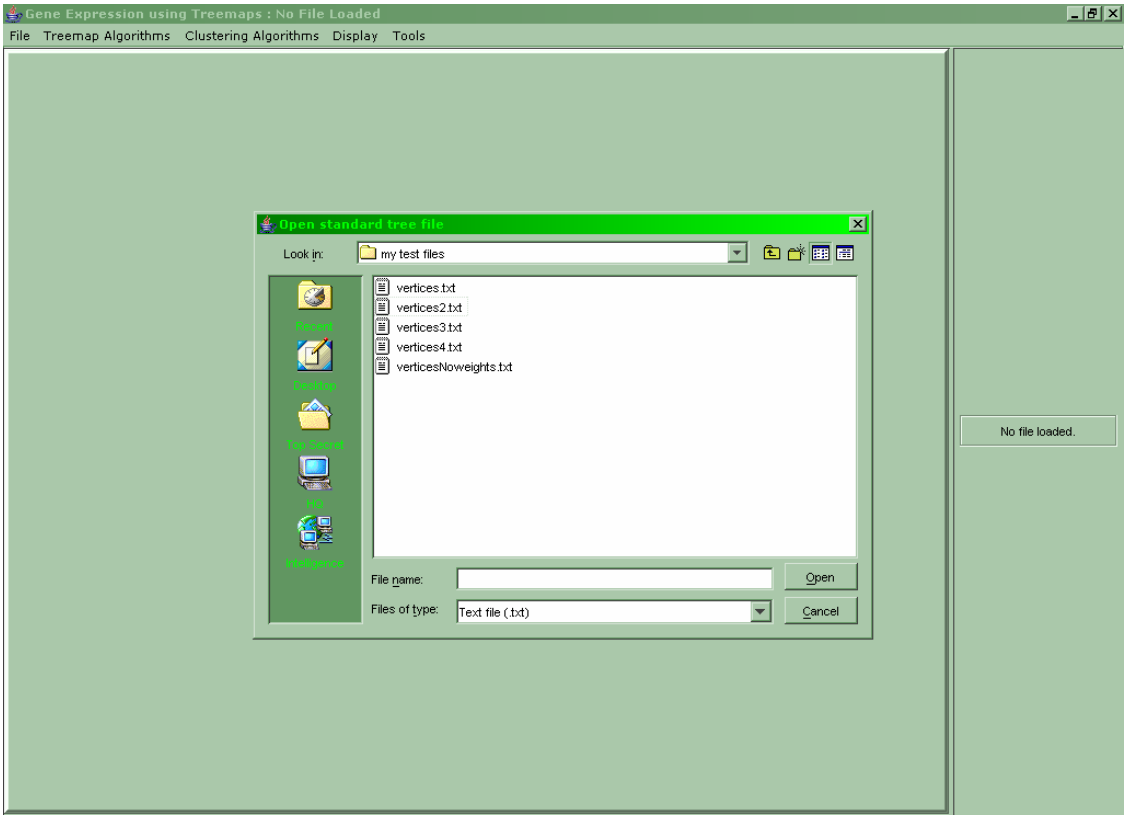


Figure A.3: The file load browser and the selected file

After locating and selecting a tree file the treemap will appear:

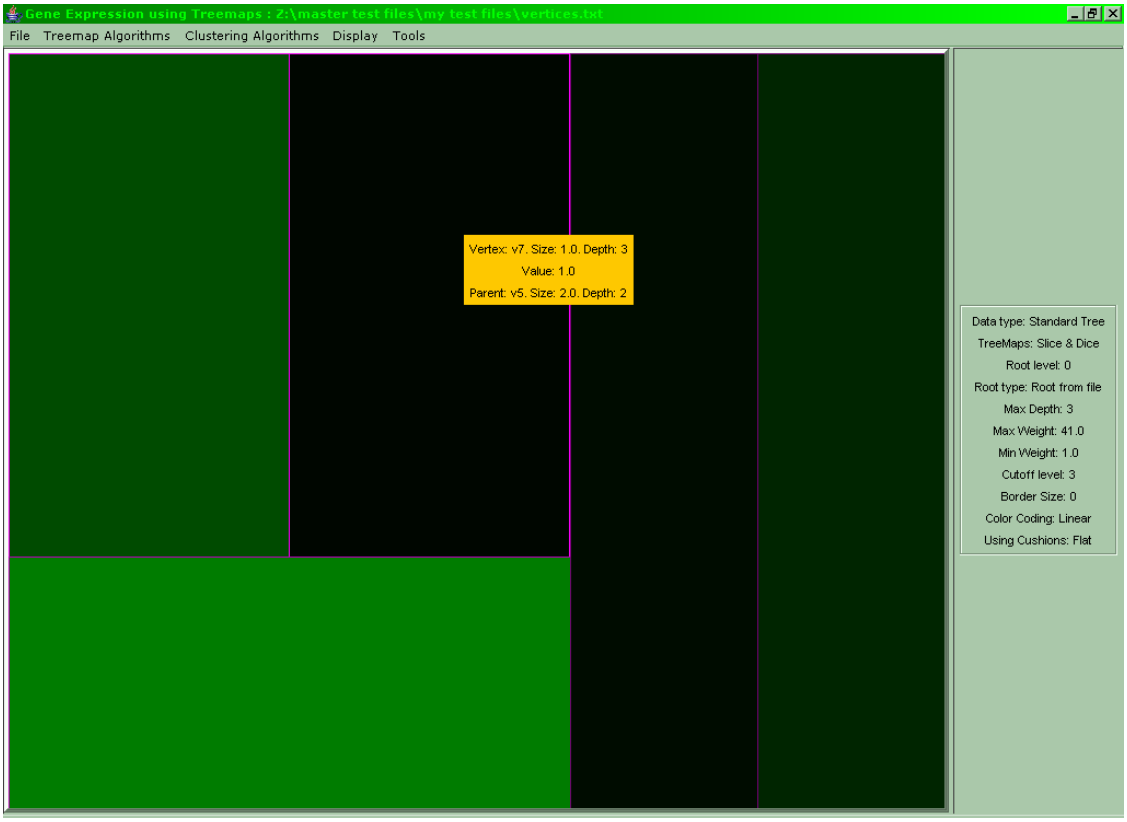


Figure A.4: The tree file described above after being loaded

The info window on the right side displays useful information about the currently loaded treemap. From this point on the user may use several features:

Firstly, a left click anywhere on the treemap will make the popup window display the full path to the root of the tree. Normally it displays just the current node and its parent as shown in figure A.4, below is the same treemap with the full path:

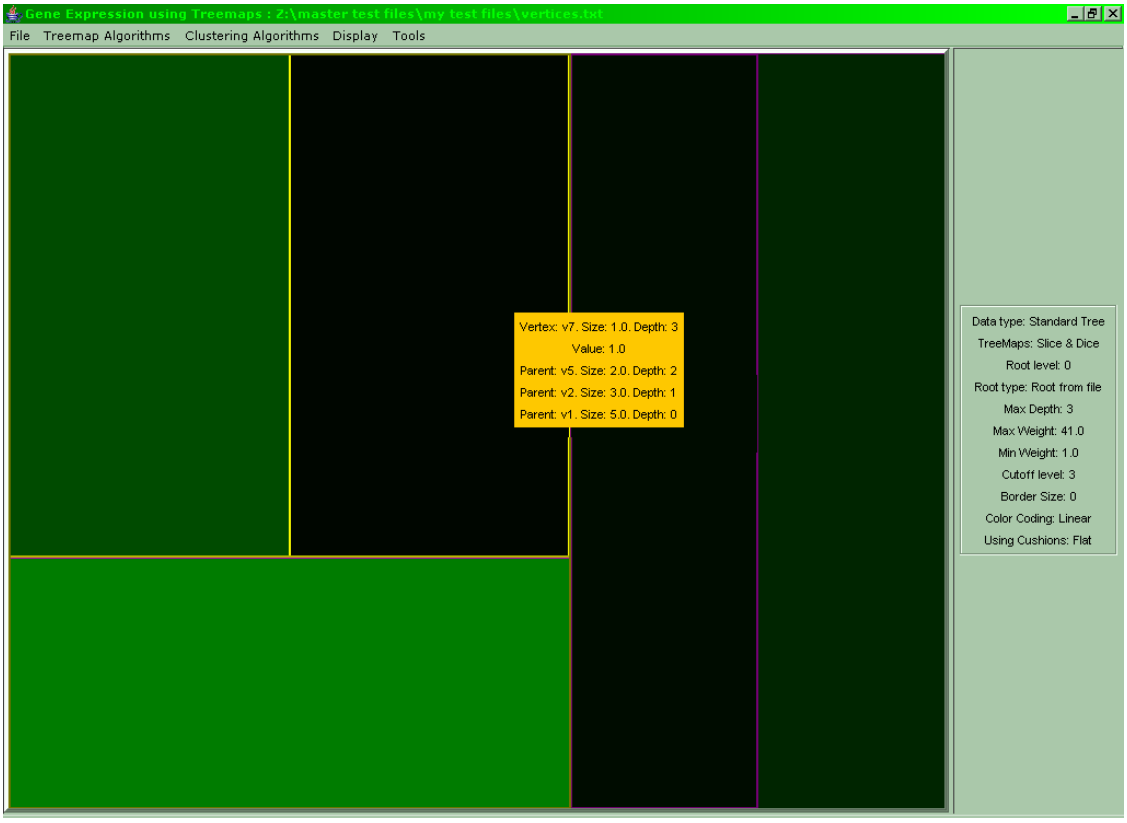


Figure A.5: The treemap from the above figure with the full path shown

The user may use the squarified algorithm instead of the slice & dice:

Figure A.6: After selecting "Treemap Algorithms -> Squarified". The standard tree displayed using the squarified algorithm

In order to return to the slice & dice treemap the user may select **`Treemap Algorithms -> Slice & Dice`**.

From the Display menu several features can be activated which affect the appearance of the treemap but not its internal structure.

Firstly the user may affect the size of each node, by default every leaf gets a size of 1, and every internal node gets the sum of sizes of its children:

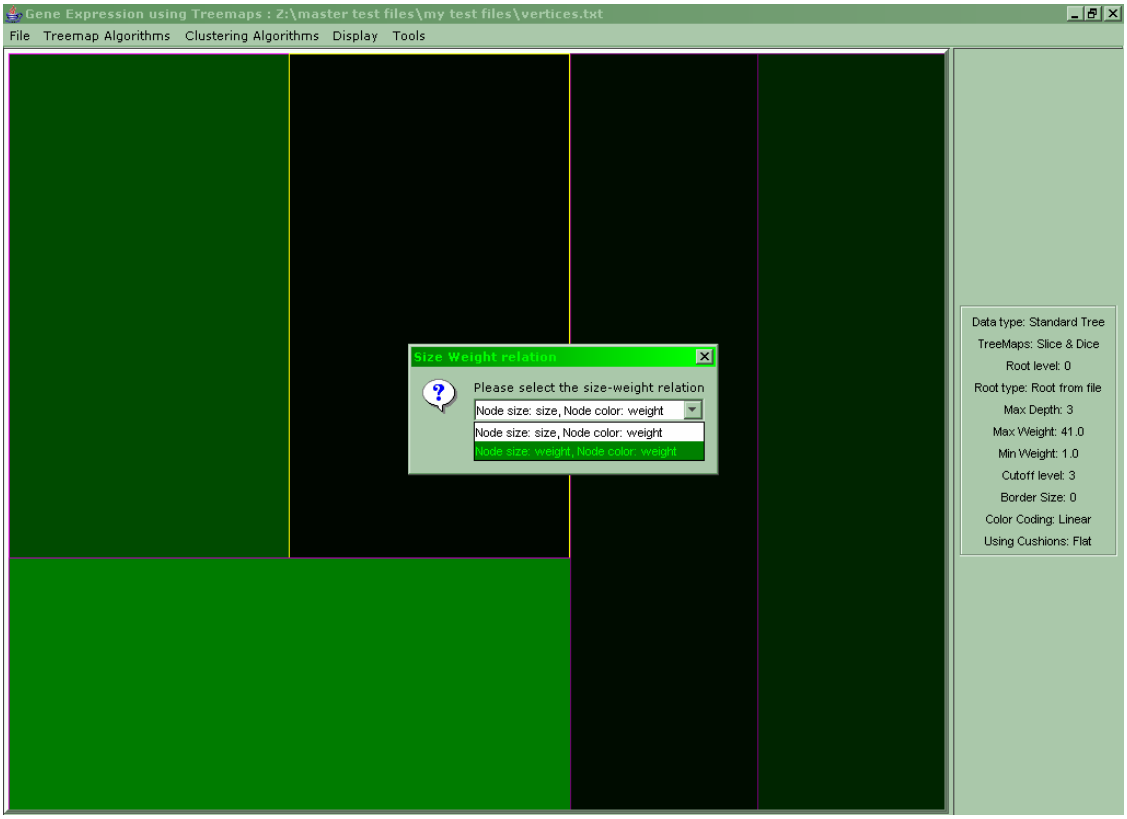


Figure A.7: Once the user has selected the "Display -> Size - Weight..." option the above dialog appears and the choice "Node size: weight, Node color: weight" should be made in order to activate the function

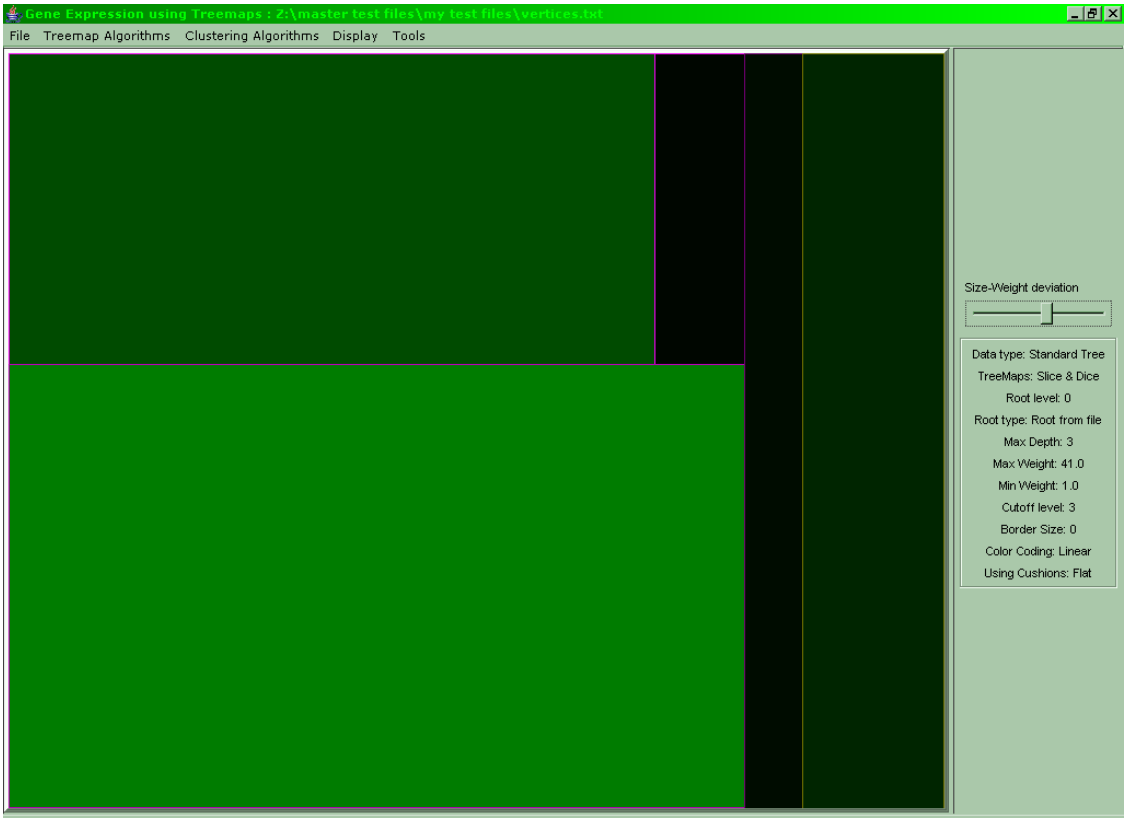


Figure A.8: The slider on the right appears, by altering it the nodes with greater weight values receive a greater size value (the brighter nodes become bigger)

To deactivate this option the user should just choose **`Display -> Size-Weight`**… and select the option **`Node size: size, Node weight: weight`**.

