



Centrum voor Wiskunde en Informatica

REPORTRAPPORT

Tree Visualisation and Navigation Clues for Information Visualisation

I. Herman, M. Delest, G. Melançon

Information Systems (INS)

INS-R9806 March 31, 1998

Report INS-R9806
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.

SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Tree Visualisation and Navigation Clues for Information Visualisation

I. Herman

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
email: Ivan.Herman@cwi.nl

M. Delest, G. Melançon

Laboratoire Bordelais de Recherche en Informatique (LaBRI), Université Bordeaux I
351, cours de la Libération, 33405 Talence Cedex, France
email: {[Maylis.Delest](mailto:Maylis.Delest@labri.u-bordeaux.fr), [Guy.Melancon](mailto:Guy.Melancon@labri.u-bordeaux.fr)}@labri.u-bordeaux.fr

ABSTRACT

Information visualisation very often requires good navigation aids on large trees, which represent the underlying abstract information. Using trees for information visualisation requires novel user interface techniques, visual clues, and navigational aids. This paper describes a visual clue for trees as well as an automatic folding (clustering) technique, both based on some mathematical concepts and results in combinatorics. Examples are shown how these techniques can be used, and what the further challenges in this area are.

1991 Computing Reviews Classification System: D.2.2, G.2.1, G.2.2., H.5.2, I.3.6, I.3.8

Keywords and Phrases: information visualisation, tree visualisation, graph visualisation, user interfaces

Note: This paper has been submitted as a journal publication. At CWI, the work was carried out under the project INS3.2 "Information Visualization".

The on-line version of this report contains parts of the figures in colour¹.

1. Introduction

The problem of displaying and interacting with abstract information, which is a central task of information visualisation, can very often be abstracted to the problem of displaying and interacting with graphs. This is the case when attempting to visualise file spaces, hypermedia document structures, internal data structure of computer programs, etc.

Graph drawing is extremely complex, and the available results draw on such diverse fields as topological graph theory, computational geometry, combinatorics, graphical user interfaces, etc. A comprehensive bibliography on graph drawing algorithms[1] cites more than 300 papers published before 1993; specialised conferences (the so-called Graph Drawing meetings) take place every year, and special issues of various journals (e.g., a special issue of the journal *Algorithmica*, in 1996) are published on the subject. The reason for this high level of activity is the fact that the simply phrased problem of drawing a graph in a plane (or in space) turns out to be, in general, extremely demanding. Some of the questions can lead to NP hard problems; a solution may involve significant computing resources, and may be totally unsuitable for interactive purposes.

Roughly speaking, there are two major, albeit intertwined, problem areas:

- *How to draw a graph on a plane (or in space)?* This line of research tries to find answers to questions like: is it possible to draw the graph on a plane without edge intersections? How to draw the graph by minimising, for

¹) See <ftp://ftp.cwi.nl/pub/CWIreports/INS/INS-R9806.ps.Z>

example, edge lengths, or by making the edge length proportional to some application-dependent measure? How to draw a graph by respecting some aesthetic constraints?

- *How to interact with (the visual representation of) a graph?* This raises questions like: how to access information stored in a node? How to “move around” in a graph in search for some specific pattern or node? How to change the visual representation as an answer to user interaction?

Instead of trying to solve the problems for general graphs, an obvious approach is to concentrate on some special class or classes of graphs, and first to develop algorithms and visualisation techniques well adapted for this class. Hopefully, these results can then be generalised. The work presented in this paper concentrates on trees only; it is the topics of our further research activities to generalise the results to more general cases.

In spite of the large body of results on graph drawing, it is not simple to use these algorithms and techniques for information visualisation. Indeed, information visualisation, which is inherently interactive, raises a number of issues which are not necessarily covered by the classical research on graph drawing. Apart from obvious problems such as speed (in the case of a graph with 3-4000 nodes, the display of the graph should not take more than a second), there are two additional, important aspects:

- *Predictability.* Two different runs of the algorithm, involving the same or similar graphs, should not lead to radically different visual representation. This is very important if the graph is interactively changed, for example by (temporarily) hiding some nodes or making them visible again. If, as a result of such interaction, the graph drawing algorithm creates a radically different view of the graph, the user will be “lost”, and the application may become unusable.

This requirement rules out a large percentage of the graph placement algorithms published in the literature. Indeed, graph visualisation is often based on optimization; these techniques look for a local minimum for a function, based on the overall topology of the graph. If the graphs changes, the new local minimum may lead to a dramatically different visual representation, which is unacceptable for interactive use.

- *Navigation on large or unusual graphs.* The examples one usually finds in publications concerned with placement and navigation algorithms rarely address graphs with more than 1-200 nodes. However, practical applications lead to thousands, or even tens of thousands of nodes; navigational techniques do not necessarily scale well for these numbers. New techniques, combined with old ones, have to be found to make an interactive environment really useful.

The number of nodes is not necessarily the only factor to consider. Some application may generate graphs with unusual structures, e.g., very “wide” compared to its size. A typical example is a tree representing hierarchical file systems (see Figure 8) which is not very deep but, comparatively, very wide. Publications rarely address navigational aids tailored at these cases.

Our research, presented in this paper, concentrated on the development of tools which help the user in navigating in large trees. These techniques are based on results in combinatorics which are not widely known by the research community in computer graphics.

The structure of the paper is as follows. Section 2. gives an overview of the tree placement algorithm we used. Although this algorithm has been published a long time ago, it is not very well known by computer graphics experts; on the other hand, it is necessary to know the essence of the algorithm to understand the results presented in further sections. Section 3. presents the usage of the so-called Strahler numbers for as navigational aid in large trees. Some further usage of the Strahler numbers is presented in section 4.. Section 5. describes the “folding” algorithm we have developed, which helps the creation of a hierarchical view of large trees, and the paper ends with some general conclusions and direction for further research in section 6.

