in Proceedings of 13th Conference on Visualization and Data Analysis (VDA 2006), San Jose, California, Jan. 2006.

Trees in a Treemap: Visualizing multiple Hierarchies

Michael Burch and Stephan Diehl

Computer Science, Catholic University Eichstätt, 85072 Eichstätt, Germany

ABSTRACT

This paper deals with the visual representation of a particular kind of structured data: trees where each node is associated with an object (leaf node) of a taxonomy. We introduce a new visualization technique that we call *Trees In A Treemap*. In this visualization edges can either be drawn as straight or orthogonal edges. We compare our technique with several known techniques. To demonstrate the usability of our visualization techniques, we apply them to find interesting patterns in decision trees and network routing data.

Keywords: Information visualization, taxonomy, tree diagrams

1. INTRODUCTION

In graph theory a tree is a connected, directed graph which has exactly one node that has no incoming edges and all other nodes have exactly one incoming edge. The former node is called the root of the tree. Trees are typically drawn as node and edge diagrams, that we call tree diagrams in the rest of this paper to distinguish the visual representation from the underlying mathematical structure.

Taxonomies are a common and powerful tool to structure information. Taxonomies are special trees where each leaf node represents some object and all other nodes represent classifications of objects represented by its child nodes.

For example, in biology there are taxonomies of species: Sparky is a dog, dogs are mammals, mammals are animals and animals are living things. In commerce, we have taxonomies of products: butter is a dairy product, dairy products are food. In operating systems files are contained in directories, that can be contained in other directories and so on. In programming languages like `JAVA` language constructs like expressions, statements, methods, classes, and packages form a taxonomy, and also the type system of the language can be represented as a taxonomy. Finally, the internet IP addresses are based on a taxonomy of networks and their subnetworks.

In many applications information can be structured using two trees. One tree is a taxonomy of some objects, and the other is a tree where each node is associated to one of the objects in the taxonomy. Thus, several nodes can be associated to the same object. We call such a tree an object tree. The challenge is to draw object trees such that for each object its position in the taxonomy is easily visible.

The rest of this paper is organized as follows. In Section 2 we briefly describe several techniques for visualizing object trees with taxonomies and compare these techniques in Section 3. The algorithms underlying our *Trees-in-a-TreeMap* visualization are presented in Section 4 followed by two case studies in Sections 5 and 6. Finally, in Section 7 we briefly discuss related work, and Section 8 concludes this paper.

2. VISUALIZING OBJECT TREES WITH TAXONOMIES

The simplest way to show an object tree and the underlying taxonomy is by drawing the tree diagrams of both of them next to each other, see Figure 1. In this case it is not immediately obvious how often and where an object of the taxonomy occurs in the object tree. To make the associations explicit, we can draw links between each object of the taxonomy to all of the nodes it is associated with in the object tree, as shown in Figure 2.

Instead of showing the taxonomy as a separate tree diagram, we can also use color coding to integrate it into the object tree. Before assigning different colors to the objects of the taxonomy, we first compute a leaf word of the taxonomy. Let

Further author information: (Send correspondence to S.D.)
M.B.: E-mail: michael.burch@ku-eichstaett.de
S.D.: E-mail: diehl@acm.org

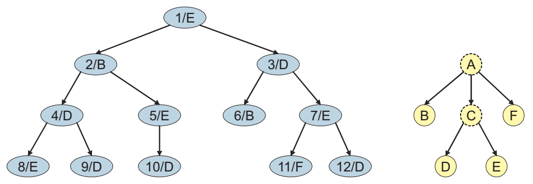


Figure 1. Two separate tree diagrams: object tree and taxonomy

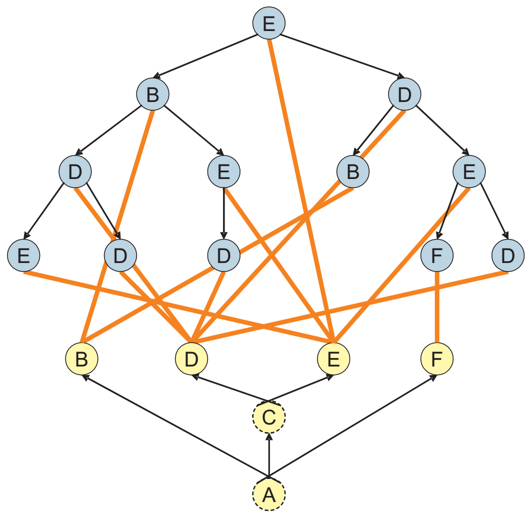


Figure 2. Linked tree diagrams of the object tree and the taxonomy

t be a tree, then the leaf word $\mathsf{lw}(t)$ is equal to t, if t is a leaf node, otherwise it is the concatenation of $\mathsf{lw}(t_1), \ldots, \mathsf{lw}(t_k)$ where $t_1, \ldots, t_k$ are the subtrees of t. For the taxonomy in Figure 1 the words `BDEF`, `FEDB`, `DEBF` are leaf words, but for example `EBFD` is not a leaf word. We use the leaf word to flatten the taxonomy. In other words, we derive a total order from a partial order and thus lose some information. Given a leaf word of a taxonomy, we can now assign the subsequent colors of a color scale to the leaf nodes according to the order of their occurrences in the leaf word. As a result, objects that are close to one another in the taxonomy get a similar color, as shown in Figure 3.