Next in the **`Display`** menu is the function which changes the color-coding. The user should select the menu: **`Display -> Color Coding`**…

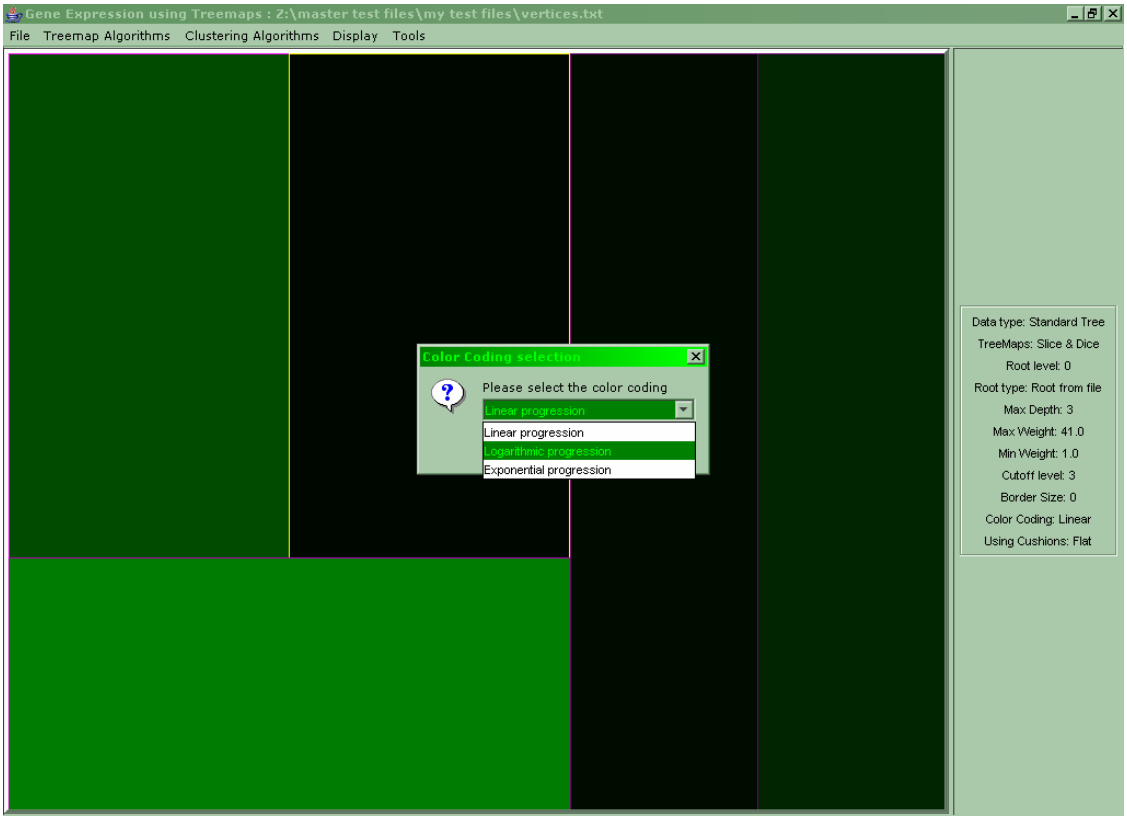


Figure A.9: The above dialog appears once the menu Color Coding… has been selected. Logarithmic progression is selected

Figure A.10: The color coding is done logarithmically, minimum values differ most from the rest (the two nodes at the center of the display). A border is added to show the internal nodes of the treemap. The slider on the right appears and allows the user to adjust the difference in intensity among the nodes

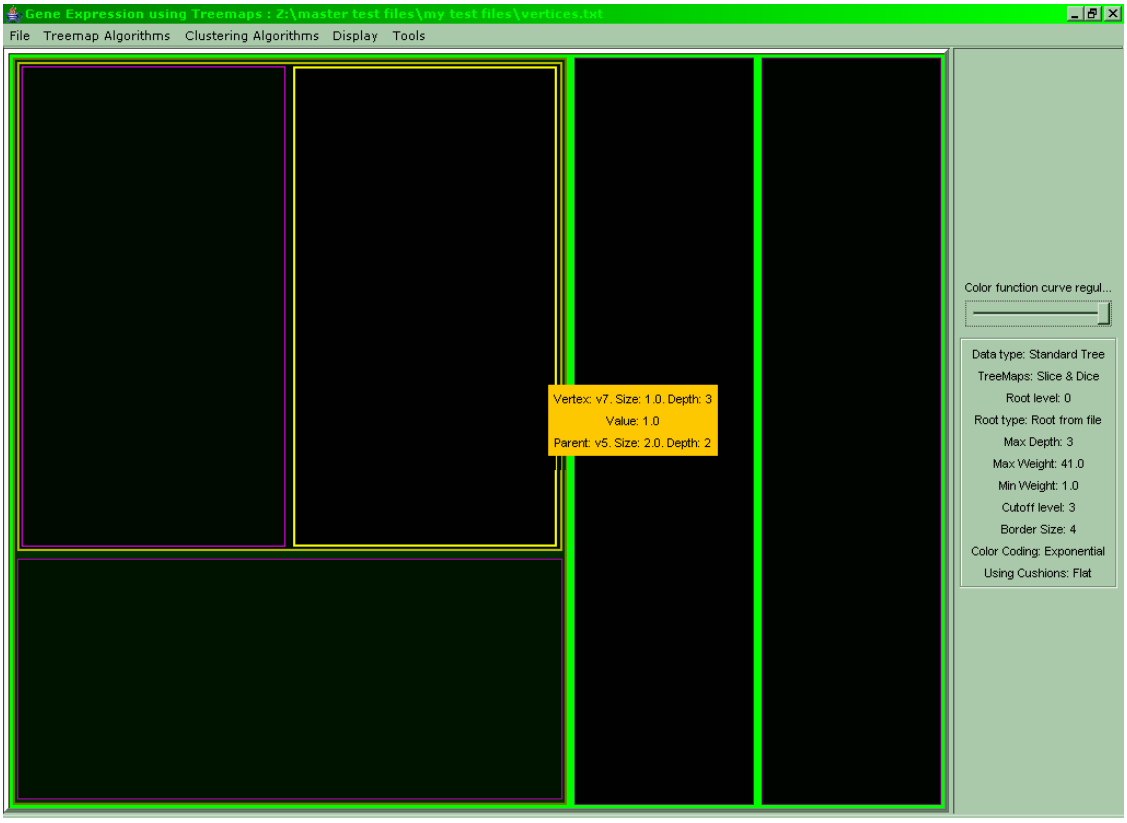


Figure A.11: Should the user select an exponential progression the nodes with maximum values differ the most (the root of the tree). Again the slider on the right allows the user to determine this difference quantitatively

In order to return to the original color coding, the user should again select the **`Display -> Color Coding`**... menu, and from the menu that follows **`Linear progression`**.

The next menu item in the **`Display`** menu is for altering the border size:

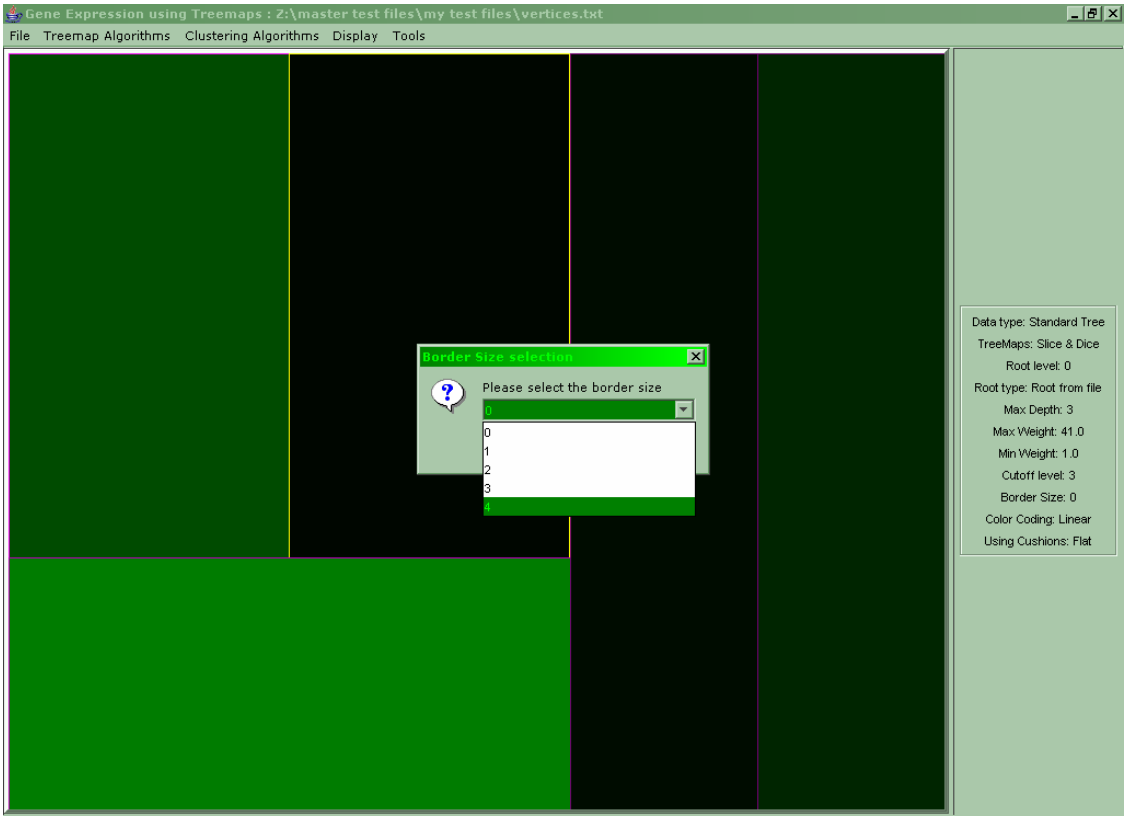


Figure A.12: When the user selects the "Display -> Border Size..." menu the above dialog appears and prompts the user to select the border size. Since this is a small treemap and space is not very valuable the maximum is selected

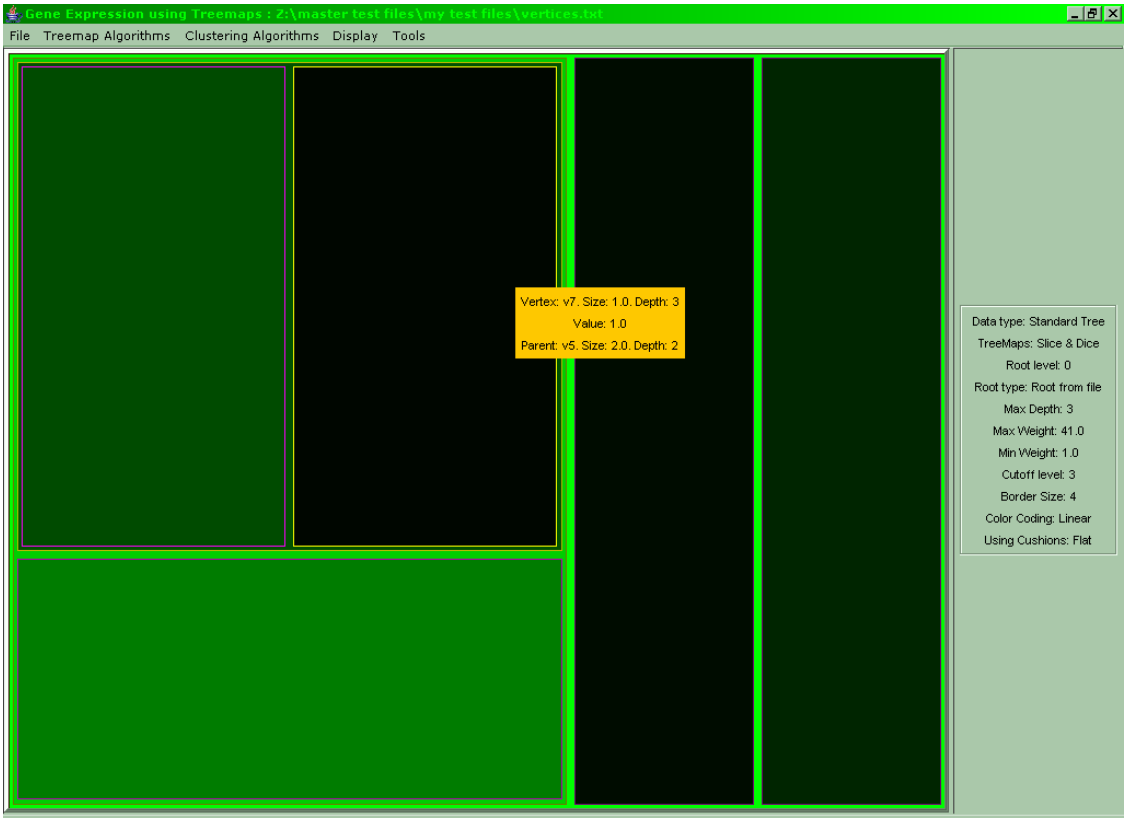


Figure A.13: The resulting treemap has an invisible border of 4 pixels to each side relative to its parent rectangle

Again in order to return to the original treemap the same menu should be selected and a border value of 0 chosen.

The next menu item: **`Display -> Thin Nodes…`** is useful for large treemaps where only small areas are given to each node and it is very probable that a node literally disappears due to lack of space. In the case of the standard tree shown above this is not the case, so activating this feature will not yield any difference. Instead a phylogenetic tree will be used to demonstrate this. Note that when loading a phylogenetic newick file a dialog will appear asking for an *order* file, for now it is sufficient to press **`"No"`**, the importance of order files will be explained later. Below we see a phylogenetic tree:

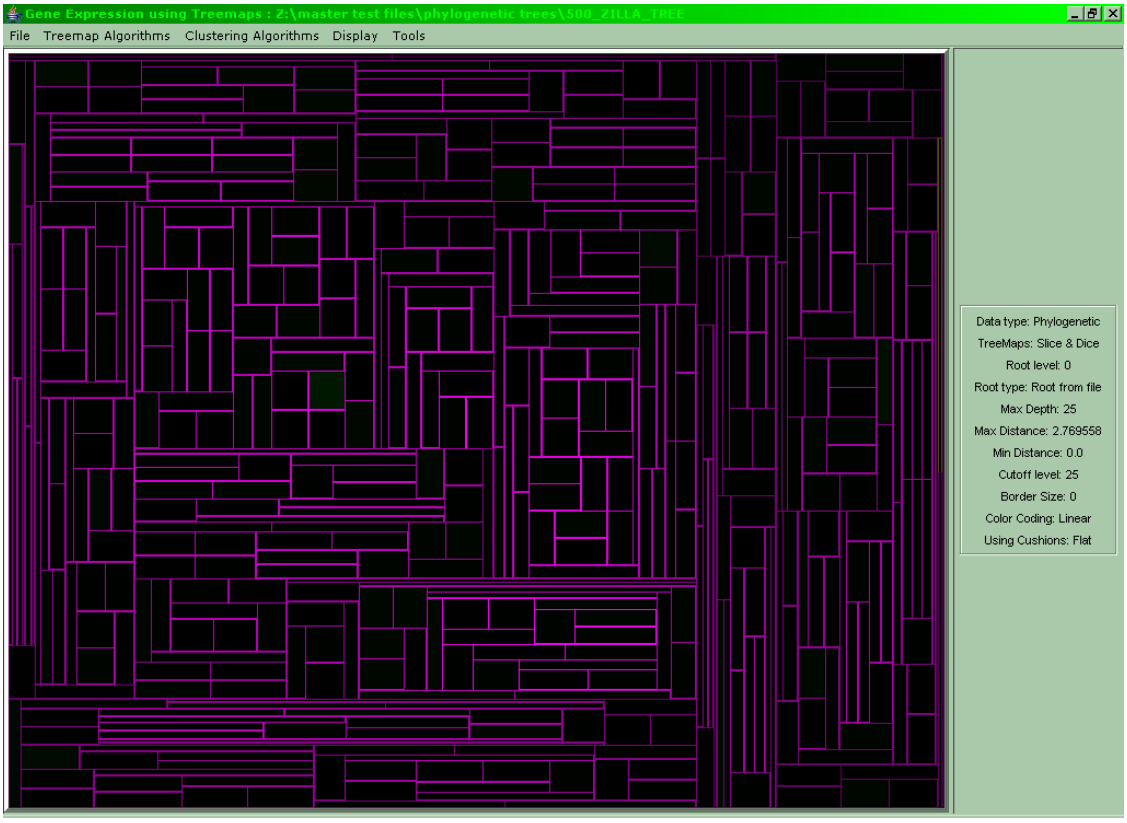


Figure A.14: In the above phylogenetic tree there is no visible leaf with maximum value (bright green)