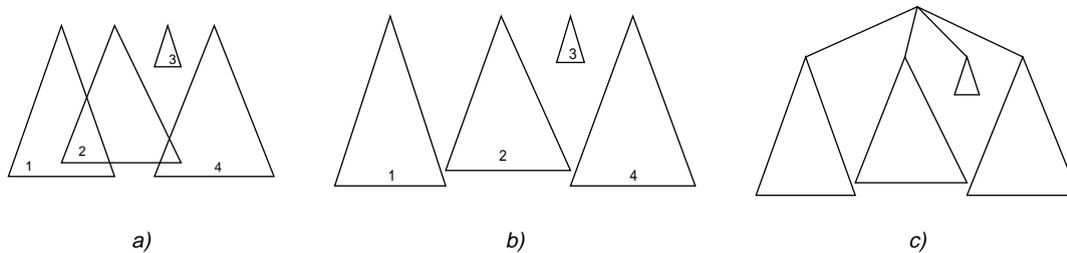


Figure 1. The Reingold–Tilford algorithm

2. Tree placement

2.1. The basic tree placement algorithm

It is somewhat surprising to realise, when going through the survey of di Battista et al.[1] that, comparatively, only a few papers deal with tree drawing algorithms. The reason probably is that an algorithm, published as far back as 1981, seems to give a very satisfying solution for the problem of tree drawing. The algorithm by Reingold and Tilford[6] provides a fast algorithm which produces aesthetically nice trees on the plane. Apart from being reasonably simple to implement, the result of the algorithm is also predictable (in the sense described above).²

Conceptually, the algorithm recursively traverses the tree; it tries to place the nodes bottom–up, i.e., from the leaves toward the root of the tree. At a given level of hierarchy, it considers the subtrees as rigid units. What it does is, from left to right:

- 1) Place the apex nodes of the subtrees (i.e., the siblings) to a proper minimal distance from one another (see Figure 1/a). This step may also take the size of adjacent sibling nodes into consideration.
- 2) Shift the subtrees, if necessary, to avoid the intersection of the subtrees. Shifting the subtree means shifting all nodes in the subtree to the right with the same distance (see Figure 1/b).³

When all subtrees have been taken care of, the position of the common parent of all siblings is calculated by placing it on the average horizontal coordinate of the siblings’ position (see Figure 1/c). The recursive step can then continue one step “higher”.

If performed literally, the algorithm would be much too slow. Indeed, shifting a subtree means to shift all the nodes in the subtree, i.e., the shifting step would involve a large amount of traversal through all the subtrees. To avoid this problem, Reingold and Tilford used the concept of a preliminary x coordinate and a ‘modifier’ field in the node. Using these temporary information, the tree placement is performed in two traversal passes through the whole tree, namely:

- 1) The first tree traversal is post–order (i.e., the node is modified *after* all children have been modified). In each node, an intermediate x coordinate value is set, plus the modifier value; this latter indicates the amount which has to be added to the intermediate x coordinate value *for all nodes in the corresponding subtree*. Using a computer graphics terminology, one can say that the children of a node are positioned in a local coordinate system of the node (i.e., the children are positioned from left to right starting from the coordinate value zero),

²⁾ To be somewhat more precise, we used the paper of Walker[10] as a basis, instead of the original paper of Reingold and Tilford. This paper publishes the algorithm in a Pascal–like language, which was useful to understand the details. Walker also made a small modification to the original algorithm, which was worth taking over

³⁾ Note that the original Reingold–Tilford algorithm shifted the rightmost subtree only (when proceeding from left to right) whereas Walker also shifted the possible intermediate nodes, to avoid visual ‘gaps’ in the placement.

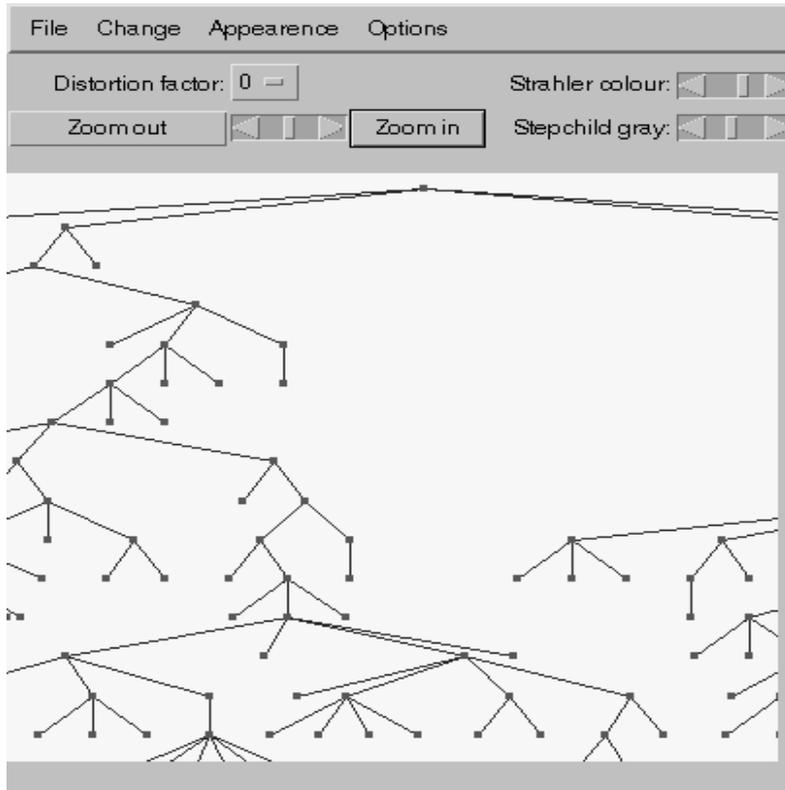


Figure 2. Missing visual clues in tree navigation

and the modifier of the node represents the coordinate transformation (in this case, a shift) which has to be applied on each node.