Adjacency matrices can be used as a space-filling visualization of graphs by representing every entry of the matrix as a single pixel. Admittedly, for trees the space efficiency is not as good as for general graphs, as there are many empty entries in the matrices. Figure 5 shows the adjacency matrix of the object tree both with unsorted dimensions[*] and with sorted dimensions: the dimensions are sorted according to a leaf word of the taxonomy and as a result rows and columns of objects that are closely together in the taxonomy are also closely together in the matrix.

Parallel coordinate views[1] can also be used to show object trees. Again we can sort the objects according to a leaf word of the taxonomy to keep certain objects closely together, as shown in Figure 6. In contrast to the classical way of drawing parallel coordinates views, we did not represent several edges between two objects at the same level by a single edge.

In previous work we have used both sorted matrices (that we called pixelmaps) as well as sorted parallel coordinates to detect clusters and outliers.[2]

As a new approach to integrating both the taxonomy and the object tree we propose to draw trees in a treemap. The treemap shows the taxonomy and the nodes of the object tree are drawn in the boxes representing the objects associated with each node. Thus for every node of the object tree its position in the taxonomy is easily visible. In addition, the treemap can also be used to hide details of the object tree by folding parts of the taxonomy. Figures 7 and Figures 8 show two different ways of drawing a tree in a treemap. In both figures the treemap is unfolded down to the level of objects.

[*]Actually here the objects are sorted by a breadth-first traversal

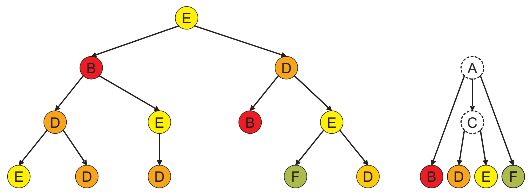


Figure 3. Colored tree diagrams

Visualization technique	Single representation	Crossings	Continuity	Clusters Outliers	Compact	Taxonomy
Separate tree diagrams	no	no	straight	no	no	tree diagram
Linked tree diagrams	no	yes	straight	difficult	no	tree diagram
Colored tree diagrams	no	no	straight	yes	medium	color (leaf word)
Unsorted matrix	yes (r.+c.)	no	not visible	no	high	not visible
Sorted matrix	yes (r.+c.)	no	not visible	yes	high	order (leaf word)
Sorted par. coord.	yes (row)	yes	straight	yes	medium	order (leaf word)
Trees in a TM (straight)	yes	yes	straight	yes	high	treemap
Trees in a TM (orth.)	yes	reduced	orthogonal	yes	high	treemap

Figure 4. Comparison of visualization techniques

In the treemap in Figure 7 each node of the object tree is represented by a small circle placed in the box that represents the object associated with the node. Edges are drawn as straight lines connecting these circles. The root of the tree is indicated by a bigger green circle. It is also possible to draw several object trees in the same treemap.

In the treemap in Figure 8 the nodes are connected by orthogonal lines, i.e., horizontal and vertical lines with bends of 90 degrees.

In Section 4 we describe the implementation of our *Trees-in-a-Treemap* visualization and the underlying algorithms. In particular, the algorithm for orthogonal trees tries to both reduce the number of edge crossings and the length of the edges – two conflicting goals.

3. A COMPARISON

To compare the different visualization techniques presented above we use the following criteria:

Single representation: Is there a single representation of each object? If the taxonomy is shown, is this single representation also used by the taxonomy?

Crossings: Are there edge crossings in the drawing of the object tree?

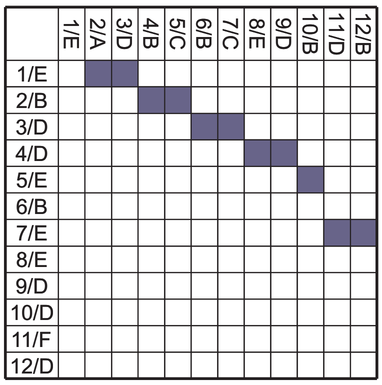
Continuity: Is it easy for the human eye to follow the lines of the edges? For orthogonal layout we tend to have longer edges, but less crossings and these crossings have a right angle. For straight edges the angle of the crossings can become very small and it can become very difficult to follow the edge.

Clusters/outliers: Does the visualization allow to detect clusters and outliers?

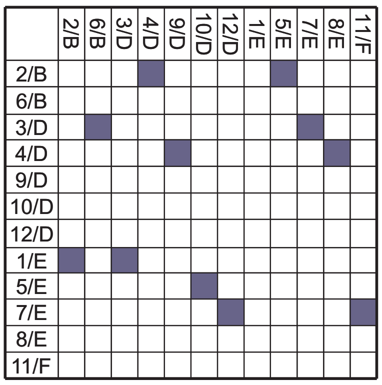
Compactness: Is the visualization compact or space-filling?

Taxonomy: Is the taxonomy shown, how is it shown, and to what extent? Some of the techniques compute a total order to approximate the taxonomy based on a leaf word.

The results of our comparison are summarized in Figure 4. It is noteworthy, that only the *Trees-in-a-Treemap* visualizations provide a single representation of each object while at the same time showing the full taxonomy. In addition, the orthogonal layout only leads to right-angled edge crossings and thus improves the continuity of the edges.