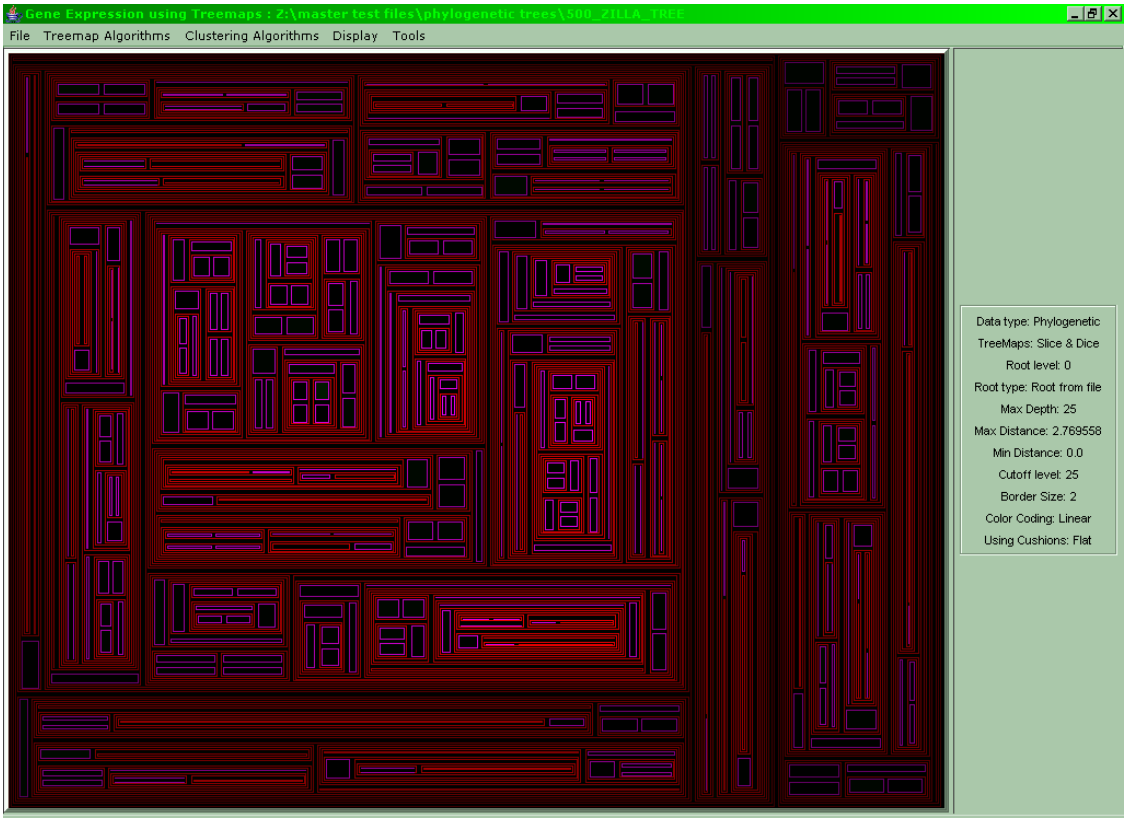


Figure A.15: Even when checking the internal nodes no maximum can be found

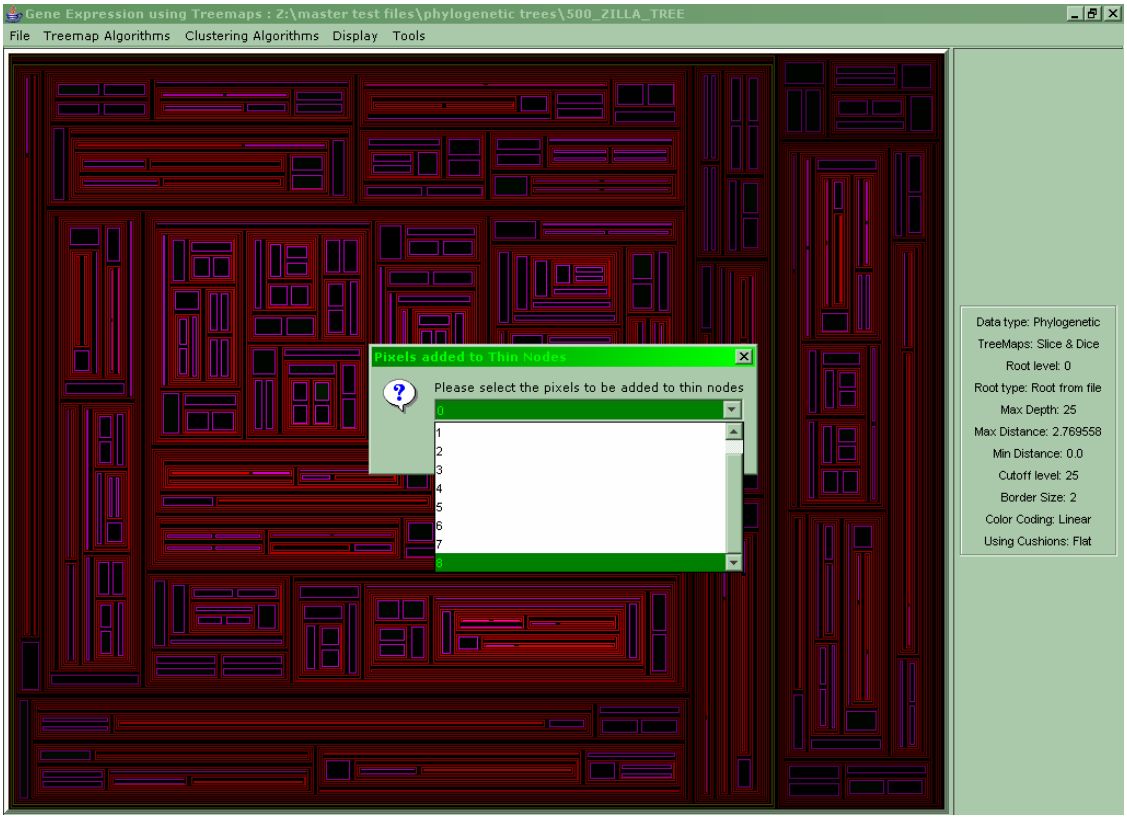


Figure A.16: After selecting "Display -> Thin Nodes…" a window appears where the number of added pixels is chosen, the maximum of 8 pixels is selected

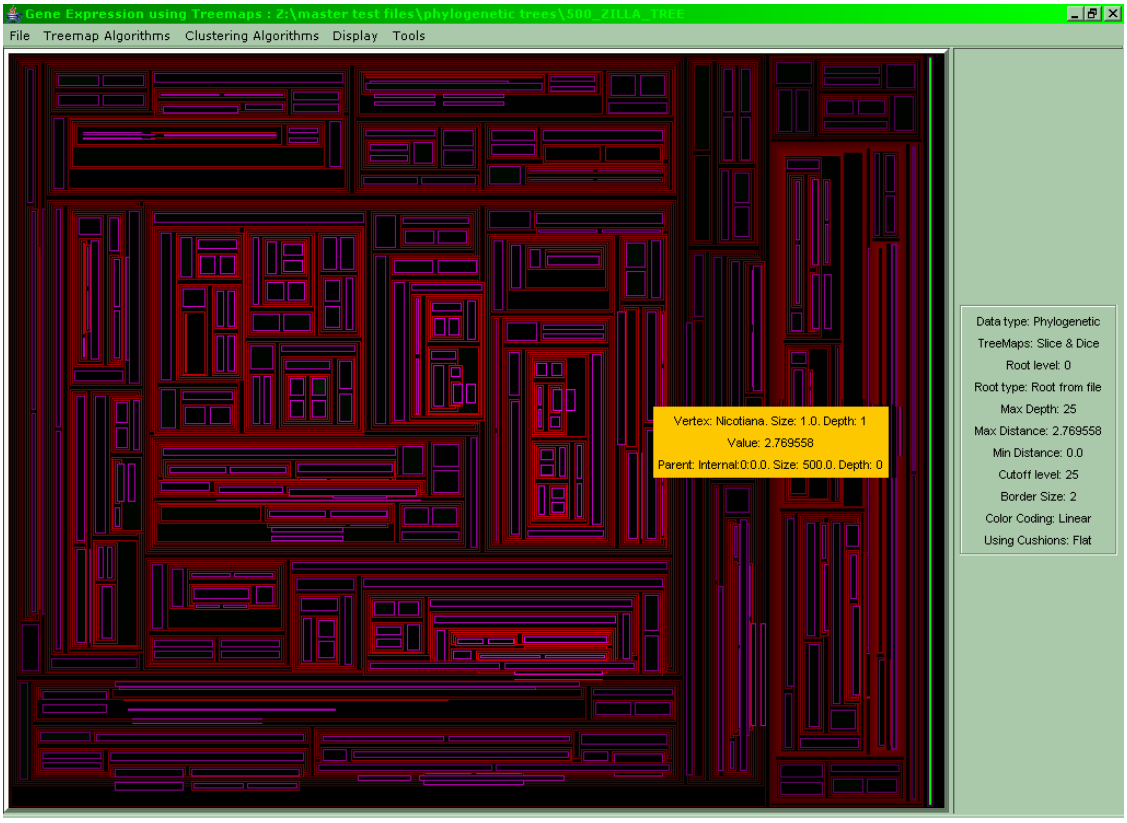


Figure A.17: The treemap becomes quite distorted, but the lost node is found

The distortion seen in the last figure is due to the lack of space already caused by adding a border, so small nodes become even smaller and the addition of 8 pixels worsens this situation. This feature should be used solely for the detection of lost nodes.

Next in the **`Display`** menu is the menu **`Cushion Treemaps`** which toggles between cushion and flat areas for each node.

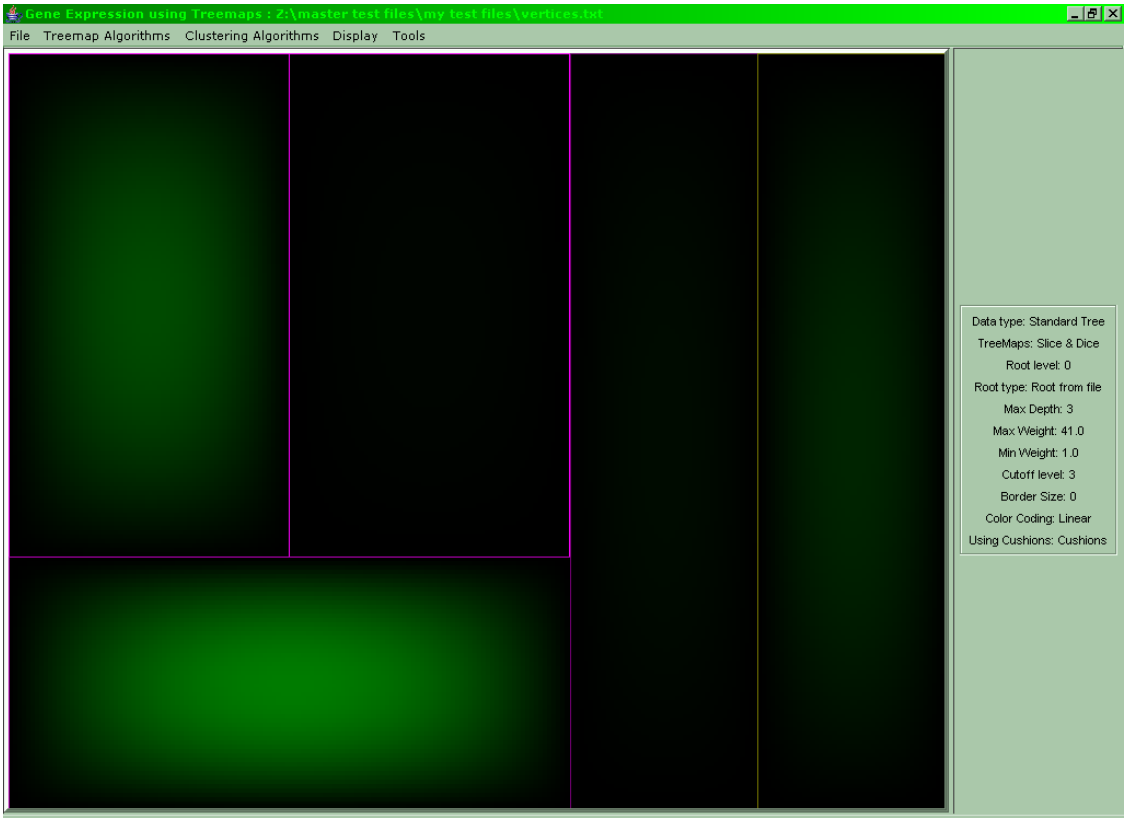


Figure A.18: When the user selects the menu Display -> Cushion Treemaps the flat areas are replaced by cushions

In order to return to flat nodes the same menu should be used.