- 2) The second tree traversal is pre-order, (i.e., the node is modified *before* its children are modified). This step calculates the final x coordinate values for each node, by accumulating the modifier values in the process (i.e., accumulating the coordinate transformations).

For our work, we have re-implemented the algorithm in Java, and incorporated it into an interactive shell for visualisation and navigation. This shell includes such standard techniques as zoom and pan, as well as a possibility for a fish-eye view of the tree (see, e.g., Sarkar and Brown[7] for the usage of fish-eye views of graphs). The figures in the paper are screen-dumps of this environment.

Using the fish-eye view proved to be a very satisfactory solution for navigation, provided that the graph on the screen was reasonably small. It has the well-known advantage that the user, while concentrating on some detail, still has an overall view of the graph. However, usage of fish-eye view has its limits; if there are too many nodes on the screen, the user is forced to use zoom and pan. This leads to the disagreeable feeling of being “lost”; a simple zoom and pan does not give enough visual clues to the user to keep his or her orientation and position within the tree. Providing such clues is one of the main techniques we have explored, and this is presented in the next section.

3. Visualisation hints of complex subtrees

The problem, when navigating in a tree using zoom and pan, is shown on Figure 2. The user does not know where to move when panning; the image on the screen does not help him or her to make the appropriate choice. (Of course, thumbnail images of the whole tree can be created in a separate window, but this is not a fully satisfactory solution either). The goal of a proper visual clue would be to indicate in which direction the tree really grows, where are the more complex parts, i.e., where should the user go to explore interesting subtrees.

Such visual clues can be obtained based on the so-called Strahler, or Horton–Strahler numbers[3,8]. The Horton–Strahler numbers can be considered as an attempt to give a quantitative information about the complexity, or the “shape” of the tree. These numbers have been extensively used in such various fields as hydrology (the original work of Strahler was related to the morphological structure of river networks), molecular biology, and have also been used in computer graphics[9], although very rarely. They have also been the subject of active combinatorial research in mathematics.

In the mathematical literature Strahler numbers are defined for binary trees, where each node is either a leaf, or has exactly two children. The Strahler number of a node can be defined as follows. Let us denote the node by N , its left child by N_l and its right child by N_r . If S_N denotes the Strahler number of a node, the following formulae define the Strahler numbers for binary trees:

$$S_N = \begin{cases} 0 & \rho_N = 0 \\ S_{N_l} + 1 & S_{N_l} = S_{N_r} \\ \max\{S_{N_l}, S_{N_r}\} & \text{otherwise} \end{cases}$$

This value has been successfully used to measure the “complexity” of binary trees: higher values indicate more complex (binary) subtrees. The values can also be seen as indicators on the amount of “energy” which is forwarded through a node to its ancestors.

This binary tree version had to be modified for our purposes to include an arbitrary tree. The modification we have adopted takes into consideration the number of children of a node; obviously, this value is also a factor of complexity in general trees. Denoting by ρ_N the number of children of the node N , the formulae for this modified Strahler number are:

$$S_N = \begin{cases} 0 & \rho_N = 0 \\ S_{N_i} + \rho_N - 1 & \forall i, j \cdot S_{N_i} = S_{N_j} \\ \max\{S_{N_i}\} + \rho_N - 2 & \text{otherwise} \end{cases}$$

where N_i , $i=1, \dots, \rho_N$ denote the children of the node N . The two factors 1 and 2 are simply present to provide a true generalisation of the binary Strahler numbers.

A further generalisation of the Strahler number is to combine these formulae with an additional (application dependent) weight of a node, i.e., a number which somehow characterises the “importance” of the node (for example, if the tree describes a file system tree, this value may indicate the size of a file). This leads to the “weighted” Strahler numbers, denoted by Sw_N , and which are defined by modifying the formulae by accumulating the weight, too:

$$Sw_N = \begin{cases} w_N \\ S_{N_i} + \rho_N - 1 + w_N \\ \max\{S_{N_i}\} + \rho_N - 2 + w_N \end{cases}$$

where w_N denotes the weight associated to the node N . In what follows, we will refer to the Strahler numbers only, which can refer either to the weighted or the simple version; it is application or user dependent which of the two versions are used. Note that the Strahler numbers can be calculated once for the full tree at initialisation time, i.e., its calculation does not influence the interactive response time of a graph visualiser system.



Figure 3. Tree navigation with visual clues

The Strahler values (whether weighted or not), describe the structural complexity of the subtree at a given node N . The word “structural” is important, i.e., this value is independent of the tree placement algorithm used to display the graph and reflects the inherent complexity of the information which is visualised. Because this Strahler value can be calculated for each node, it can be used to implement the necessary visual clue for a subtree. The result can be seen on Figure 3 (see also Figure 6 for a colour version). The way the Strahler number is used in these figures is twofold:

- A minimum and maximum edge width is defined, and this range is linearly mapped against the range of Strahler numbers. Wider edges lead toward more complex subtrees; the effect is a bit like a network of rivers with wider portions carrying more information.
- Similarly to linewidth, a colour is chosen with a specified hue; the saturation of the colour varies when displaying the edges. This is done by defining a linear mapping between the maximum and minimum saturation and the available Strahler number ranges⁴. (Note that the user might choose not to use colours, e.g., if a black and white screen is used, or the content of the display has to be printed on a black and white printer.)