	1/E	2/A	3/D	4/B	5/C	6/B	7/C	8/E	9/D	10/B	11/D	12/B
1/E		■	■									
2/B				■	■							
3/D						■	■					
4/D								■	■			
5/E										■		
6/B												
7/E											■	■
8/E												
9/D												
10/D												
11/F												
12/D												



	2/B	6/B	3/D	4/D	9/D	10/D	12/D	1/E	5/E	7/E	8/E	11/F
2/B				■					■			
6/B												
3/D		■								■		
4/D					■						■	
9/D												
10/D												
12/D												
1/E	■		■									
5/E						■						
7/E							■					■
8/E												
11/F												

Figure 5. Adjacency matrices with unsorted and sorted dimensions

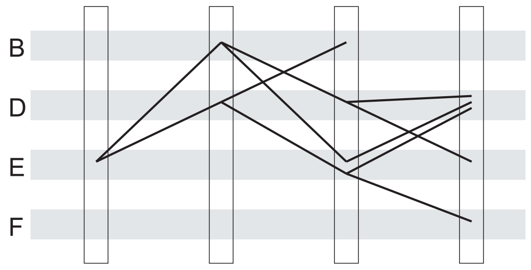


Figure 6. Sorted Parallel Coordinates

4. LAYOUT OF TREES IN A TREEMAP

Each node, be it the root, an inner node, or a leaf node of the tree can be represented by a treemap box. The boxes of child nodes are always contained in the box of their parent node and do not overlap. The size of each box can be adjusted to a given metric, e.g. in software repositories the size of a file, the lines of code or the number of changes. Even the color of each box could be used to represent another metric, e.g. the age of a file or the person, who did the last commit. But such a color coding would be very difficult, because the edges and nodes already use color coding. This could lead to the problem that the edges and nodes of the trees disappear in some boxes and in others not. So the visualization would be even worse to read for the user. On the other hand the color coding can be either on the boxes of the tree map or on the trees in the treemap. The user can interactively choose the way of color coding he prefers in a given situation.

To draw an object tree in the treemap it is not sufficient to only determine for each node which box represents its associated object, but one has also to decide where to place the node within this box. In our implementation we use three different approaches to compute the placement of the object tree nodes. We discuss the three methods as well as their advantages and disadvantages in more detail below and compare them.

4.1. All nodes randomly

The naive approach to compute a layout for the object trees in the treemap is to randomly choose the position of each node in its corresponding box. This generates a very chaotic layout, but it also has two big advantages: First, it works very fast and, second, the resulting edges do not overlap in most cases. But on the other side the angles between the edges become sometimes very small and thus the human eye can not easily follow these edges. This makes the resulting visualization sometimes very difficult to use.

4.2. All roots randomly, all other nodes centered in the box

Another approach is to only place the roots randomly. All other nodes are positioned in the center of each corresponding box. This algorithm is also very fast and avoids the chaotic layout of the first approach. But as a downside there are lots of edges which overlap. This layout is absolutely sufficient if one is only interested in detecting if there exist relations at all between some nodes. In this respect this approach can be compared to the one used by parallel coordinates views.

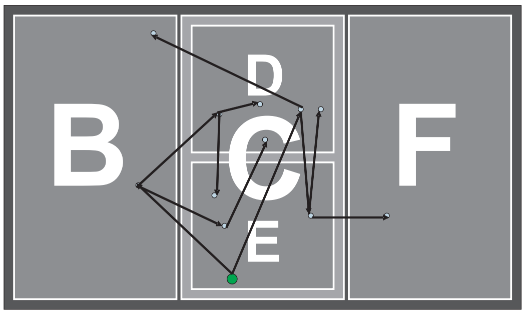


Figure 7. Trees in a Treemap: edges are drawn as straight lines

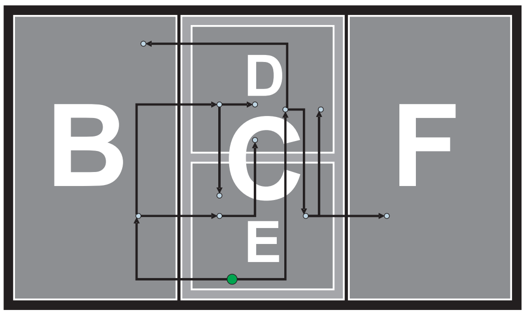


Figure 8. Trees in a Treemap: edges are drawn as orthogonal lines

4.3. Orthogonal layout with respect to a minimal edge crossing number

One advantage of orthogonal layout of object trees is that all edge crossings have an angle of exactly 90 degrees and so can better be followed. Crossing lines with acute angles have the disadvantage that their trace can be lost very easy. Furthermore all nodes have a different position and so can not be hidden from other nodes and even the edges do not cross a node. This means we need more space than in the approach given in section 4.2 but fortunately all edges and all nodes are always present.

Additionally, in our approach the number of edge crossings should be as small as possible. This makes the graphics more readable. Unfortunately, computing a layout with a minimum number of edge crossings is an algorithmic problem known to be *NP*-complete.[3] So we have to restrict our approach in a way that bounds the number of edge direction changes. Trivially in this orthogonal layout there are only four possible directions for an edge: Left, right, up and down. A direction change means that one edge takes a different direction from the one it had before. To avoid cycles we omit the contrary direction. This means if an edge has direction left, only directions up and down are possible ones and right will be forbidden.