The last item of the `Display` menu is used exclusively for animating microarray treemaps, so a microarray is used for this feature. Below a screenshot of two animating microarray treemaps is shown:

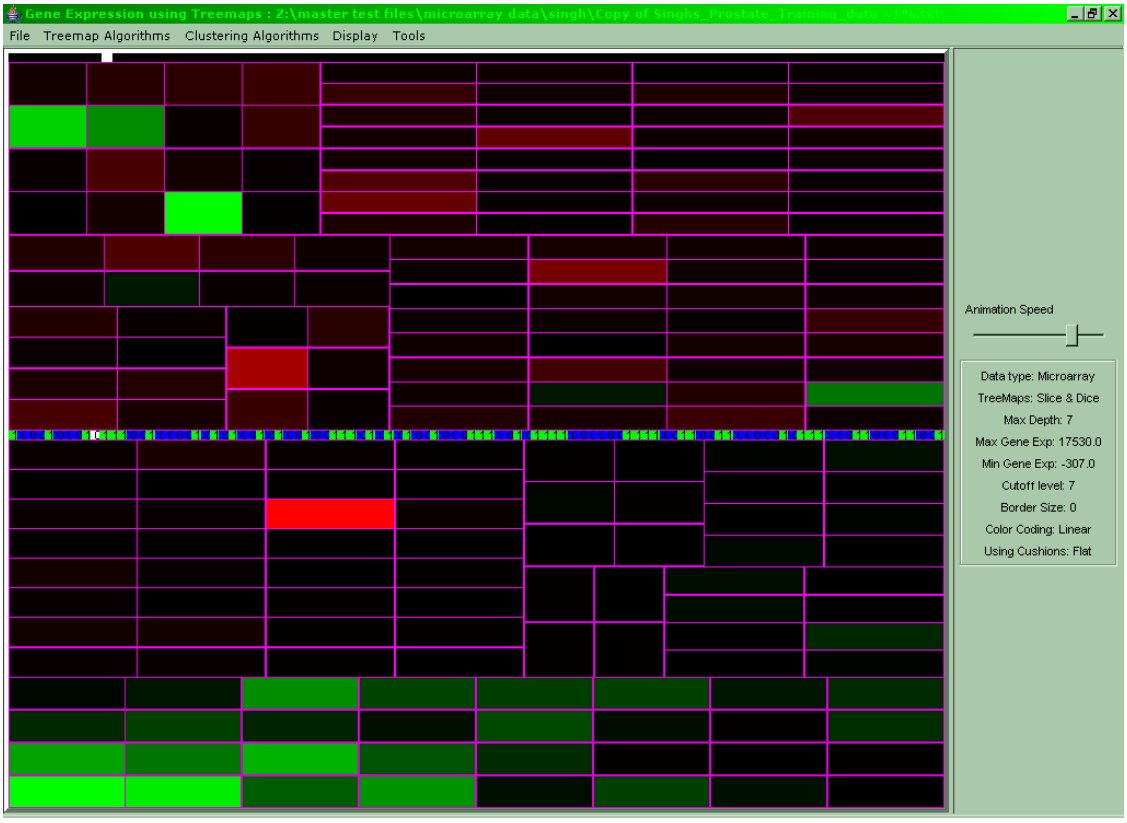


Figure A.19: A double clustered microarray shown as two animating treemaps, each node represents a gene expression

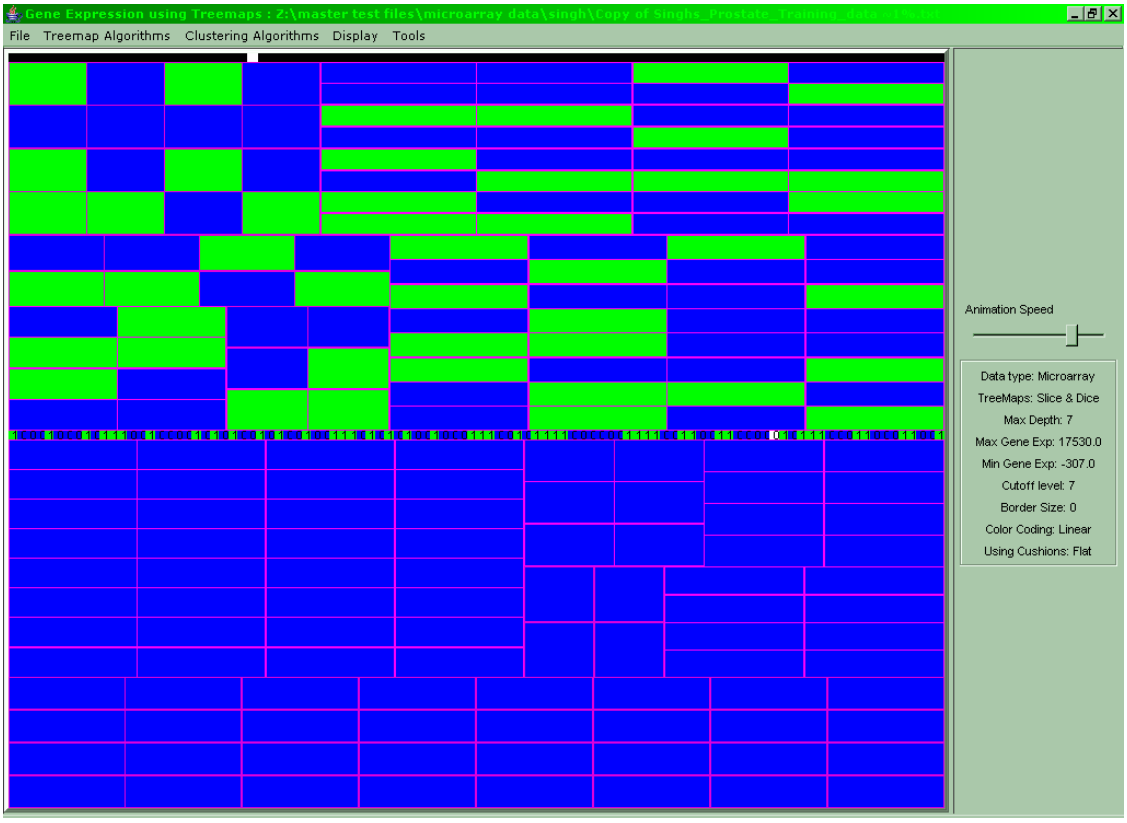


Figure A.20: Figure A.24 after selecting the "Display -> Show Classification" menu item. Every leaf is shown in the color corresponding to its classification

In order to return to the gene expression view the user should select once again the **Display -> Show Classification** menu.

The Tools menu contains functions regarding the altering of the tree structure. The first menu item allows the user to cut off levels of nodes from the tree in order to reveal internal nodes of the tree:

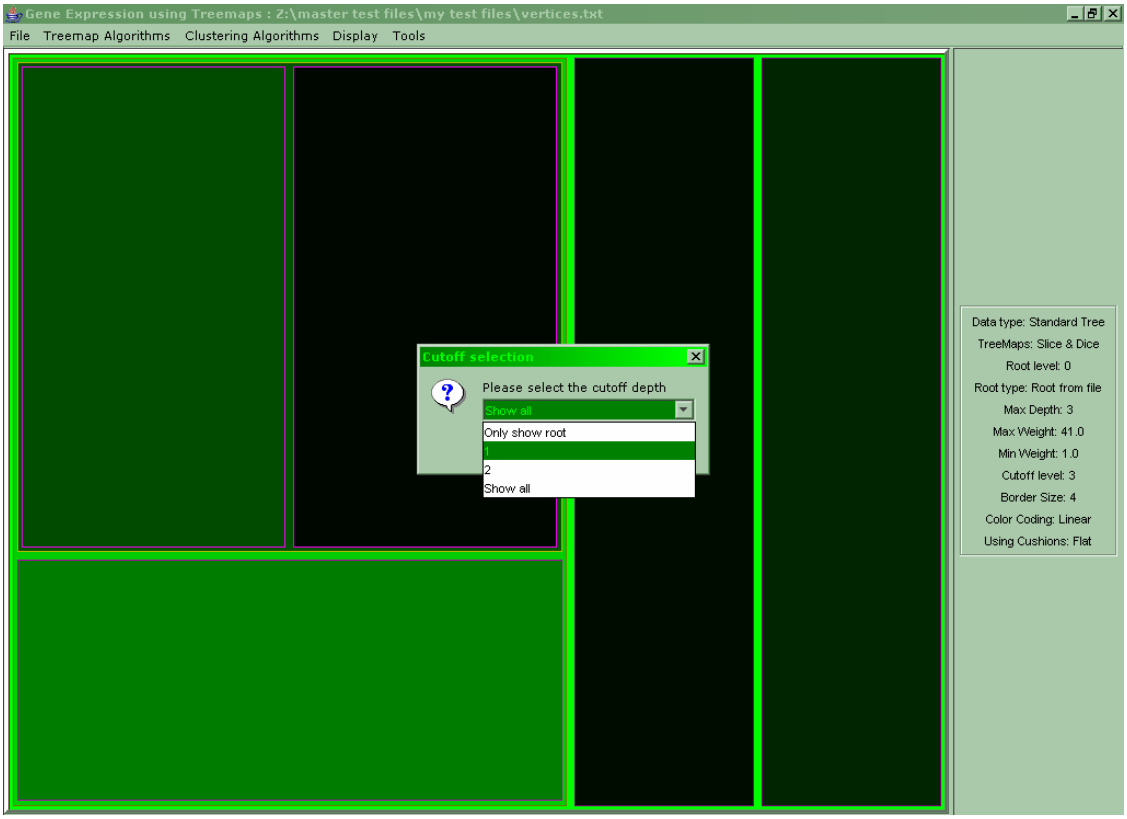


Figure A.21: When the user selects the "Tools -> Cut Leaves…" menu the above dialog appears asking for the cut-off level. With level 1 selected everything on levels 2, 3, etc. disappears

Figure A.22: After activating the cut-off feature only levels 0 and 1 are displayed

To revert to the full tree again the same menu item should be activated and in the dialog that follows the option `Show All` should be selected.

The next menu item in the `Tools` menu enables the user to display the treemap as a standard tree:

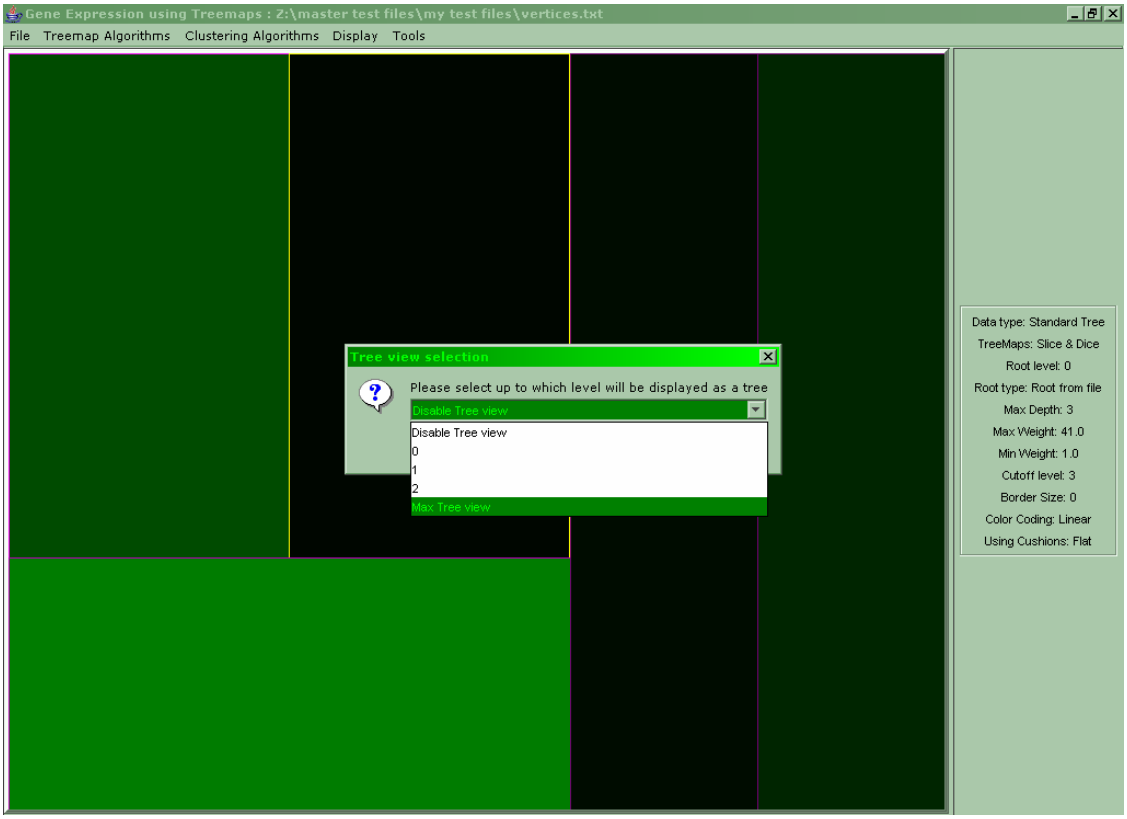


Figure A.23: Activating the menu item Tools -> Show as Tree... causes the above dialog to appear which asks for the level up to which to display as a tree. Currently the maximum is selected which is the entire treemap

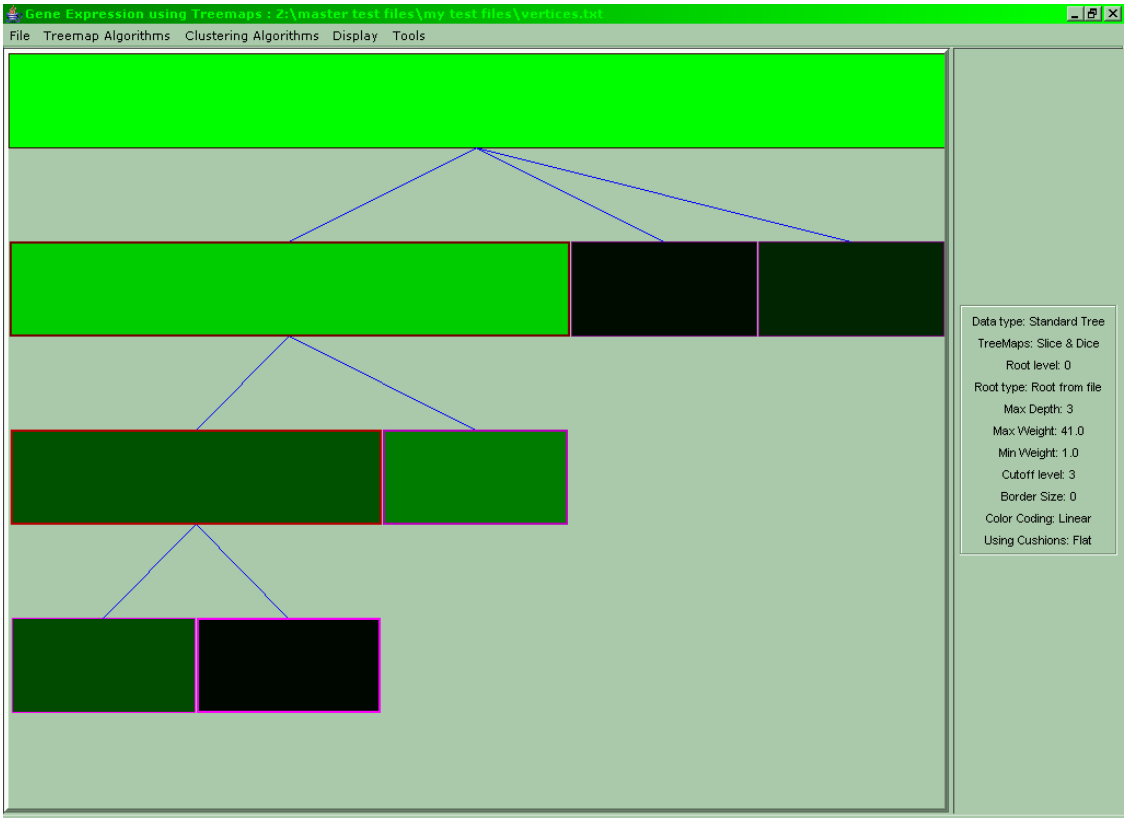


Figure A.24: After setting the entire treemap to appear as a tree the above tree is shown. The width of the screen is divided into as many leaves as the tree has and every parent receives a width which corresponds to the number of leaves originating from its subtree

To go back to the full treemap the same menu item should be selected and the choice **`Disable Tree view`** should be made.