Once these mappings are established, the edge is displayed using the colour saturation value and the edgewidth corresponding to the child’s node it connects to. The visual effect is to emphasise those links which lead toward more complex subtrees.

Figures 2, 3, and 6 represent the same portion of a tree, with the same zooming and panning factors. If the traditional view is used (Figure 2), and the user wants to slide the window over the graph, it is not clear at all in which direction this should happen (if the essential parts of the tree are to be explored). Is it worth sliding to the right from the root, following the edge in the upper right-hand corner of the window? Similarly, there is a subtree starting at

⁴) Other mappings, e.g., logarithmic, could also be used at this point.

the lower half of the picture; which direction to choose in this subtree? These questions are automatically answered if the Strahler based visual clues are used, see Figures 3 or 6. When experimenting with the results (we have used it, e.g., to navigate in a tree representing hierarchical file systems, or through internal data structures of compilers) the usage of the combined effect of both the linewidth and the colour change produced the best visual clues, if the display quality was appropriate. Our experience was very satisfactory as for the help this technique gives in navigation.

Note that, at this stage, we deliberately tried to use relatively simple graphics in defining the visual clues; the goal being information visualisation, the possible target applications may have to rely on simple graphics facilities only, duly represented by Java's AWT package. More complex visual clues could be defined and implemented if more elaborate tools were used (e.g., Java2D, or other, more powerful graphics hardware and packages) but, at this stage, we chose not to use them.

In this section, we have concentrated on the usage of these visual clues for the purpose of pan and zoom. However, even if used for a full tree, the effect is a better overall impression of the tree, which is an important feature by itself, too (see Figure 7). (The tree is not really large, but the size of this paper does not allow us to include a really large example.)

4. Further usage of Strahler numbers

Complexity colouring is not the only way Strahler numbers can be used. To show the importance of these numbers, this section just hints at two other means to influence the outlook of a tree.

- The Reingold–Tilford algorithm produces very “balanced”, or “symmetrically looking” trees. The relevant step in the algorithm is the one referred to by Figure 1/c: after having “placed” all subtrees, a node's position is established by taking the middle of all the children's position, which produces the symmetrical effect.

The Strahler number allows us to modify this step by placing the parent “closer” toward its high complexity subtree. This is done simply by changing the placement of the parent using the weighted average of the children's Strahler numbers.

Figure 4 shows the effect of this change on the same tree. The tree's position is “skewed” to make the high complexity subtree more visible. Whether this is the kind of view the user wants to have is clearly applications and user-dependent, and should be given as an option to the end-user.

- If, in contrast, even greater symmetry is required, a simple rearrangement of all children of a node can help. This means that, prior to the run of the Reingold–Tilford algorithm, the children are reordered by placing the one with a maximum Strahler in the middle, and have children with decreasing values symmetrically arranged on both sides. Note, however, that some application may not allow a reordering of the nodes; indeed, the order may contain some semantics which is important for the application.

None of these modifications are particularly complex to implement; they show, however, that a combinatorial characterisation of trees may have a beneficial effect on choosing the right view of a tree, depending on the application's needs.

5. Interactive and automatic folding

Although zooming, panning, fish-eye, and complexity visual clues are all extremely important when navigating on a tree, beyond a certain size it becomes unavoidable to have a hierarchical view of a tree. What this means is that a specific subtree has to be hidden (“folded”), i.e., only the root of the subtree should stay visible. When the tree becomes too large, this is the only way the user can still manage the necessary interaction.

Folding raises two important issues, which will be examined in more details later in this section. These are as follows.

- Folding (and unfolding) should be fast and the result should be predictable in the sense described in the introduction. If folding a subtree, or opening it again, creates a totally different visual image of the tree, the user

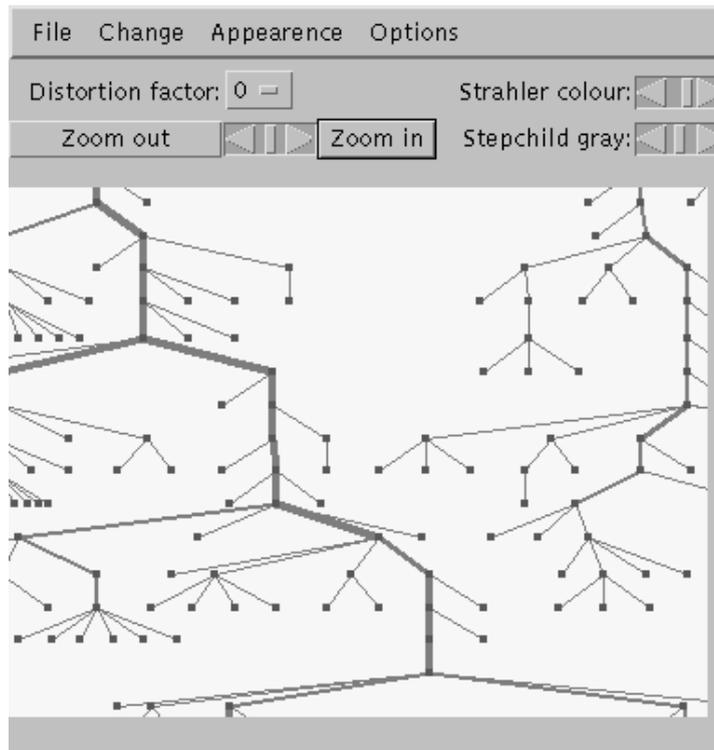


Figure 4. Tree placement with Strahler skew

will not be able to find his/her way in the new image. This is an important aspect of predictability. (Although this requirement seems to be obvious, we have seen applications which seem to ignore this requirement).