The number of such direction changes is to blame for the large search space of possible candidates for an optimal edge. So bounding this number makes the computation much faster even for a very high number of edges. Note, that two different boxes can always be reached by an orthogonal edge which changes at most once its direction. Furthermore, one has to keep in mind, that the more edge direction changes there are, the more difficult it becomes for the human eye to follow the edge. Some tests resulted in an edge direction change number of 2 or 3. So the best possible layout is the one which

- has a very low number of edge direction changes
- has a nearly minimal edge crossing number and
- avoids very long edges

Even an optimal position of the treemap boxes in each hierarchy level can be computed to minimize the edge crossing number and edge lengths. But this is also a very difficult task to do. This problem is known as the Optimal Linear

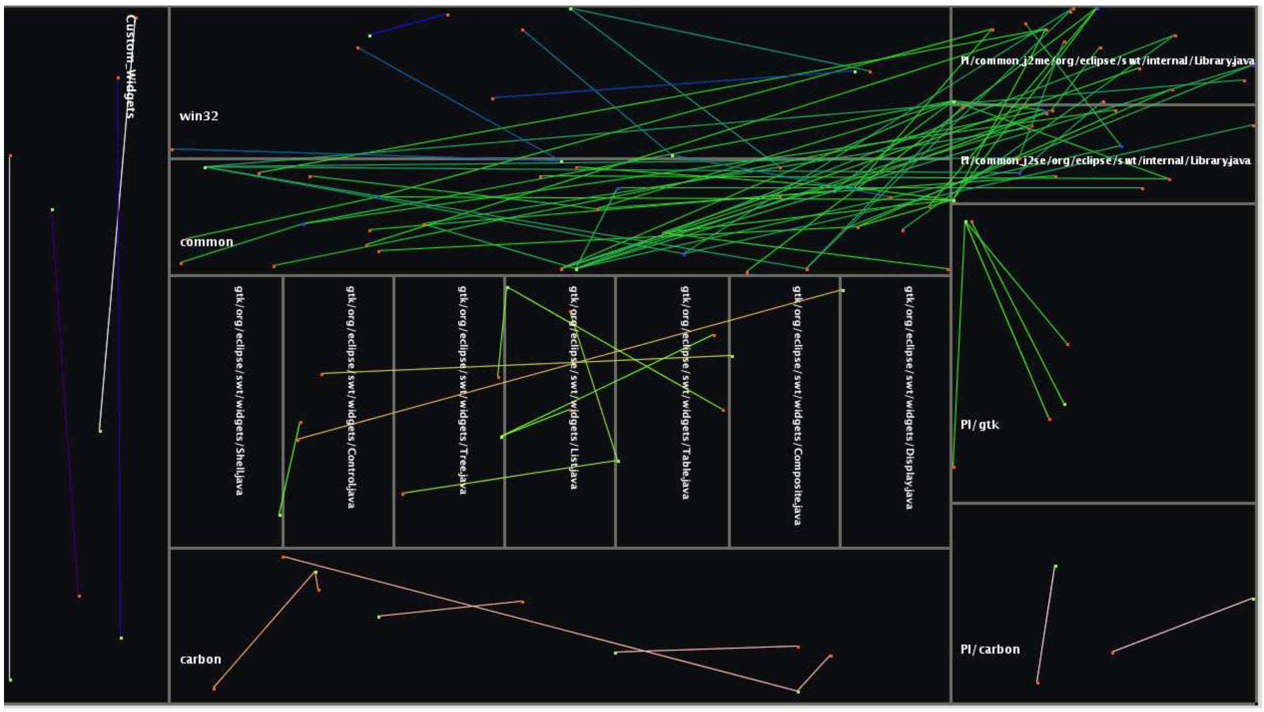

Custom_Widgets
win32
common
gtk/org/eclipse/swt/widgets/Shell.java
gtk/org/eclipse/swt/widgets/Control.java
gtk/org/eclipse/swt/widgets/Tree.java
gtk/org/eclipse/swt/widgets/List.java
gtk/org/eclipse/swt/widgets/Table.java
gtk/org/eclipse/swt/widgets/Composite.java
gtk/org/eclipse/swt/widgets/Display.java
carbon
PI/common_j2me/org/eclipse/swt/internal/Library.java
PI/common_j2se/org/eclipse/swt/internal/Library.java
PI/gtk
PI/carbon

Figure 9. SWT decision tree with fully expanded GTK directory

Arrangement(OLA) Problem[3] and proven to belong to the class of *NP*-complete problems. As long as we are dealing with small treemaps with only a few items in each hierarchy level, this is not a problem at all. But as the problem instance of this optimization problem gets bigger we have to use a heuristical approach and compute a nearly optimal solution. To get more precise we will explain where the OLA problem appears in the lets say optimal sorting of the treemap.

Let

$$coupling : Box \times Box \longrightarrow \mathbb{N}$$

be a function which maps each pair of boxes to the number of couplings these two boxes have in the given dataset. Keep in mind that each box corresponds to a special item in the rule set.

Boxes $b_1, \ldots, b_n$ in a hierarchy level are optimal linear arranged with permutation π, if and only if the permutation

$$\pi : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\}$$

of the boxes $b_1, \ldots, b_n$ minimizes

$$\sum_{i=1}^{n-1} coupling(b_i, b_{i+1}) \cdot \mid \pi(b_i) - \pi(b_{i+1}) \mid$$

We apply the optimal linear arrangement approach recursively in all hierarchy levels starting with the root which is trivially optimal arranged. After computing an optimal or in the case of a large item number a nearly optimal solution to the OLA problem we start to layout the trees in the treemap. A descrition to this layout algorithm is given in the following.

First, we need a good position for the root node of the object tree in its corresponding treemap box. Placing the root next to the box in which the next child node of the root has to be placed, makes the starting edge as short as possible. The starting edge is always precomputed by a straight orthogonal line from the starting node to the destination box, no matter how many edge crossings it has with already existing edges. This starting edge is taken for the currently optimal edge. The algorithm compares this one with all other generated edges which lead from the starting point to the destination box. If there is a better one which meets the former conditions it will replace the starting edge and now is the currently best one. The algorithm terminates when all possible candidates have been compared to the currently optimal edge.