Finally, the last function of the **`Tools`** menu allows the user to set the center of the tree as its root:

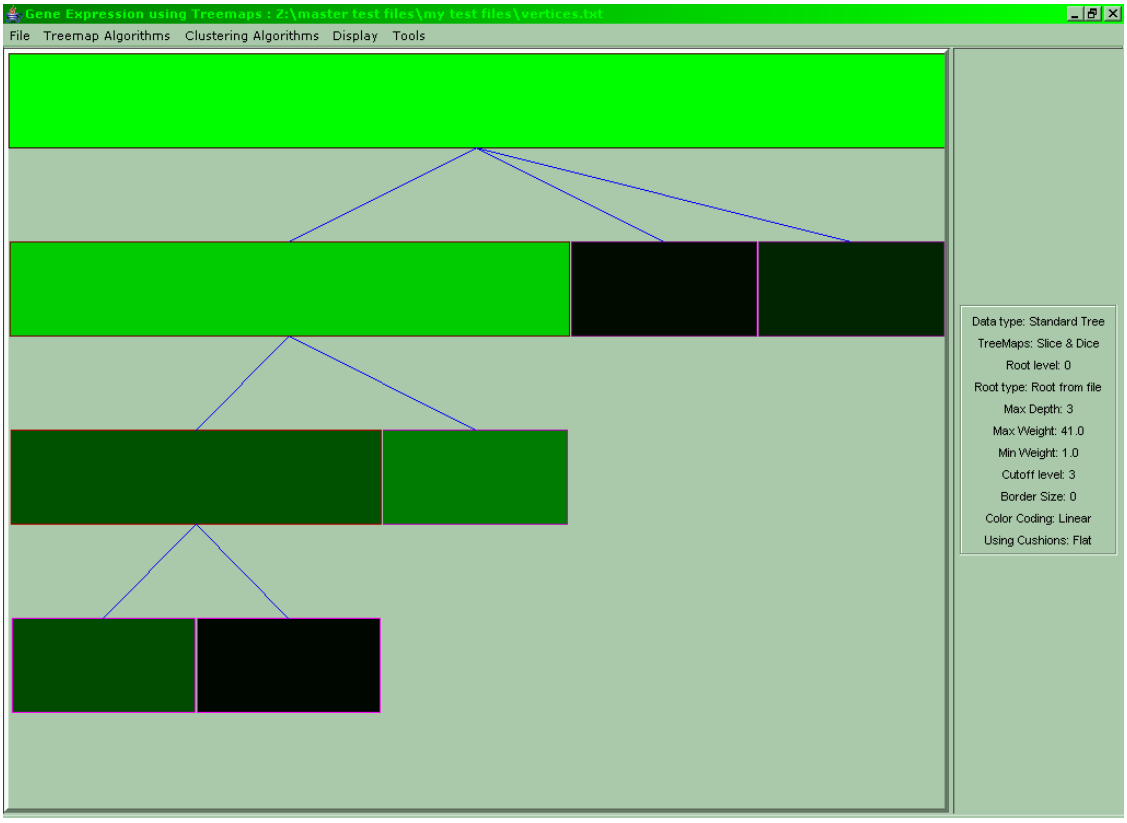


Figure A.25: The original tree shown as a tree to display the changes about to be made by selecting the Tools -> Redesign to Tree Center

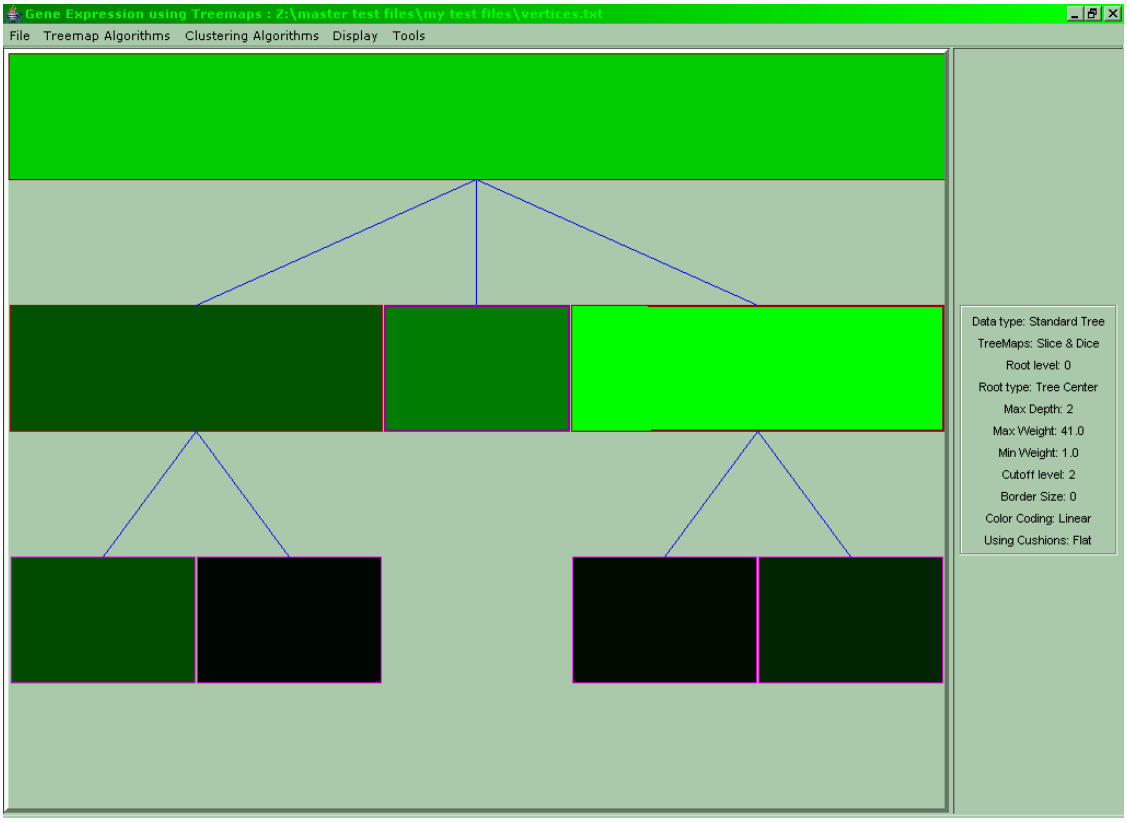


Figure A.26: After setting the center of the tree as its root

This menu item is also a toggle function, meaning that to return to the original root the same menu should be chosen.

By right clicking on any node of the treemap several other functions are made available:

Figure A.27: The right click menu on a standard tree. The node selected is named v7

The first function of this menu, `Zoom to Root`, allows the user to set the currently selected node as the root of the visible tree. In the case shown above this would lead to the following figure:

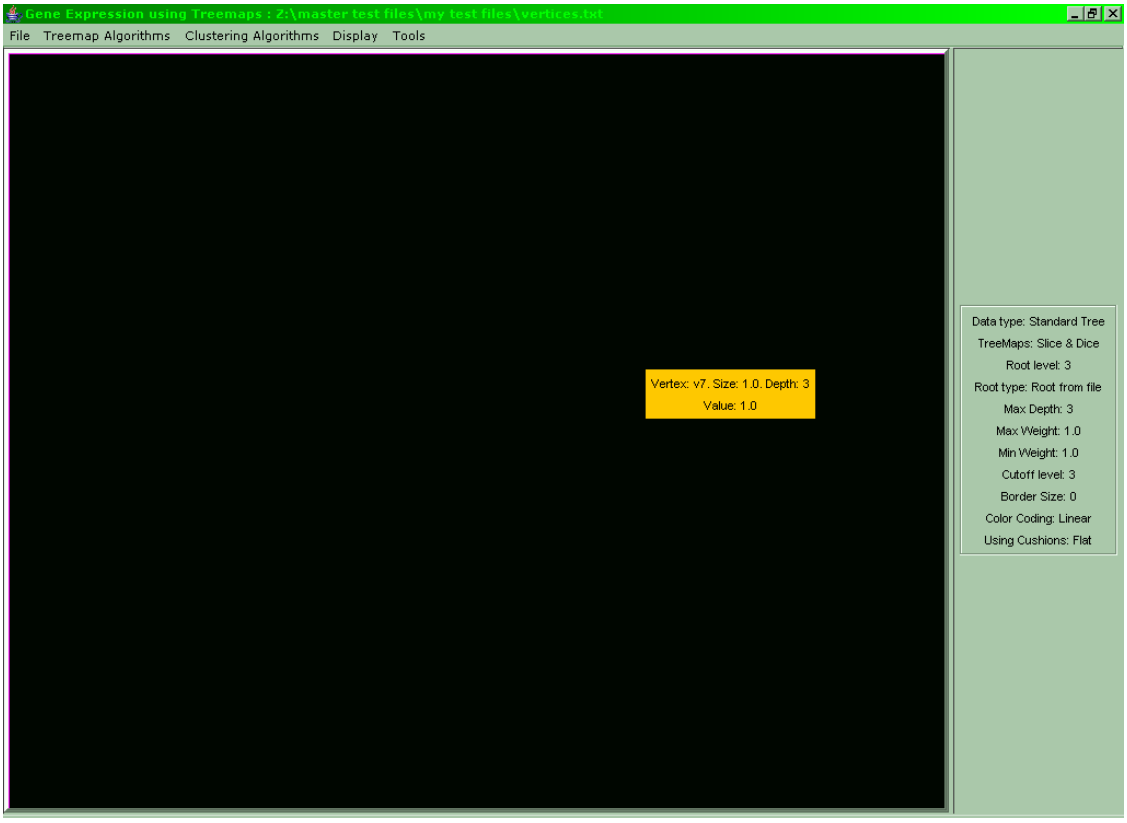


Figure A.28: After zooming into a leaf node the entire tree becomes just one node which is the previously selected leaf v7

When in such a zoomed state right clicking on any node and selecting `Go up one Level` results in the following tree:

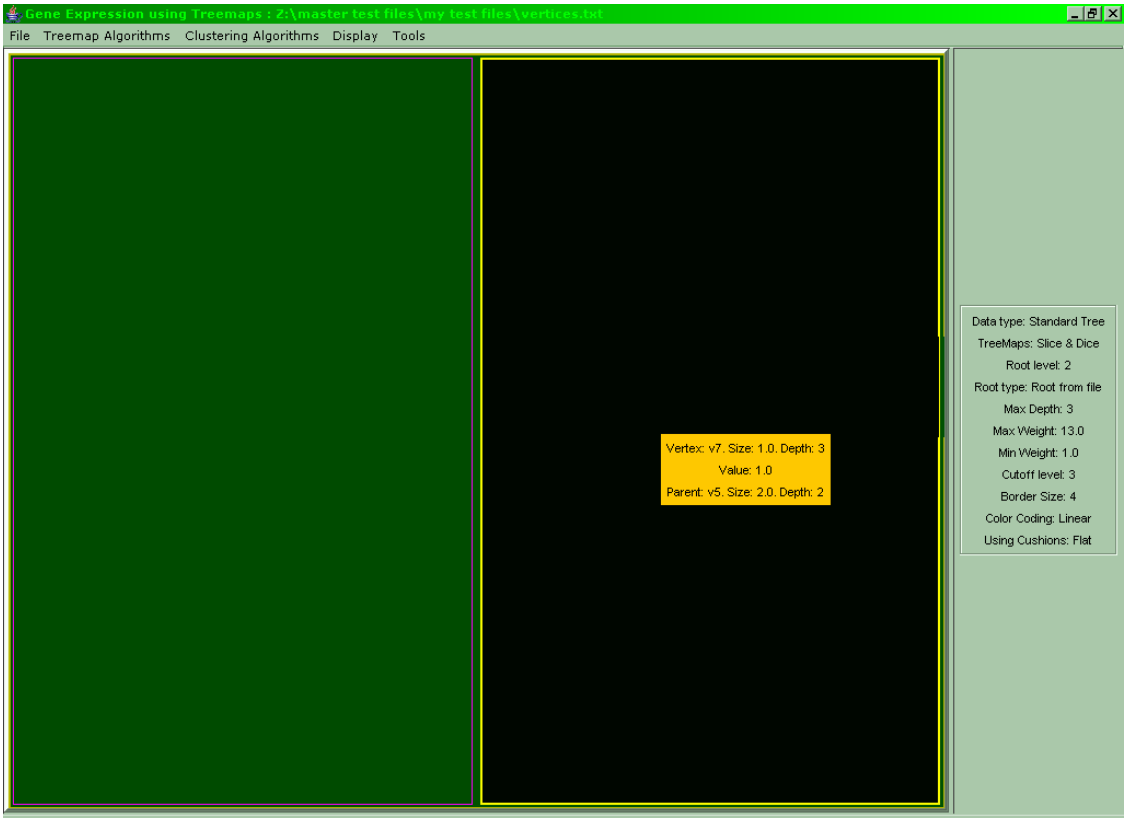


Figure A.29: When "Go up one Level" is selected on the treemap, the parent of node v7 (which was until now the root) becomes the visible root and the subtree originating from it becomes visible

By repeating this process the user can reach the original root of the tree at which this command yields no result.

The last feature of the right click menu, `Redesign to Root`, allows the user to set the currently selected node as the root of the treemap by redesigning the tree and not by cutting off any nodes:

Figure A.30: The previously selected leaf v7 has just been promoted to the root of the tree. All other nodes connected to it (children and its parent) are currently only children

Another useful feature is the ability to export standard and phylogenetic trees as newick files (microarrays can also be exported, but that function will be discussed later). What is interesting is that structural alterations on the tree like redesigning its root or zooming into a node are saved. So a tree with a different root than the one the user loaded or a subtree of the original tree can be exported.

The functions described so far work equally for standard and phylogenetic trees.

For microarrays some features are available while others are not. Moreover some features are exclusively for microarray datasets.

After loading a microarray the user is presented with the following dialog in case the microarray dataset is classified:

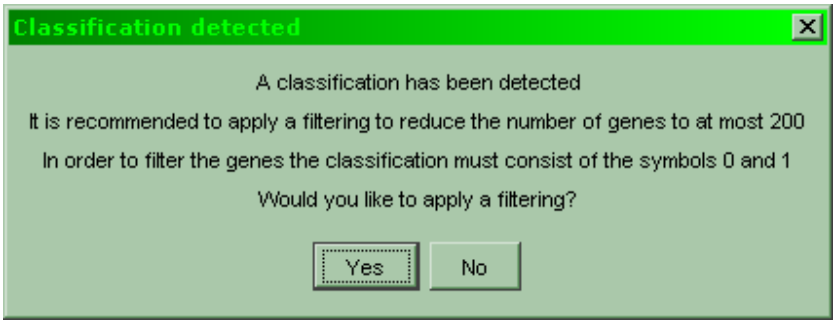


Figure A.31: The dialog prompting the user to choose whether to filter the microarray or not

Should the user select `Yes` the following dialog appears:

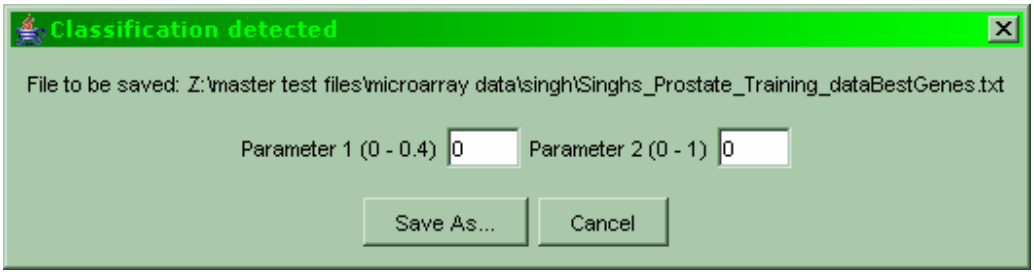


Figure A.32: The dialog asking for the user to enter the parameters for the filtering. Effective values are 0 to 0.4 for parameter 1 and 0 to 1 for parameter 2. Values of 0 for both parameters will leave the file as it is

After the algorithm concludes another dialog will appear:

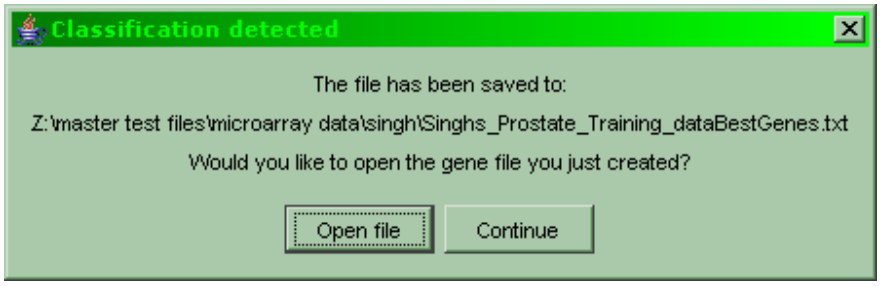


Figure A.33: The dialog that follows the conclusion of the filtering algorithm

`Open File` opens the newly created file while `Continue` continues with the file previously loaded.