- Although folding can be (and should be) doable interactively, this should not be the only way for folding; choosing a series of “foldable” subtrees manually is a tedious work if the tree is too large. An automatic folding algorithm becomes necessary in such a case, i.e., an algorithm which automatically identifies nodes which define unusually large or small subtrees, and which therefore make the overall image of the tree difficult to grasp. If identified, these nodes can then all be folded in one step.

We will address both these issues separately.

5.1. The folding/unfolding step

One of the advantages of the Reingold–Tilford algorithm is its predictability. This means, in this case, that if the tree is (interactively) modified, the visual effect on the tree itself is local. This was, in fact, one of the reasons we chose this algorithm, and we rejected some other, optimization based approaches.

Furthermore, folding a subtree (or, later, unfold a subtree) can be done without necessarily repositioning the full tree from scratch. Indeed, if a subtree is folded, i.e., only its apex node remains visible, only the ‘shift step’ has to be repeated, starting from the layer where the change occurred, and propagate this change toward the root. In other words, the recursive placement of subtrees is not performed, and the shift operation is also restricted to the affected siblings. If unfolding is done, the “new” subtree has to be positioned but, again, it is enough to propagate the changes toward the root, instead of reordering the full tree. This is a feature of the Reingold–Tilford algorithm which, to our knowledge, has not been exploited before. Although it is difficult to estimate the speed improvement, because it is dependent on the exact position of the apex within the full tree, on the average, folding and unfolding may be two–three times faster than running the algorithm on the full tree again.

5.2. Automatic fold

While the fold/unfold step itself has proven to be very simple, finding the nodes which are good candidates for automatic folding is a much more complex task. The approach we have taken can be outlined as follows:

- 1) For a tree T with n nodes, a “folding function” $f_n(T)$ is defined, which assigns a numerical value to a tree T . This function should somehow characterise whether the tree takes “too much place” on the screen.
- 2) If n is fixed, f_n is considered as a random variable over the set of all trees with n nodes. The statistical behaviour of this function is then analysed, with the goal of finding its mean and standard deviation.

The standard deviation and the mean of f_n yields two variables, α_n and β_n , which define the “normality” range for the value of $f_n(T)$. In other words, if the condition

$$\alpha_n \leq f_n(T) \leq \beta_n \quad (\text{EQ 3})$$

holds, the tree T has a “normal” geometry, whereas it is abnormal otherwise.

- 4) If both 1) and 2) above are available, an automatic folding algorithm can work by recursively considering each subtree in the tree, and fold all “abnormal” ones. To be more precise, the following steps are executed:
 - i) The function $f_n(T)$ is evaluated for each node of the tree.
 - ii) The tree is traversed in post-order, i.e., the children of each node are examined first. If this child is “abnormal”, it is folded and the values of $f_n(T)$ are re-evaluated for the full tree, with the folded subtree considered now as a leaf. (Note that in some cases a full re-evaluation of the values for the full tree may not be necessary or is very easy, but this depends on the choice of the folding function).

In other words, the algorithm collects all the abnormal nodes bottom-up and folds them on the fly.

The algorithm has a halting condition on the number of nodes; if this value falls under an application-dependent value for a subtree, no folding occurs, and the algorithm stops.

Of course, this is only the skeleton. Finding a good folding function, whose statistical behaviour is known and whose evaluation on a tree is not prohibitively slow, is not a simple matter. The first approach we tried was to base the folding function on the Strahler numbers; this looked like a promising choice, because the statistical behaviour of the binary Strahler numbers are known[3]. However, the extended version of the Strahler numbers proved to be statistically unmanageable; besides, these numbers reflect a structural complexity rather than a geometric one.

Our final choice for the folding functions was based on the simple observation that a tree becomes very wide, i.e., it distorts the placement of the tree, if the number of leaves in the tree becomes unusually large. On the other extreme, if the number of leaves is unusually low, the image of the tree tends to be much too high compared to the information it provides, and may lead to too much empty place on the screen. Consequently, we concentrated on counting the number of leaves in a tree, and we have defined the folding function as:

$$f_n(T) = \text{Number of leaves in } T$$

Because the random variables are finite, the way of determining their statistical characteristics is based on combinatorial reasoning. One can ask the following question: given the number n and k , what is the number of trees which have exactly n nodes and k leaves? Although not trivial, the answer to this question is known[4]: if we denote this number by $B_{n,k}$, then:

$$B_{n,k} = \frac{1}{n-1} \binom{n-1}{k} \binom{n-1}{k-1}$$

The possible number of trees which have exactly n nodes is also known[12]:

$$B_n = \frac{1}{n+1} \binom{2n}{n}$$

Using these formulae, the probability values of the folding function can be described as:

$$\wp \{f_n(T) = k\} = \frac{B_{n,k}}{B_n}$$

This means that if, for a given tree, the value of $f_n(T)$ is unusually large or unusually small, this means that the number of leaves is unusually large, respectively small, i.e., the (sub)tree has a distorting effect on the display of the tree. What “unusually” large or small means has to be determined by the statistical characterisation of the f_n random variables.