The treemap is internally represented as a 2-dimensional matrix in which each matrix entry has a special property. This encoded property can be used to decide whether a pair of treemap coordinates has been used by a node or an edge. The algorithm needs this information all the time to decide if there can still be a better edge with less edge crossings, or if the currently computed edge cannot be improved. If the best possible edge has been found the treemap matrix will be updated and the algorithm tries to find out a layout for the next edge.

The algorithm works with a depth first search strategy. This keeps the memory requirement very small and even works faster than a breadth first search strategy. We have also tried a mixture of the former two strategies but there was no improvement. To draw an edge between two nodes, the corresponding boxes have to be computed first. The boxes restrict the coordinates where the nodes can be placed. Next, we compute one edge which has at most one direction change and which leads from the starting point to the destination box. In most cases this edge is not the optimal one. So the algorithm has to look for a better one, which means one with less edge crossings. Note, that there is no relation between the length of an edge and the number of crossings. In many cases we found that this relation was even reciprocally proportional: The longer the edge is the less edge crossings it has. This phenomenon makes the solution to this problem extremely difficult. This is the reason why we use depth first search in our algorithm. We search for longer edges to reduce the number of edge crossings. If there exist two edges with the same number of edge crossings the shorter will be chosen.

Since the treemap is divided into a two dimensional lattice which consists of finitely many points the space for drawing edges without overlapping each other will decrease very rapidly. Regions where many edges were located are mainly avoided by this algorithm because these are the main reason why the edge crossing number increases. So to avoid this problem the algorithm chooses longer edges with small edge direction changes. This fact causes additional running time of the algorithm.

4.4. Algorithm in Pseudo Code

<u>INSTANCE:</u>

- Trees $T_i = (V_i, E_i)$ and for each node in V_i the corresponding box bounds in the optimal linear arranged Treemap TM.
- A maximal number of edge direction changes c_{max}.
- A maximal edge length number l_{max}.
- A minimal edge length number l_{min}.

<u>SEARCHED:</u> A nearly optimal orthogonal layout of the trees T_i in the treemap TM.

```
for (int i=1;i<=n;i++) {
    Compute Root Position;
    for all edges do {
        compute orthogonal edge with minimal
        edge crossings and minimal length with
        respect to all already computed edges
        and with respect to all given parameters
    }
}
```

We want to improve our approach by giving parallel edges some distance. Each environment of an edge has a punishing function, which is stronger nearby the edge and lessened the more distance there is. So the path of an edge which is computed later has to use the route which has the smallest punishment.

Punishment functions are also used for

- **edges over a box bounding**
  Box bounds are important to distinguish different boxes from each other. An edge could hide a box bounding so that the user has a wrong interpretation of the treemap.

- **edges over a node**
  Nodes are absolutely necessary to see where an edge starts and where it ends. So these are not allowed to be overpaint by other edges.
- **edges over edges**
  Apart from crossing an edge we want to avoid two edges laying over each other. The best situation is when they have no point in common at all.
- **self overcrossing**
  This is also a problem because a self overcrossing edge could be recognized as a cycle. So we want to avoid cycles whenever possible.

5. CASE STUDY: SWT

To show the strengths of our visualization technique we explored parts of the Open Source software project `Eclipse SWT` (Standard Widget Toolkit). For this, we reconstructed for every version of each file in the software archive of `SWT` in which transaction it has been committed to the software archive.[4] After that, we applied a modified version of the AprioriAll algorithm[5] on the set of transactions. The result of this algorithm is a set of sequence rules of the form

$$a_1 \to \ldots \to a_m \Rightarrow c_1 \to \ldots \to c_n$$

Such a rule describes that when first a_1 is committed, followed by or together with a_2, followed by or together with $\ldots$, followed by or together with a_m, there is a particular probability (called the confidence of the rule) that later c_1, followed by or together with $\ldots$, followed by or together with c_n will also be committed. We call $a_1 \to \ldots \to a_m$ the antecedent and $c_1 \to \ldots \to c_n$ the consequent of the rule. By sharing common prefixes a set of sequence rules can be turned into a decision tree. Thus, in our case the object tree is a decision tree, the objects are the files and the taxonomy is given by the directory structure.

Figure 10 shows the root level of the *Trees-in-a-Treemap* visualization of the decision tree we obtained for `SWT`. As we can see there exist several green colored nodes – each representing the root of a decision tree. These roots are placed randomly within the box, while all other nodes of the trees are located in the center of the box. There is only one red colored node in the middle. Each single decision tree is drawn with a different color. In our visualization technique also the nodes show a color coding:

- green colored nodes always represent root nodes
- blue colored nodes represent nodes in the antecedent and
- red colored nodes represent nodes in the consequent

of the corresponding rule. So we can also detect if a node is the root of the tree, lies in the antecedent or in the consequent. But this is not possible with the approach described in section 4.1 as it is used in figure 10.

At the root level, the visualization does not convey much information about the rule structure itself, but we can see that there exist lots of trees which we can examine in more detail by expanding the root box in the treemap.

Figure 11 represents an intermediate level of the treemap, i.e., not all boxes have been fully expanded down to the level of files. As we can see, the subdirectories `_SWT_Custom_Widgets`, `win32`, `common`, `gtk`, `carbon`, and `_SWT_PI` are located right under the root level.

Here we use a layout which places each node randomly in its corresponding treemap box. As each tree is colored differently, it is possible to detect patterns in the rule set, e.g. clusters.