When the microarray has loaded the following will appear on screen:

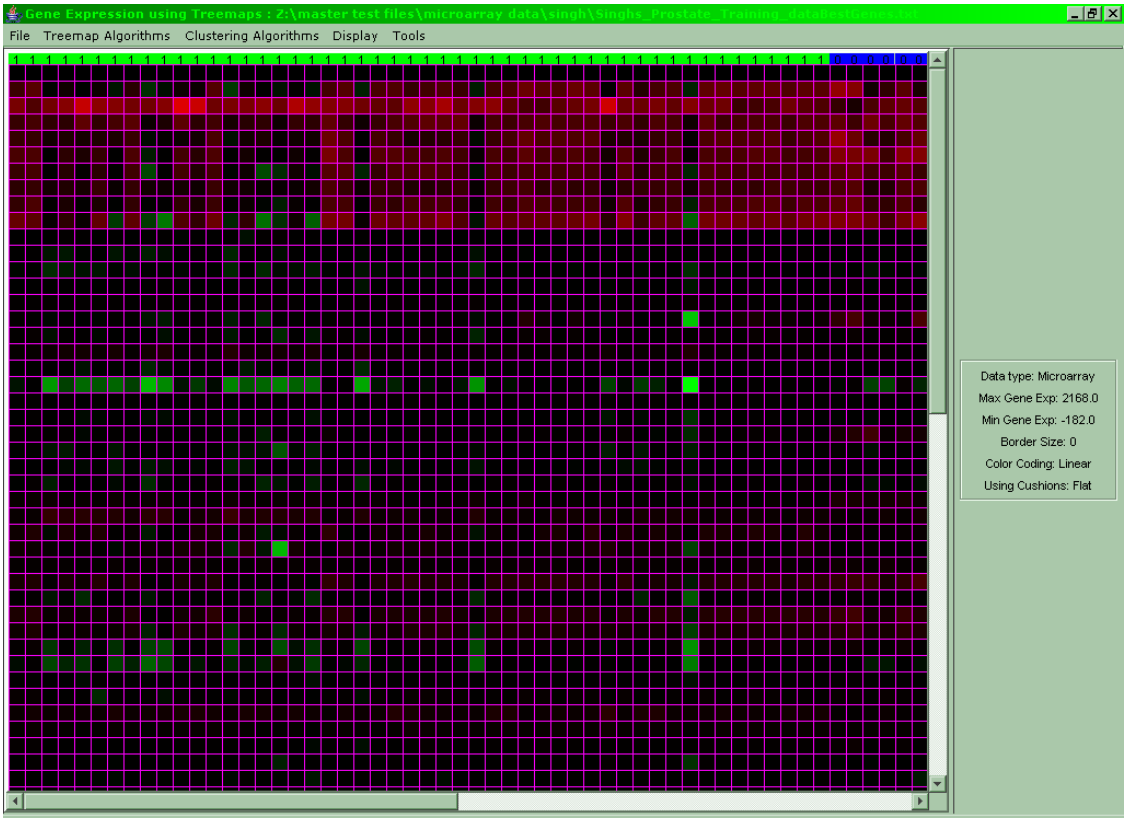


Figure A.34: A microarray has just been loaded

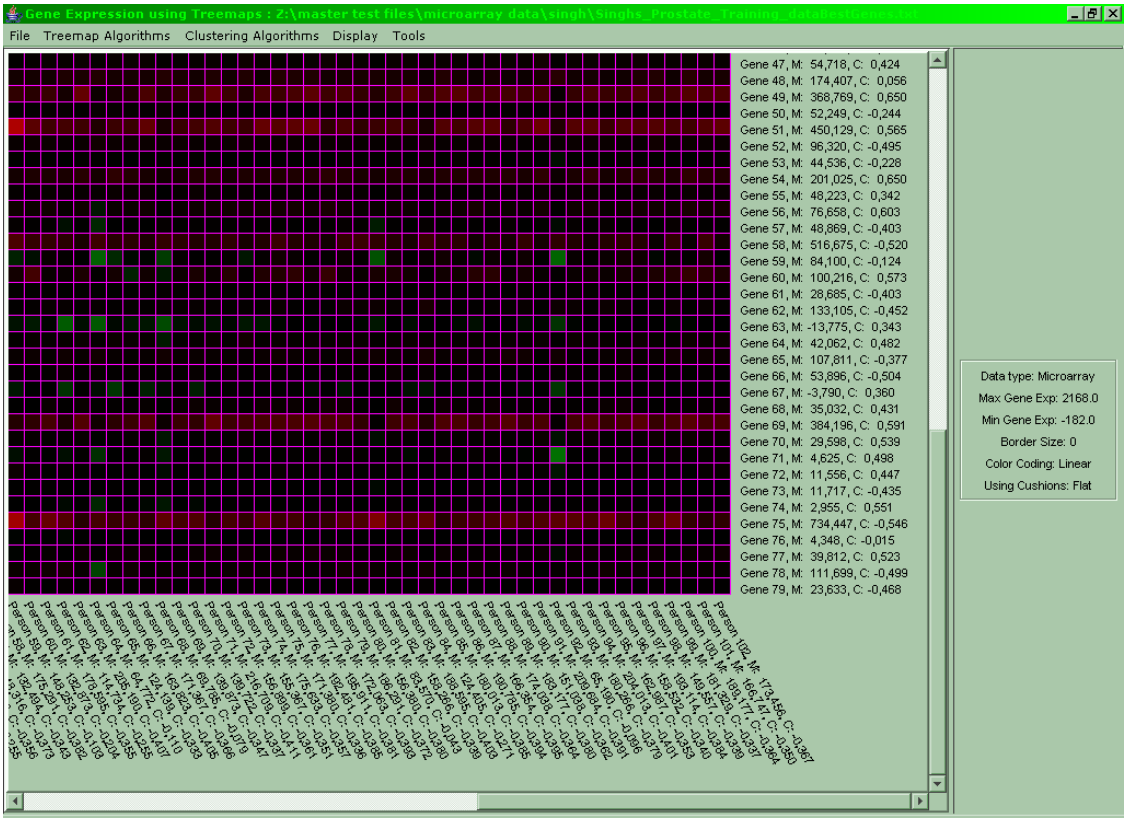


Figure A.35: To the bottom and the right of the microarray the names of the patients / genes are displayed as well as their respective mean values and correlation coefficients

The user may at this point use only the menu items **`Color Coding`**…, **`Border Size`**… and **`Cushion Treemaps`** from the **`Display`** menu in order to alter the appearance of the microarray. Everything else in the **`Display`** and **`Tools`** menu is deactivated, also a right click will not have any effect.

But now a new menu item becomes active: The **`Clustering Algorithms`** menu's first item, **`Perform Clusterings`**… is enabled and by selecting it the following dialog appears:

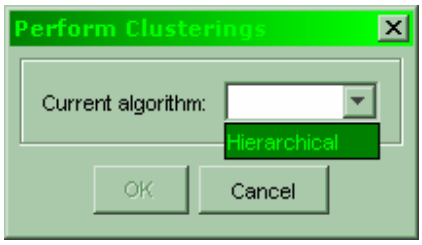


Figure A.36: The dialog for the Perform Clusterings… menu item

Once the user selects the hierarchical algorithm, its respective parameters appear:

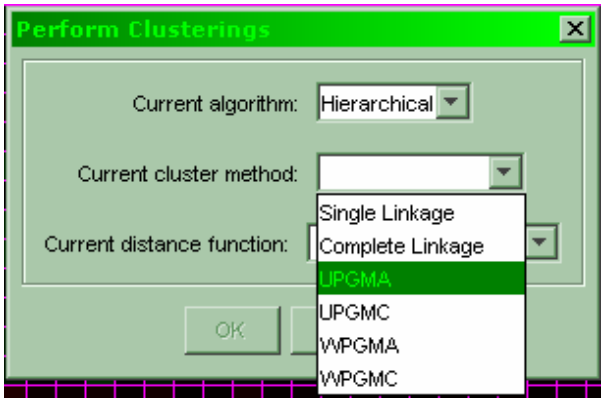


Figure A.37: The cluster method selection for the clustering dialog

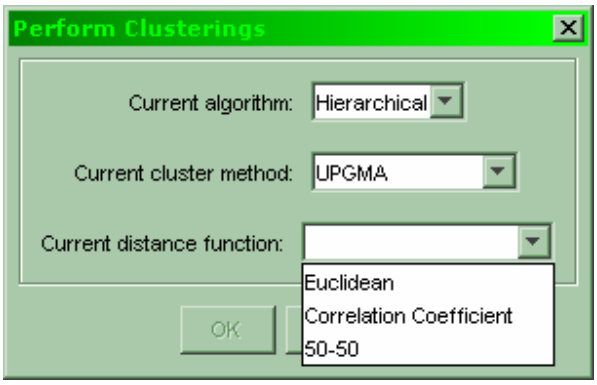


Figure A.38: The distance function selection for the clusterings dialog

Once all parameter slots are filled, the ok button becomes enabled and allows the user to perform the clustering. After that two progress windows will appear which monitor the progress of the clustering, each window shows one clustering:

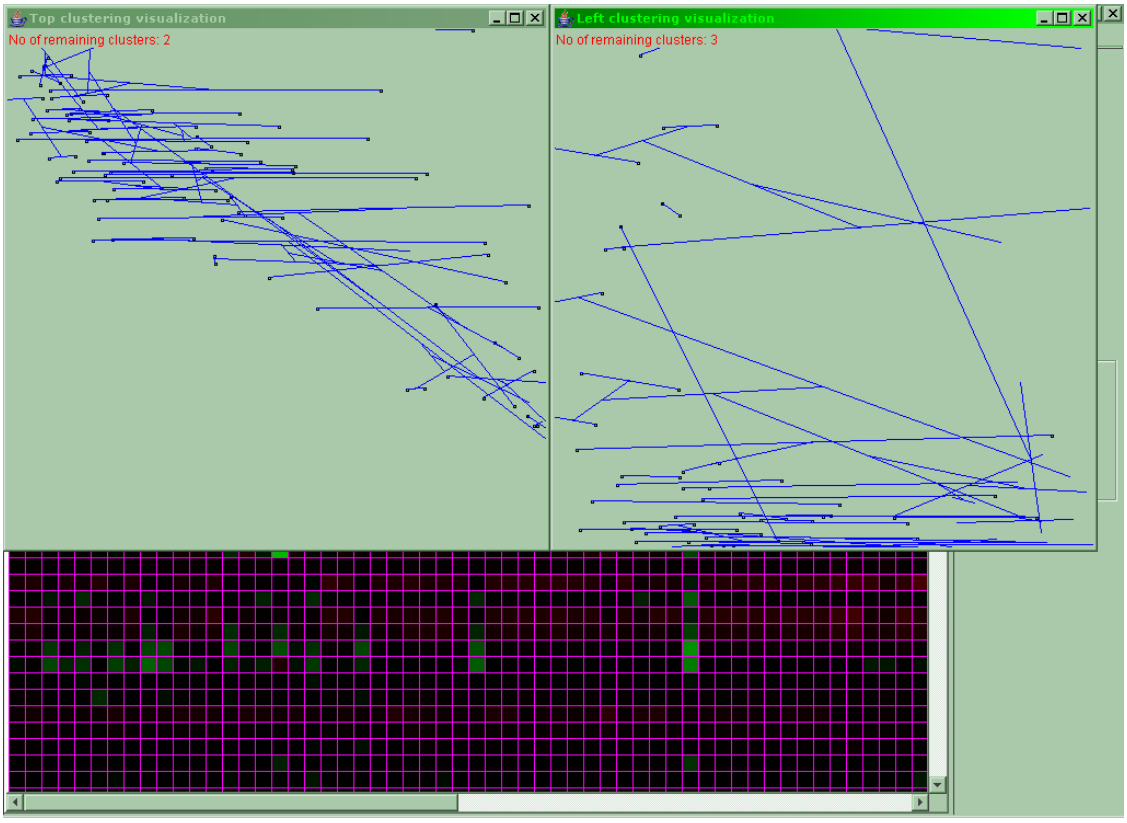


Figure A.39: The monitoring windows which display the progress of the visualization

These are a rough visualization of the clustering progress by symbolizing every set of gene expressions to be clustered (either through their mean value, or their correlation coefficient, or a 50-50 between the two) with a small square on screen. Every set which is clustered with another receives a blue link to its peer. This blue line symbolizes the cluster joining these two nodes. Should a cluster be involved with another cluster instead of a node then the blue line starts at the middle of another blue line etc. The x-axes of these visualizations represent the mean values and the y-axes the correlation coefficients. In the top left corner of every display the user may read the number of clusters remaining to be clustered. When this counter drops to two the clustering is complete since for two clusters the join does not require the algorithm's calculations. It is again noted that these are rough visualizations with a main purpose to display the progress of the clustering and provide a visual aid as to when the clustering process will be complete.

After the clustering is complete the visualization displays disappear and the clustered microarray can be seen:

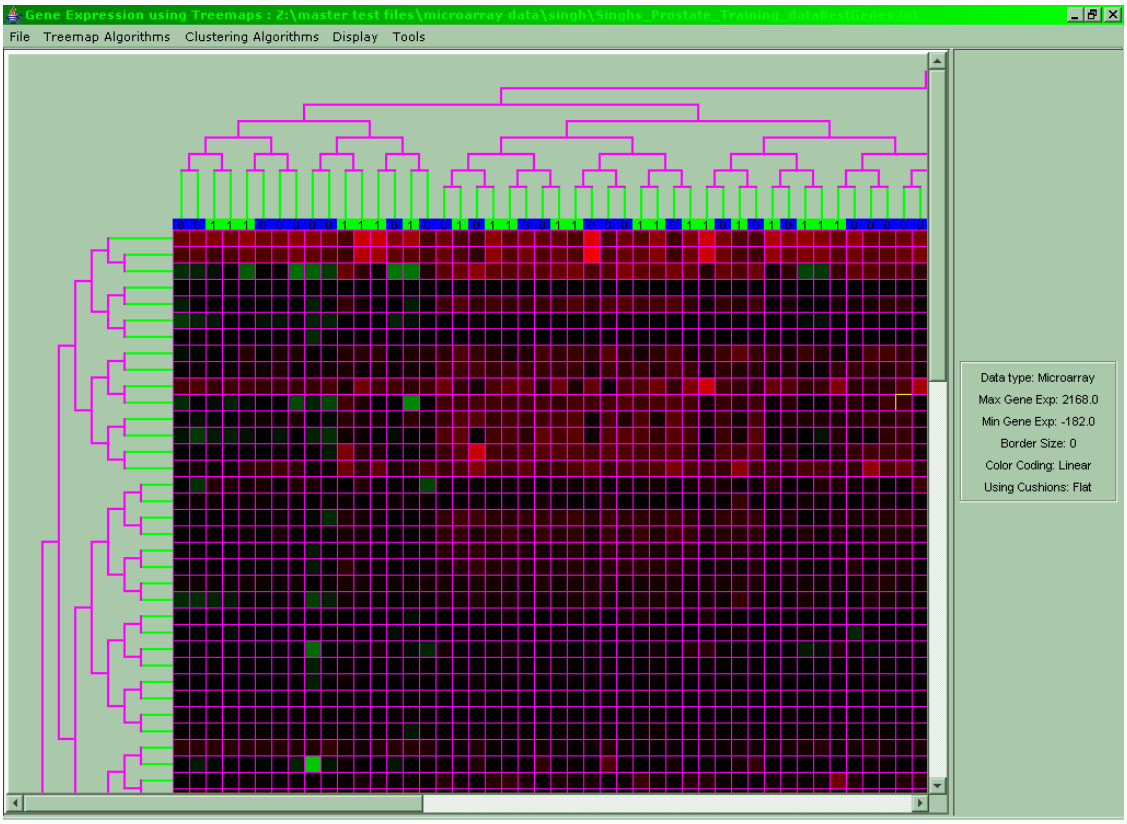


Figure A.40: The microarray after it has been clustered

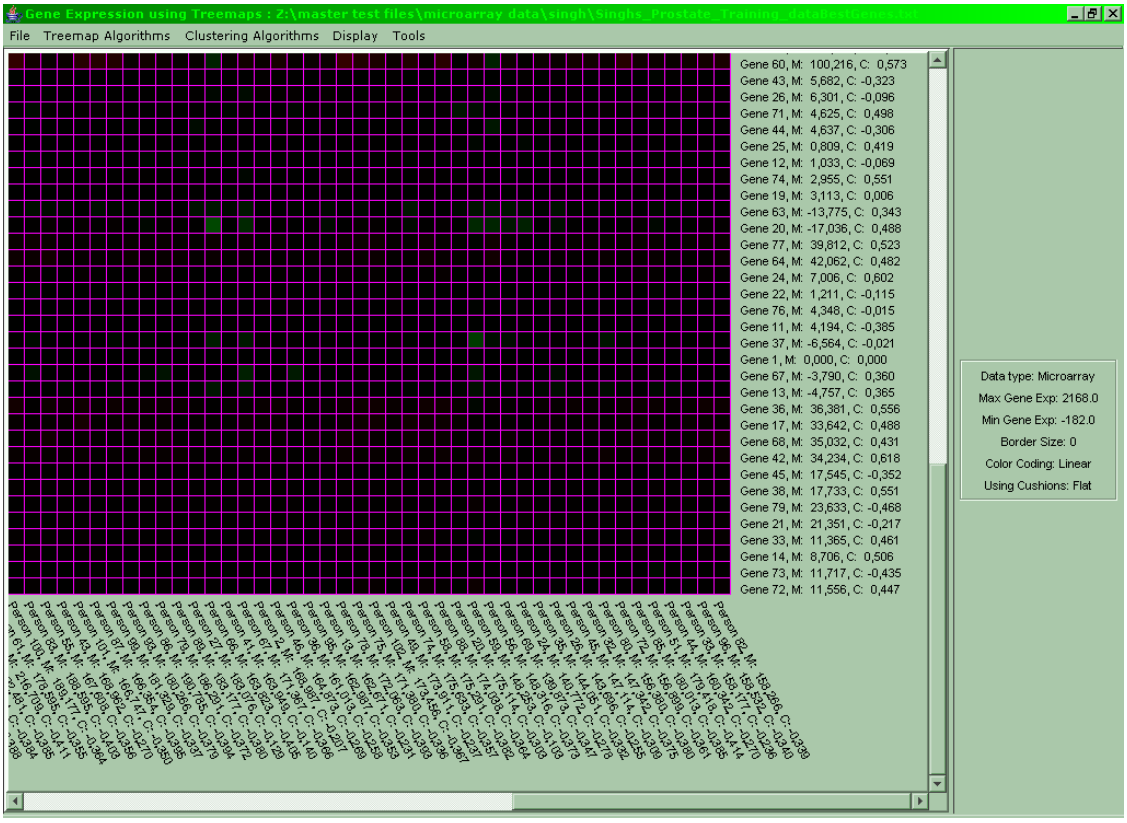


Figure A.41: The order of the genes and patients has obviously changed and siblings are close with regard to the clustering metric used (mean value, correlation coefficient or 50-50) although this may not always be the case depending on the cluster method used (Single Linkage, Complete Linkage, etc.)

Note that in order to return to the unclustered data a reloading of the file will be necessary.

At this point the other menu item of the **`Clustering Algorithms`** menu becomes enabled and allows the user to **`Display Clusterings as Treemaps`**:

Figure A.42: The previous cluster trees displayed as treemaps

By displaying the treemaps as trees can the similarity with the previous clustering trees be seen more clearly:

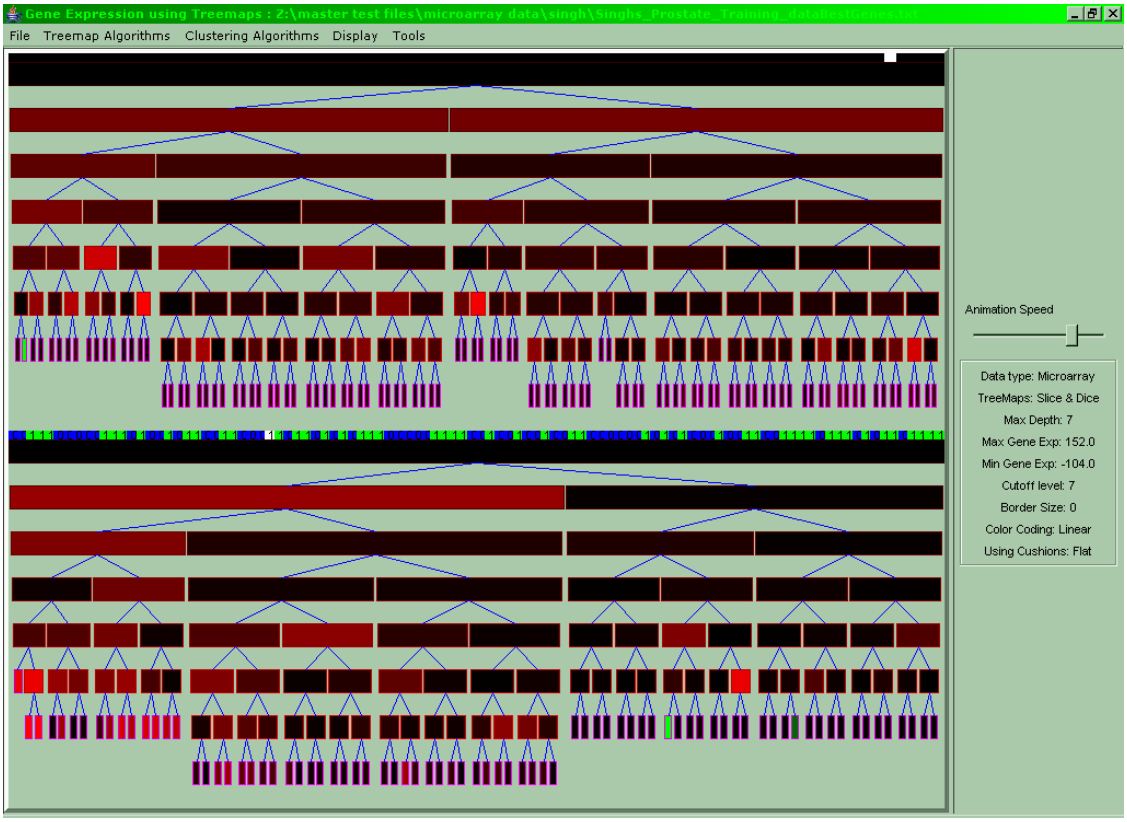


Figure A.43: The clustering trees shown as standard trees

The slider to the right controls how fast the animation will proceed. Setting it to the far left means that the animation is paused.

Whilst in this animating treemap mode the user has all menu items of the `Display` and `Tools` menu at his disposal except for the `Tools` menu item `Redesign to Tree Center`. Generally the changing of the root is something without meaning in the context of clustering trees so functions with that aim are deactivated. This holds also for the right click menu of standard and phylogenetic trees. Instead, another right click menu is provided:

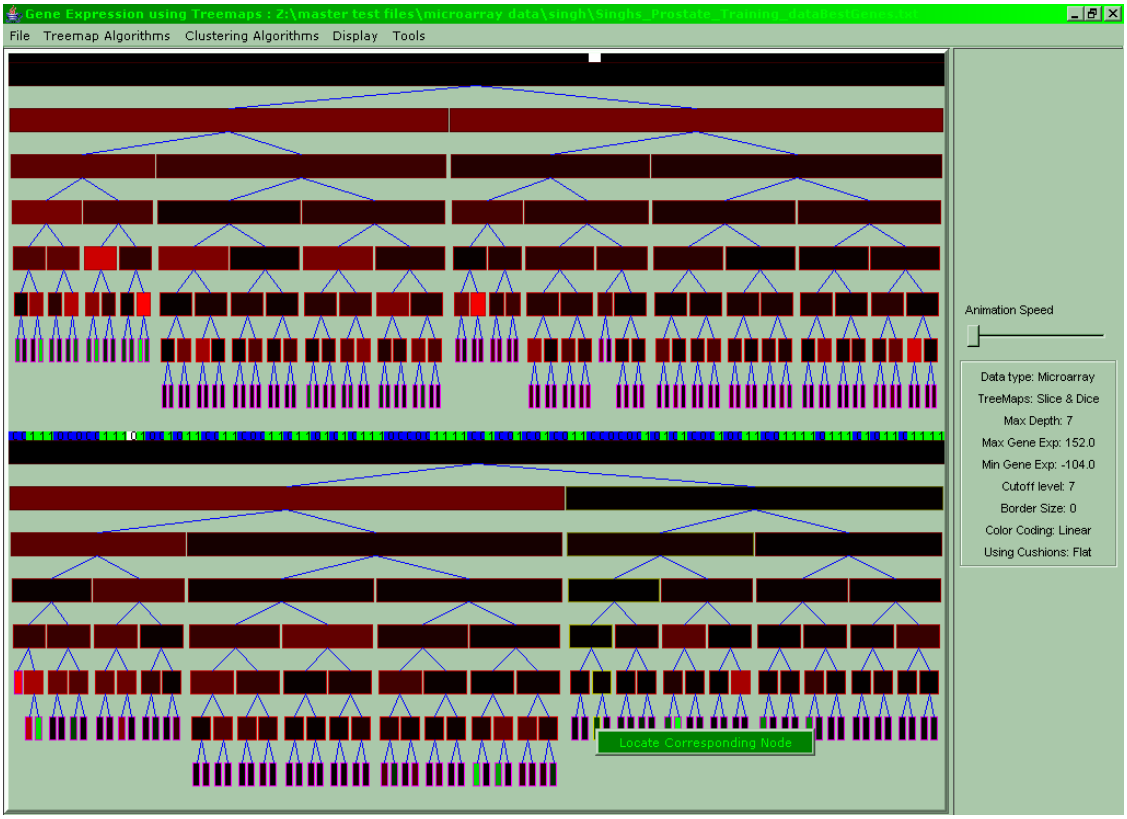


Figure A.44: The right click menu whilst in animating treemap mode contains only one item

The only item of that menu is named **`Locate Corresponding Node`**. When this menu is activated it pauses the animation and locates the gene expression selected in the other treemap:

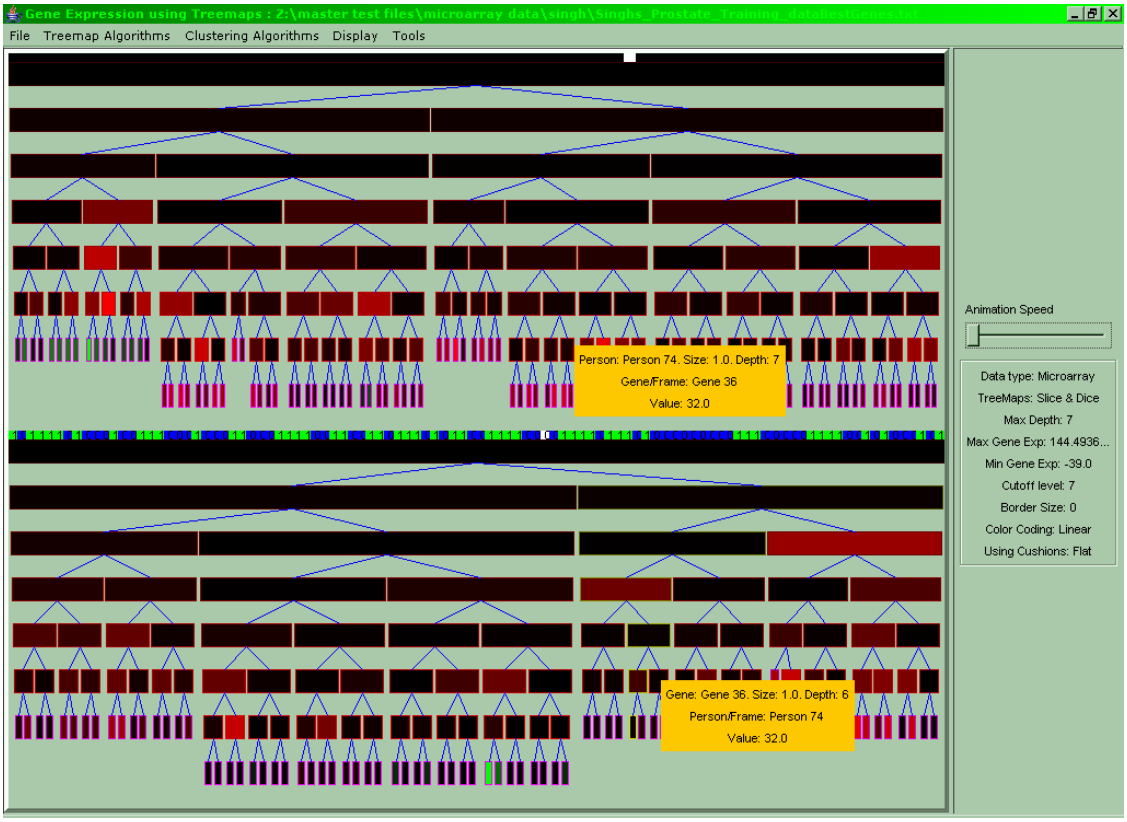


Figure A.45: The corresponding gene expression in the other treemap is located

Note that while the menu is displayed when clicked on non-leaf nodes there is no effect except for the pause in the treemap animation because clusters do not have the property of having a corresponding clone in the other treemap.

In order to escape from this animation lock the user should right click on any node in either treemap and select once again the menu **`Locate Corresponding Node`**. Then the two popup windows disappear and the user may alter the animation speed to commence the animation.

There exists also the function of exporting the animating treemaps in newick format. The user selects **`File -> Export Newick File`**... and a save dialog appears. If the user saves to a file named *name* the software creates *number of genes* + *number of patients* files named *nameGene 1.nw, nameGene 2.nw, …, nameGene g.nw, namePatient 1.nw, , namePatient 2.nw „…,namePatient p.nw*. Note that due to the coloring difference between microarrays and newick/standard tree files it is possible for the software to regard some newly created *nw* file as a newick file and thus display it using green as

positive and red as negative values. A newick file will be colored this way only if it has not a single negative value.