The mean of the random variable is simply $n/2$ (the formula defining $B_{n,k}$ is symmetrical). The standard deviation, i.e., the values for α_n and β_n , can, of course, be calculated explicitly, but this would be way too slow, or indeed impossible, due to the characteristics of the factorial functions. However, it can be shown that, statistically, f_n can be approximated by a normal distribution, and that the approximation:

$$\alpha_n \approx \left\lfloor \frac{n}{2} - 1.96 \sqrt{\frac{n}{8}} \right\rfloor$$

$$\beta_n \approx \left\lceil \frac{n}{2} + 1.96 \sqrt{\frac{n}{8}} \right\rceil$$

holds for a value for large n values. This approximation can be deduced based on a much more general (and highly non-trivial) theorem described in a paper of M. Drmota[2]. The values for smaller n 's can be calculated off-line once, and stored in a table; this completes the algorithm. Having done these off-line calculations, it turns out that the approximation holds for n greater than 10 already (which is probably lower than a reasonable halting condition for the algorithm anyway!).

Our choice for the folding function makes the implementation of the algorithm very fast, because the steps a) and b) can be, in a sense, merged. Figure 5 shows the details of the algorithm; the call to `isNormal` (not shown on the figure) takes care of the statistical evaluation of the n and k values, as well as the halting condition of the algorithm.

Figure 8 shows the effects of the folding algorithm. The tree (on the upper left hand first image) represents a portion of a hierarchical file system. The number of nodes is not very high (about 250), but the structure of the tree is wide for its depth. This leads to the very dense figure of some of the subtrees. The image on the upper right hand corner represents the same tree after having used the automatic folding step, which properly identified the “wide” subtrees. The green nodes (as opposed to the red ones) represent the “folded” subtrees. Note that these green nodes show only the “top” of the folding hierarchy; if one opens up one of these nodes, further folded nodes may also appear. (This is due to the fact that the algorithm folds all unusual subtrees by moving from bottom-up.)

The upper right hand corner image is somewhat artificial, because the tree has not been “reordered” yet, i.e., the placement algorithm has not been used yet to produce a new image of the tree. Hence the large, blank areas. The lower image shows the result of drawing the folded tree again; the simplified structure allows us to zoom in without losing an overall view. This image also shows that the combination of the folding step with the visual clues based on the Strahler numbers, described in section 3., is very powerful; the thick lines ending in some of the green nodes on the second image clearly indicate that not only something has been folded there, but also whether the folded subtree is complex or not, i.e., whether it is worth further exploration.⁵ In other words, the use of the Strahler numbers in the image partially compensates for the loss of information induced by folding, i.e., it helps the user to keep a proper mental image of the full tree.

⁵) Note that we have used the “weighted” Strahler number in this case, where the weight of a leaf is proportional to the size of the file it represents.

```

static class FoldData {
    int n;
    int k;
}
FoldData boxOneNode(Node node)
{
    FoldData retval = new FoldData();
    if( node.children.size() == 0 ) {    // this is a leaf
        retval.n = 1;  retval.k = 1;
    } else {
        retval.n = 1;  retval.k = 0;
        Enumeration e = node.children.elements();
        while( e.hasMoreElement() ) {
            FoldData kidData = boxOneNode((Node)e.nextElement());
            retval.n += kidData.n;retval.k += kidData.k;
        }

        if( node.parent != null        &&    // not the root node
            isNormal(FoldData) == false ) {
            node.foldSubtree();           // fold the subtree
            retval.n = 1;  retval.k = 1;  // as if it were a leaf
        }
    }
    return retval;
}

```

Figure 5. The automatic folding algorithm

5.3. Variation on automatic folding

The previous section described the essence of the automatic folding algorithm; when included into an application, the algorithm must be adapted to the application's needs. None of the variations described below represent a significant change on the algorithm itself, but they can be essential for a full integration of the algorithm into a real application. Here are some of the possible variations:

- The algorithm, as described above, folds indiscriminately any node which is “abnormal”. While this is a proper choice when, for example, looking at the content of a hierarchical file system, some application may want to treat a specific subtree as “unfoldable” (i.e., they always want it to appear on the screen), or “foldable as a unit” (i.e., either the full subtree is folded or not, but not a part of it). What this means in practice is that some additional flags have to be added to each node, and the algorithm should take these flags into account, too, when deciding on the folding step.
- The two inequalities in (EQ 3) are not considered separately in the algorithm. In other words, no difference is made whether abnormality occurs because the value of $f_n(T)$ is smaller than α_n , or whether it greater than β_n . However, in some cases, differentiating between these two cases might be useful. For example, if the window, in which the tree display occurs, is unusually wide, than the subtrees which might have to be folded are those which are rather narrow and deep. This corresponds to the case when $f_n(T)$ is smaller than α_n . Conversely, the other inequality may only be considered if the window is unusually narrow and high. This means that the display of a large tree can be adapted to the real estate of the display.

6. Conclusions and further research

The framework using the results described in this paper are successfully used for various research projects. A system has been developed for the company called ACE (Associated Computer Experts), in Amsterdam; the system is used to visualise the internal data structures of compilers, developed by ACE. The tools are also used by another team at CWI to visualise traces of large scale parallel programming environments. The computer graphics team of LaBRI uses the tools to visualise the scene graphs of virtual reality applications, and it is our plan to use the tools to visualise the content of web sites. Other applications are still to come.

Beyond the explicit results themselves, this work has shown us how beneficial a synergy between information visualisation and combinatorics can be. Combinatorics has helped us to produce complexity colouring and automatic folding, which are important features for information visualisation; conversely, although not detailed in this paper, the questions and problems raised by the specific application led to questions which proved to be unusual, at best, for the combinatorics community.