A closer look also reveals that there is no relation between the `_SWT_Custom_Widgets` subdirectory and any other directory. The same is true for the `carbon` and the `gtk` subdirectories.

Figure 9 gives an overview of the whole `gtk` subdirectory. As before, we can see that the `gtk` subdirectory has no relation to other directories. But in addition, the relations between the single files of the `gtk` subdirectory are shown. There

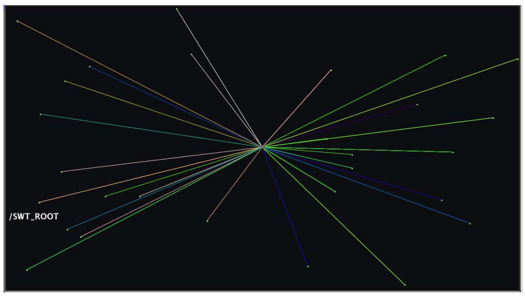


Figure 10. SWT decision tree in root directory

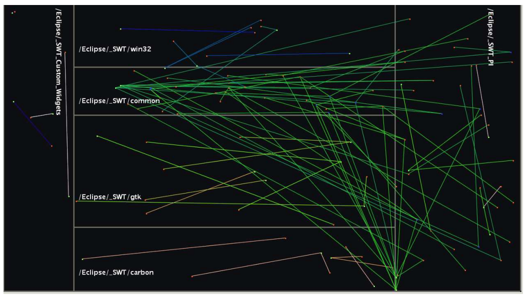


Figure 11. Intermediate expansion step of SWT decision tree

are also some cross-cutting relations between different parts of the directory tree. The `common` subdirectory is very closely related to files in the `_SWT_PI/common_j2se/org/swt/internal` subdirectory at the upper right corner of this figure.

Figure 12 shows the same rule set as Figure 9 but with an orthogonal layout of the decision tree which additionally avoids lots of edge crossings and generates parallel lines. The human eye can follow the tree edges much better if there are only angles of 90 degrees. In contrast to the previous example, in the orthogonal layout we can easily see that there is no relation between the `common_j2me` subdirectory and the `win32` subdirectory. This was very difficult to see in Figure 9.

6. CASE STUDY: TRACE ROUTES

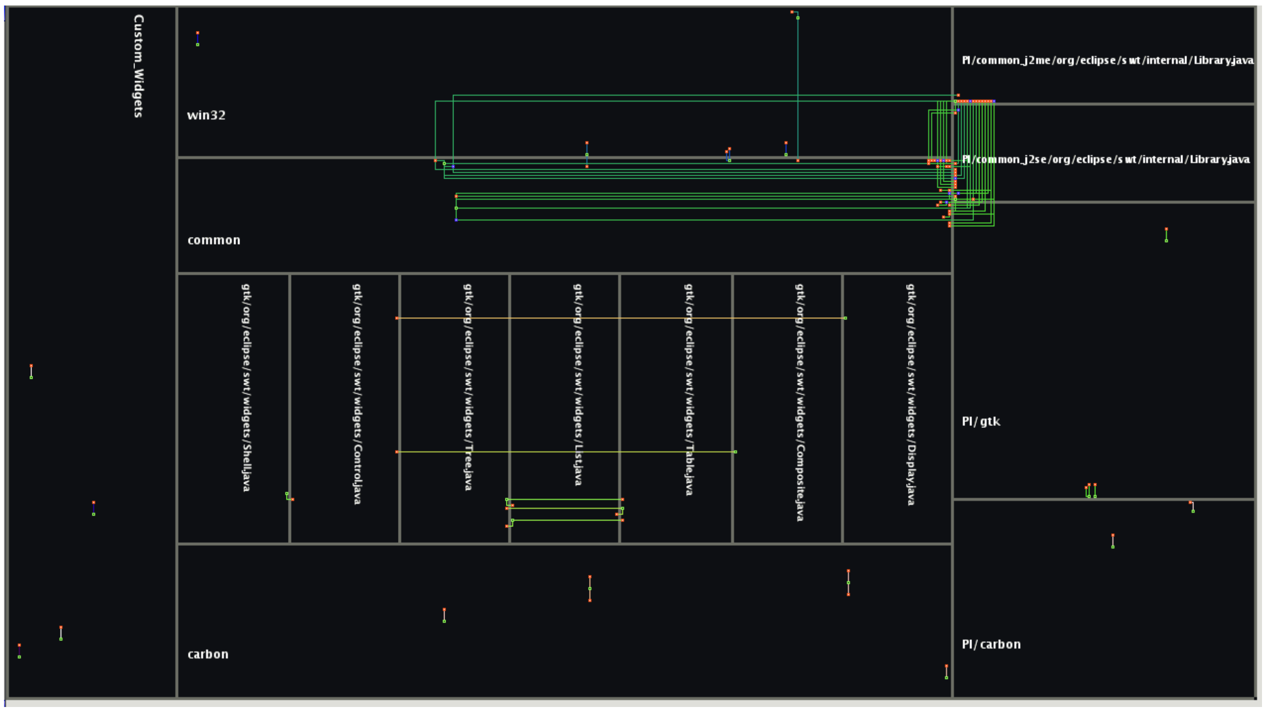
Figure 13 shows a visualization of internet routing data which have been collected from October 2001 to July 2002. This data set[6] was generated with the program `traceroute` and run from a host within the LAN of the Physics Department of the University of Rome ”La Sapienza”. The IP address of that host was 141.108.2.4.

The data set is very large and complicated what makes it difficult to examine it directly. In the following we show how our visualization technique helps to get a better understanding of this kind of data and how it helps to discover patterns or detect anomalies.