Finally a function which works for phylogenetic and standard files as well as for animating treemaps is that of displaying a list which shows all leaves originating from a selected node:

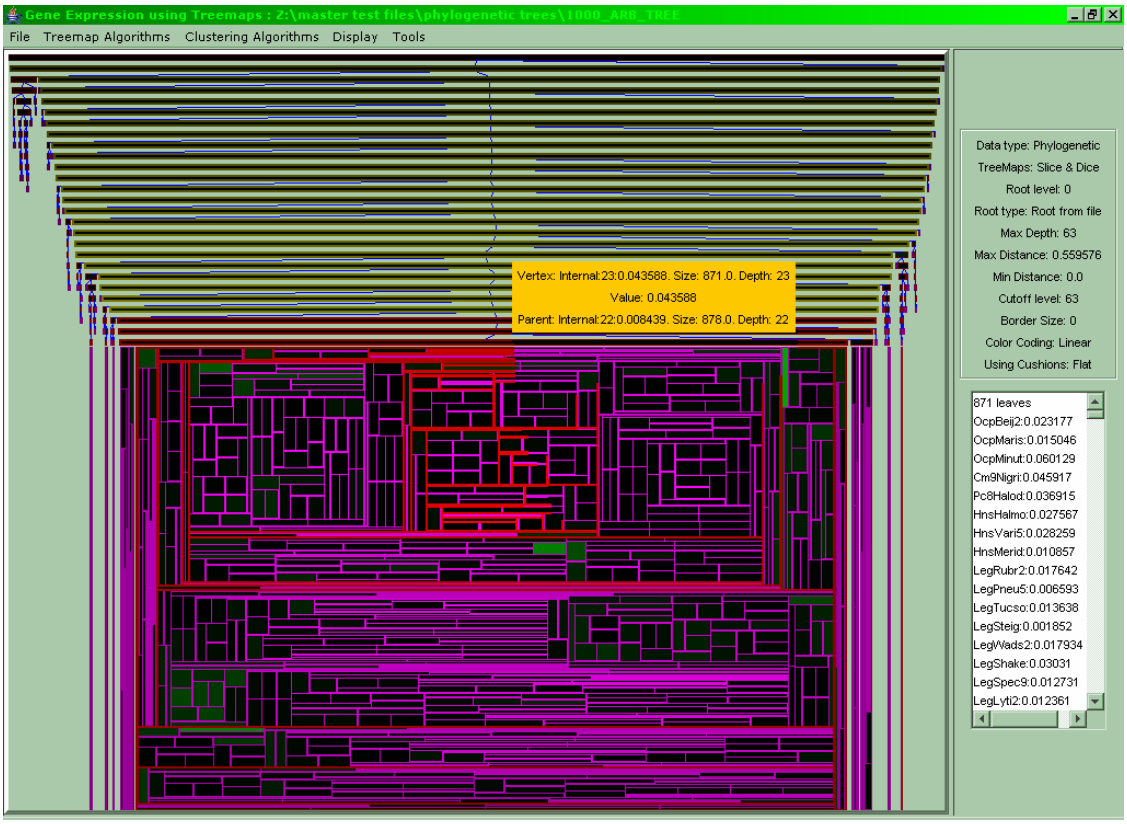


Figure A.46: If the user clicks with the middle mouse button on any node, the list on the right appears which lists every leaf originating from the selected node

Furthermore when the user selects a node from the list on the right the selected node is highlighted with a border and a popup window with its info:

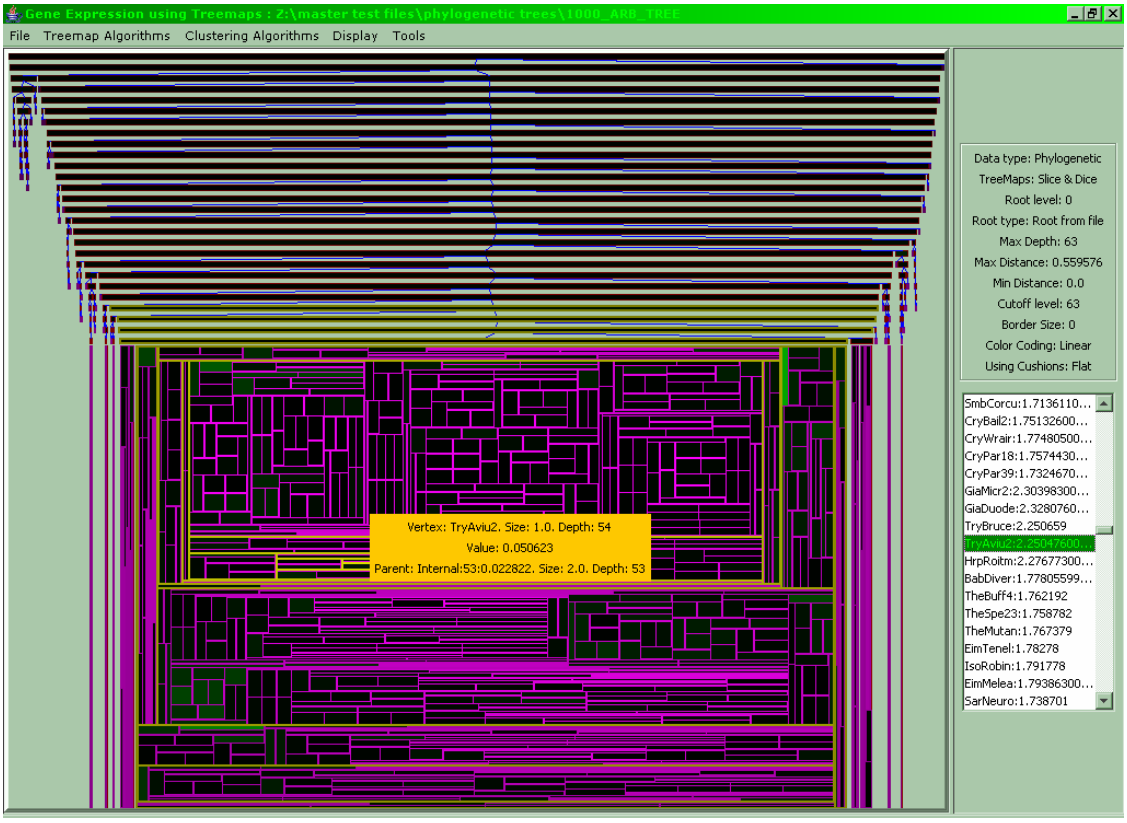


Figure A.47: Whenever the user selects a node from the list on the right the corresponding node is located, selected and information about it displayed

Besides loading a phylogenetic file and viewing the tree with its distances, the user may also load a file which contains the orders of the leaf nodes/species of the phylogenetic tree.

Whenever the user loads a phylogenetic tree a dialog appears and asks whether an order file should be loaded as well:

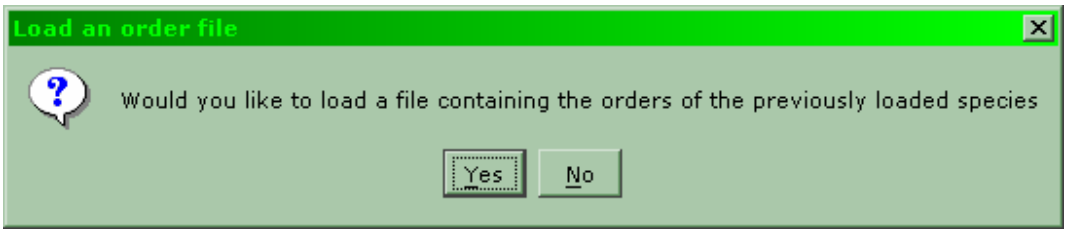


Figure A.48: The dialog asking for the order file

If the order file is valid, meaning that every leaf node in the newick file appears exactly once in the order file, then the phylogenetic file appears normally:

Figure A.49: Initially the orders of the species are not visible, only the popup window shows the order of the currently selected node

By selecting the menu `Display -> Show Orders...` the below dialog appears:

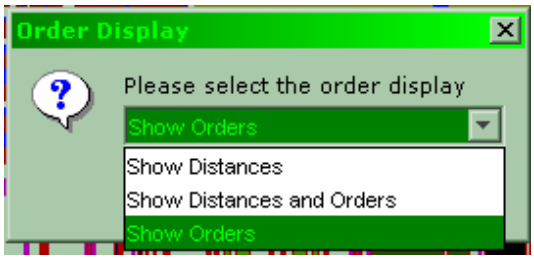


Figure A.50: The dialog for displaying the orders of the species

The user may choose to have the nodes colored according to either their distance as can be seen in figure A.49 above, or according to their order (by selecting `Show Orders` in the dialog from figure A.50):

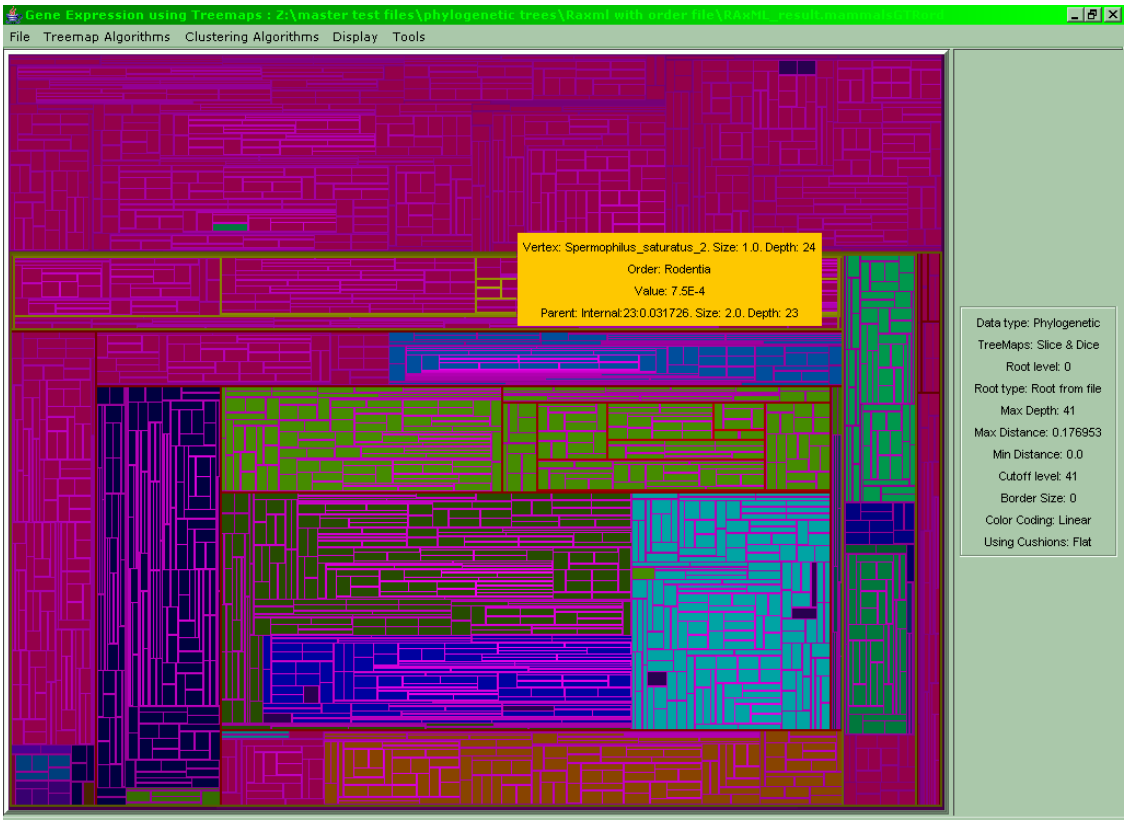


Figure A.51: The color coding has changed to reflect the orders of the species

The user may also combine distance and order for every node where the left half of the node displays the distance and the right half the order (choice **`Show Distances & Orders`** from the dialog shown in figure A.50):

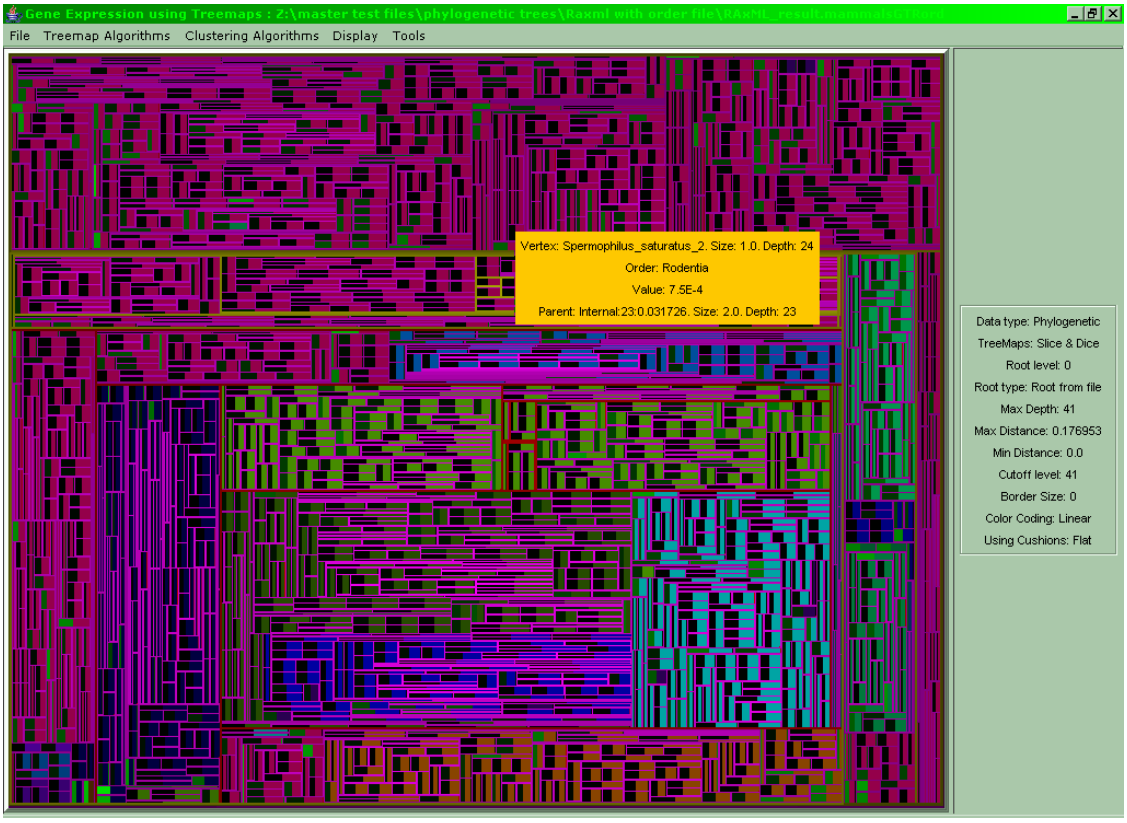


Figure A.52: The left part of each node corresponds to its distance from its parent while the right part is the node's order

While the order file states only leaf nodes, the software calculates whether an internal node's children are all of the same order and, if so, colors it accordingly. This is done by performing a DFS through the tree and during the return of the last child of an internal node to its parent checking if this child and every one of its siblings are of the same order. Below is a screenshot of the phylogenetic tree from figure A.50 with its internal structure shown:

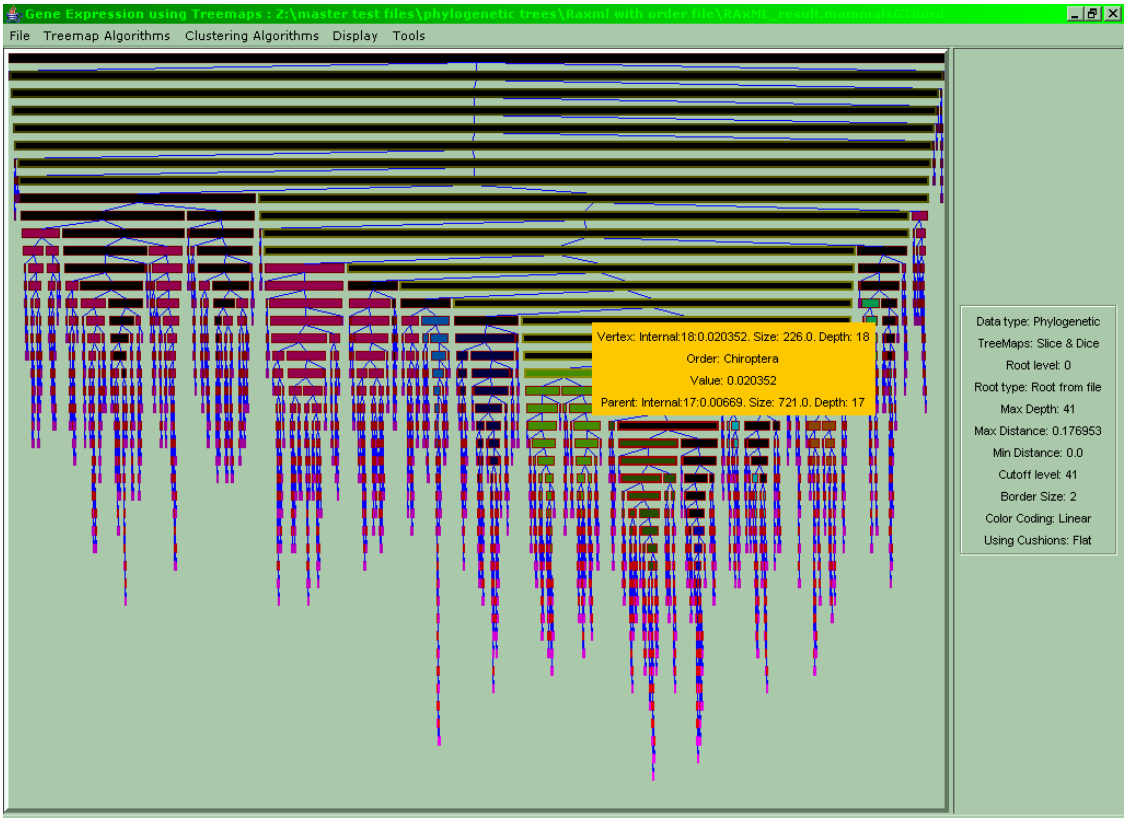


Figure A.52: It can be seen that several internal nodes were assigned orders based on their children's orders