There are a number of research issues which are still unexplored, and we hope to work on them together in the future. Some of these are:

- Are there alternative, better folding functions which might yield an even better folding step? For example, one might deduct a general statistical characterisation of the trees used by a specific application, and adapt the folding functions for this class, or adapt an existing folding function to a special tree population.
- How can complexity visualisation be used in the case of Directed Acyclic Graphs (DAGs)? DAG visualisation is much more complex than tree visualisation; new and better ways for displaying and navigating in DAGs are still to be developed. It is possible to generalise the Strahler numbers for DAGs, which may be a suitable point to start with in deriving good visual clues for DAGs.
- We emphasised, in section 3., that the visualisation clues are implemented using relatively simple graphics only, to abide to the requirement of simple application environments. If this restriction falls, there might be other possibilities to be explored. For example, instead of having constant colours along the edges, shaded colours can be used (a one dimensional Gouraud shading) which would improve the colour of visual clues. In such a case, the linewidth can be used to convey some other information.
- The benefits of using 3D techniques in tree/graph visualisation is still an open issue. Robertson et al.[5] have experimented with 3D techniques before; we also tried out some of their ideas as well as other 3D mappings (e.g., mapping the trees onto the surface of a sphere). As far as we were concerned, the results were inconclusive; being “lost in space” is a well-known phenomenon, which still requires further research. It is possible that our techniques, or a variation thereof, may be useful to improve 3D visualisation, too.

Note that Ware and Franck[11] show how important the usage of motion clues and stereo viewing are, when the task is visualising graph-based information; combining these clues with the additional clues described in this paper is worth considering, too.

Acknowledgements

We are especially grateful to Bèhr de Ruiter (CWI), who developed the interactive shell around the algorithms, including zoom, pan, and fish-eye. His remarks throughout the development of the tools are very well appreciated. Discussion with Jean-Marc Fedou (Université de Nice), Jean-Philippe Domenger (LaBRI), and Farhad Arbab (CWI) gave us extremely valuable feedback to our ideas; we are all grateful for their help.

Part of the work has been financed by a grant of the Franco-Dutch research cooperation programme “Van Gogh”.

References

- [1] G. di Battista, P. Eades, R. Tamssia, and I.G. Tollis: “Algorithms for drawing graphs: an annotated bibliography”. In: *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

- [2] M. Drmota: “Systems of functional equations”. In: *J. Random Structures and Algorithms*, **10**(1-2):103–124, 1997.
- [3] P. Flajolet, J.C. Raoult, J. Vuillemin, “The number of registers required for evaluating arithmetic expressions”. In: *Theor. Comput. Science*, **9**:99–125, 1979.
- [4] G. Kreweras and P. Moszkowski, “A new enumerative property for the Narayana numbers”. In: *J. Stat. Plann. Inference*, **14**:63–67, 1986.
- [5] G.G. Robertson, S.K. Card, and J.D. Mackinlay: “Information visualization using 3D interactive animation”. In: *Communication of the ACM*, **36**(4):57–71, 1993.
- [6] E. M. Reingold and J.S. Tilford: “Tidier drawing of trees”. In: *IEEE Transaction on Software Engineering*, **7**(2):223–228, 1981.
- [7] M. Sarkar and M.H. Brown: “Graphical Fish-eye views of graphs”. In: *Proceedings of the CHI’92 conference*, ACM Press, 1992.
- [8] X.G. Viennot: “Trees”. In: *Mélanges offerts à M.P. Schützenberger*, eds. A. Lascoux and D. Perrin, Hermes, Paris, 1990.
- [9] X.G. Viennot, G. Eyrolles, N. Janey and D. Arquès: “Combinatorial analysis of ramified patterns and computer imagery of trees”. In: *Computer Graphics (SIGGRAPH’89)*, **23**:31–40, 1989.
- [10] J.Q. Walker II: “A node–positioning algorithm for general trees”. In: *Software — Practice and Experience*, **20**(7):685–705, 1990.
- [11] C. Ware and G. Franck: “Evaluation stereo and motion cues for visualising information nets in three dimensions”. In: *ACM Transaction on Graphics*, **15**(2):121–140, 1996.
- [12] H.S. Wilf: *Generatingfunctionology*. Academic Press, Boston — San Diego — New York — London, 1994.