A closer look at Figure 13 reveals, that the root node (green color) is positioned in the leftmost box which is IP address 141.108.2.4. This root node is the starting point for the path colored in blue/green and which ends (red color) at IP address 192.9.104.17. But the path splits at IP address 212.1.200.25, which is indicated by the blue path starting from there and ending at IP address 4.1.122.230. So each time the routing path splits, an additional path in a different color is shown. Color coding is here used to distinguish between different paths. The end of a path is represented with a red colored node. The above coloring scheme of different paths helps to distinguish the routes of the packets sent.

If we take a look at the IP addresses 192.9.104.15 and 192.9.104.17 we detect an anomaly: The packets are sent forth and back between these two addresses and finally end up in one of these.

Another advantage of our visualization technique is the presence of the hierarchical structure of the IP addresses. Each address consists of four numbers in the range of 0–255. Each of these numbers can be seen as a level of a taxonomy. The numbers from left to right indicate the levels of the network, i.e. from a net to its subnets. This additional information gives us an overview whether the route of a packet changes between different local networks or not.



Custom_Widgets
win32
common
carbon
gtk/org/eclipse/swt/widgets/Shell.java
gtk/org/eclipse/swt/widgets/Control.java
gtk/org/eclipse/swt/widgets/Tree.java
gtk/org/eclipse/swt/widgets/List.java
gtk/org/eclipse/swt/widgets/Table.java
gtk/org/eclipse/swt/widgets/Composite.java
gtk/org/eclipse/swt/widgets/Display.java
PI/common_j2me/org/eclipse/swt/internal/Library.java
PI/common_j2se/org/eclipse/swt/internal/Library.java
PI/gtk
PI/carbon

Figure 12. SWT decision tree with fully expanded GTK directory (orthogonal)

The blue path at the upper part of Figure 13 consists of very long parallel lines which alternate between IP address 157.130.60.249 and IP addresses 192.9.24.17 and 4.1.122.230 respectively. Examining the addresses in more detail we detect that the packets were sent across more than one hierarchy level.

On the other side we see the light blue path right under the dark blue path. This one represents the route of a packet which was mainly sent between IP addresses starting with number 152.*.*.*. So here, there are not many relations between different levels of the hierarchy. This means the packets have not left this network very often.

Also a comparison of two paths is possible with our visualization technique. If we look at the dark blue and the light blue path again, we find that these two paths have a common subsequence of IP addresses that are not prefixes or postfixes of the whole sequence. They share the same consecutive IP addresses starting from 152.63.19.29 to 152.63.55.82. This pattern can be directly detected, because there is a sequence of neighbored blue node pairs.

If more than one red colored node is placed within the same box this means that more than one path exists which finally ends at the same IP address. This is for example the case for the green and the light blue paths which both end at IP address 192.9.24.17.

7. RELATED WORK

Multitrees[7] can be used to draw multiple hierarchies, but they are not applicable in our case, as they are too restrictive, because there must be a single path to each object.

Separate and colored tree diagrams have for example been investigated by Mukherjea et. al.,[8] respectively by Graham et. al..[9]

The basic idea of treemaps[10] is to recursively divide an area into non-overlapping subareas according to a given hierarchy. Many variants have been developed during the last decade including ordered,[11] squarified[12] and cushion treemaps.[13] Our current implementation would certainly benefit from using these instead of the straightforward approach of computing the treemap currently used.

ZTrees[14] are similar to our *Trees-in-a-Treemap* visualization with straight lines. We are not aware of any work on combining orthogonal graph drawing with treemaps.

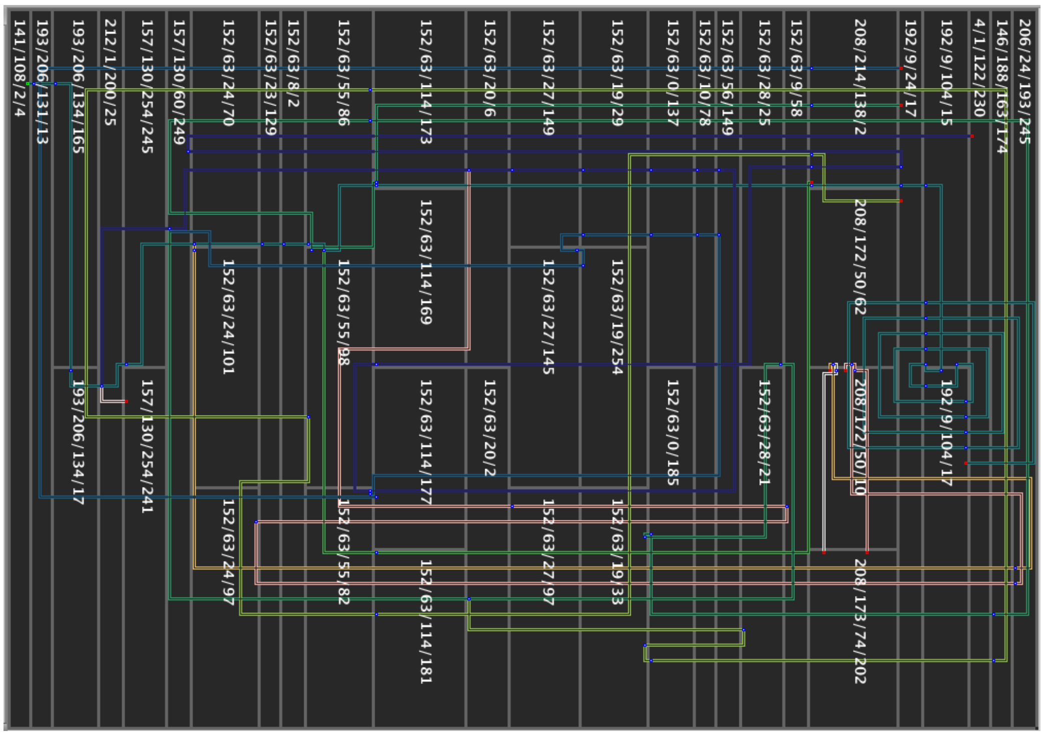


Figure 13. Several internet routes from the same host drawn in a fully expanded hierarchy of IP addresses