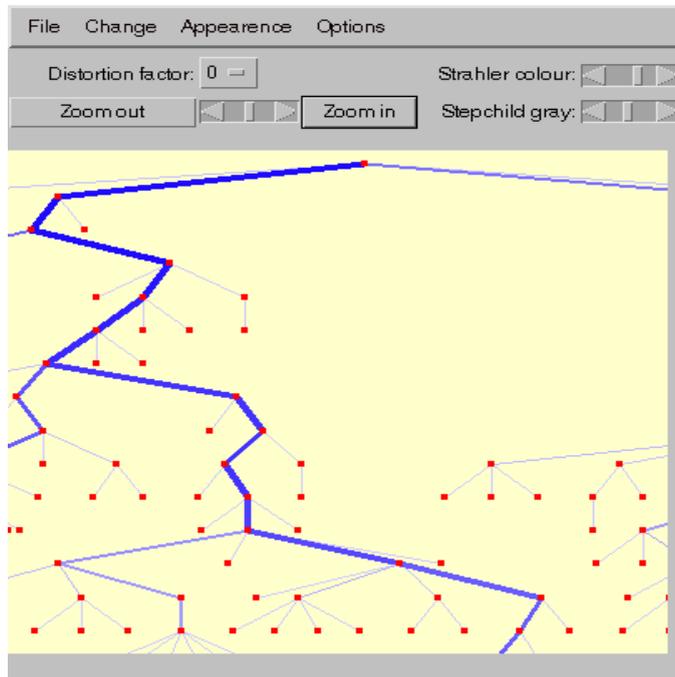


Figure 6. Coloured Strahler values on a zoomed portion of a tree

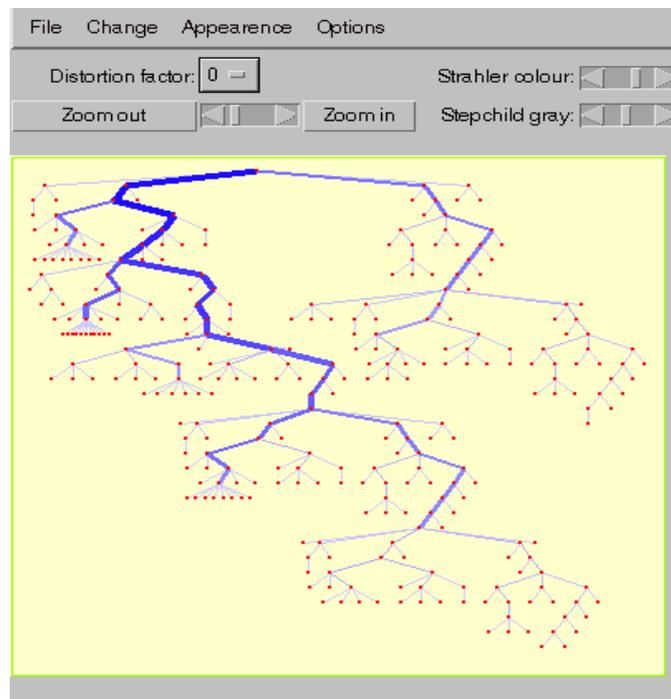


Figure 7. A full tree coloured with Strahler numbers (301 nodes)

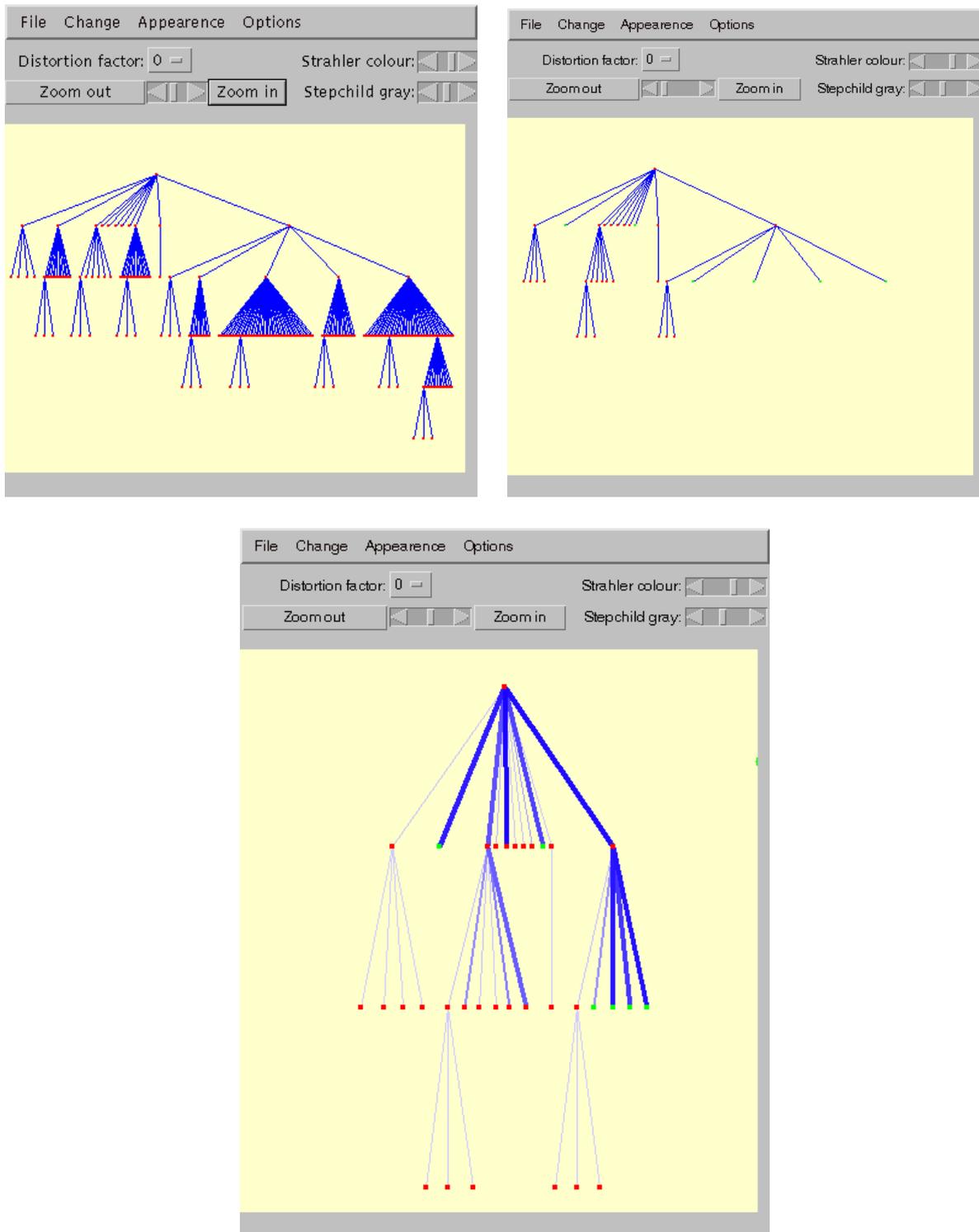


Figure 8. The effect of the automatic folding algorithm (before and after the folding step; zoom in on the simplified tree)