Robertson[15] uses animation to show multiple hierarchies and connect single hierarchy nodes with links. We don't need animation in our visualization technique because the correspondence between taxonomy and object tree can be seen within the treemap boxes directly.

Fekete[16] uses Bezier curves in his InfoVis toolkit to show links between nodes in the treemap. The starting and end points of the curves are always placed in the middle of the treemap boxes. This would cause lines overlaying each other if he would use straight lines instead of Bezier curves.

8. CONCLUSIONS

The visualization of hierarchical information has always been a focus of research in information visualization.

In many applications we find trees of objects that are also classified in some taxonomy. In this paper we looked at several known and some new techniques to draw this kind of trees such that for each object its position in the taxonomy is easily visible.

In particular, orthogonal layout of trees in a treemap turned out to offer many advantages over the other approaches: there is a single representation for each object, the visualization is compact, the number of edge crossings is reduced, the orthogonal edges are easy to follow, clusters and outliers can be detected and above all the taxonomy is visible in form of the treemap. In addition, parts of the treemap and as a consequence, parts of the object tree can be collapsed.

To show the usability of our new visualization technique, we looked at two different data sets: decision trees and network routing data. In the first case, we found, in particular, cross cutting relations, i.e., dependencies between directories in different parts of the directory tree. In the second case, we immediately spotted an anomaly – packets being repeatedly sent back and forth.

ACKNOWLEDGMENTS

Peter Weißgerber gave helpful comments on earlier revisions of this paper. Moreover, he implemented the mining of the SWT sequence rules.

REFERENCES

1. A. Inselberg and B. Dimsdale, “Parallel Coordinates: A Tool for Visualizing Multi-Dimensional Geometry,” in *Proc. of Visualization ’90*, pp. 361–378, IEEE Press, (San Francisco, USA), 1990.
2. M. Burch, S. Diehl, and P. Weißgerber, “Visual Data Mining in Software Archives,” in *Proceedings of ACM Symposium on Software Visualization SOFTVIS’05*, (St. Louis), May 2005.
3. M. R. Garey and D. S. Johnson, *Computers and Intractability A guide to the Theory of NP-Completeness*, W.H Freeman Publ., 1979.
4. T. Zimmermann and P. Weißgerber, “Preprocessing CVS Data for Fine-Grained Analysis,” in *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, (Edinburgh, Scotland, U.K.), 2004.
5. R. Agrawal and R. Srikant, “Mining Sequential Patterns,” in *Proc. International Conference on Data Engineering (ICDE’95)*, (Taipei, Taiwan), 1995.
6. C. project, “Network data sets – traceroutes – data sets.” `http://www.cosin.org/extra/data/traceroute/imdb.html`, 2005.
7. G. W. Furnas and J. Zacks, “Multitrees: enriching and reusing hierarchical structure,” in *Proceedings of the SIGCHI conference on Human factors in computing systems CHI’94*, ACM, (Boston, Ma), 1994.
8. S. Mukherjea, J. D. Foley, and S. Hudson, “Visualizing complex hypermedia networks through multiple hierarchical views,” in *Proceedings of the SIGCHI conference on Human factors in computing systems CHI’94*, ACM, (Denver, Colorado, United States), 1995.
9. M. Graham, J. B. Kennedy, and C. Hand, “A Comparison of Set-Based and Graph-Based Visualizations of Overlapping Classification Hierarchies,” in *Proceedings of the Working Conference on Advanced Visual Interfaces AVI’2000*, (Palermo, Italy), 2000.
10. B. Johnson and B. Shneiderman, “Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures,” in *Proceedings of IEEE Visualization Conference*, (San Diego, CA), 1991.
11. B. Shneiderman and M. Wattenberg, “Ordered Treemap Layouts,” in *Proceedings of IEEE Symposium on Information Visualization INFOVIS01*, 2001.
12. M. Bruls, K. Huizing, and J. J. van Wijk, “Squarified Treemaps,” in *Proceedings of Joint IEEE TCVG Symposium on Visualization*, 2000.
13. J. J. van Wijk and H. van de Wetering, “Cushion Treemaps: Visualization of Hierarchical Information,” in *Proceedings of IEEE Symposium on Information Visualization INFOVIS99*, (San Francisco), 1999.
14. L. Bartram, A. Uhl, and T. Calvert, “Navigating Complex Information with the ZTree,” in *Proceedings of Graphics Interface 2000*, 2000.
15. G. Robertson, K. Cameron, M. Czerwinski, and D. Robbins, “Animated Visualization of Multiple Intersecting Hierarchies,” in *Journal of Information Visualization, Vol. 1, No. 1, March 2002*, 2002.
16. J.-D. Fekete, “The InfoVis Toolkit,” in *Proceedings of the 10th IEEE Symposium on Information Visualization (InfoVis’04), IEEE Press, 2004, pp. 167-174.*, 2004.