Volume 17 (1998), number 2 pp. 153–165

COMPUTER GRAPHICS *forum*

Tree Visualisation and Navigation Clues for Information Visualisation

Ivan Herman[†], Maylis Delest[‡] and Guy Melancon[†]

[†] Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
email:{Ivan.Herman,Guy.Melancon}@cwi.nl
[‡] Laboratoire Bordelais de Recherche en Informatique (LaBRI), Université Bordeaux I
351, cours de la Libération, 33405 Talence Cedex, France
email:Maylis.Delest@labri.u-bordeaux.fr

Abstract

Information visualisation 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: using the so-called Strahler numbers, a map is provided that indicates which parts of the tree are interesting. A second idea is that of "folding" away subtrees that are too "different" in some sense, thereby reducing the visual complexity of the tree. Examples are given demonstrating these techniques, and what the further challenges in this area are.

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

1. Introduction

The problem of displaying and interacting with abstract information, which is a central task of information visualisation, can 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 relies on such diverse fields as topological graph theory, computational geometry, combinatorics, graphical user interfaces, etc. A comprehensive bibliography of 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 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 interrelated, problem areas:

- *How to draw a graph on a plane or in space?* This line of research tries to find answers to questions such as: is it possible to draw the graph on a plane without edge intersections? How to draw the graph by minimising edge lengths, for example, 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 such as: 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 a special class, or classes, of graphs, and initially develop algorithms and visualisation techniques which are well

© The Eurographics Association and Blackwell Publishers Ltd 1998. Published by Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK and 350 Main Street, Malden, MA 02148, USA.

suited for this class. Hopefully, these results can then be generalised. The work presented in this paper focuses on trees only; our further research activities will apply 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 (e.g., in the case of a graph with 3–4000 nodes, the display of the graph should not take more than a second), there remain two 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 means that great care should be taken on which layout algorithm is chosen. For example, a number of graph layout algorithms use optimisation techniques, i.e., they look for a local minimum of 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 layout 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 at these numbers. For example, the size of the data structures used internally by a compiler can be significant; the tree representing a simple "Hello world!" program in C may contain 50–60 nodes[‡]. Visualisation of WWW sites, of database query results, of the data structures representing virtual reality scenes, etc., are other examples for possibly huge trees occurring in practice. 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 7) which is not very deep but, comparatively, very wide. Publications rarely address navigational aids tailored to these cases.

Our research has 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 layout algorithm we used. Although this algorithm was published some 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 use of Strahler numbers for 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 to create a hierarchical view of large trees, and the paper ends with some general conclusions and directions for further research in Section 6.

2. Tree Layout

2.1. The Basic Tree Layout Algorithm

It is somewhat surprising to realise, when going through the survey of di Battista et al.[1] that, comparatively few papers deal with tree drawing algorithms. The reason is probably 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[4] 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).[§]

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, from left to right, is:

1. Place the apex nodes of the subtrees (i.e., the siblings) at a proper minimal distance from one another (see Figure 1/a). This step may also take the size of adjacent sibling nodes into consideration.

[†] The term "preservation of a mental model" is also used to describe this requirement, see, e.g. Misue et al.[2] or Lin Huang et al.[3].

[‡] In fact, this was one of the problems which triggered our research work on the subject.

[§] To be somewhat more precise, we used the paper of Walker[5] 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 modification to the original algorithm, which was worth adopting.

© The Eurographics Association and Blackwell Publishers Ltd 1998

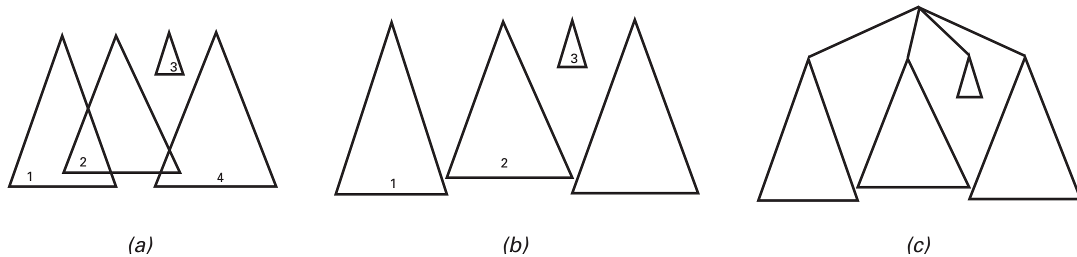

Figure 1: *The Reingold–Tilford algorithm*

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).¶

When all subtrees have been handled, 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 precisely as described, 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 this temporary data, the tree layout 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 computer graphics terminology, one can say that the children of a node are positioned in the local coordinate system of the node (i.e., the children are positioned from left to right starting from the coordinate value zero), 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).

¶ 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 layout.

For our work, we have re-implemented the algorithm in Java, and incorporated it into an interactive interface for visualisation and navigation. This interface 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[6] 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 a particular 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. 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 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[7, 8]. The Horton–Strahler numbers can be considered as an attempt to give quantitative information about the complexity, or the "shape" of the tree. These numbers have

© The Eurographics Association and Blackwell Publishers Ltd 1998

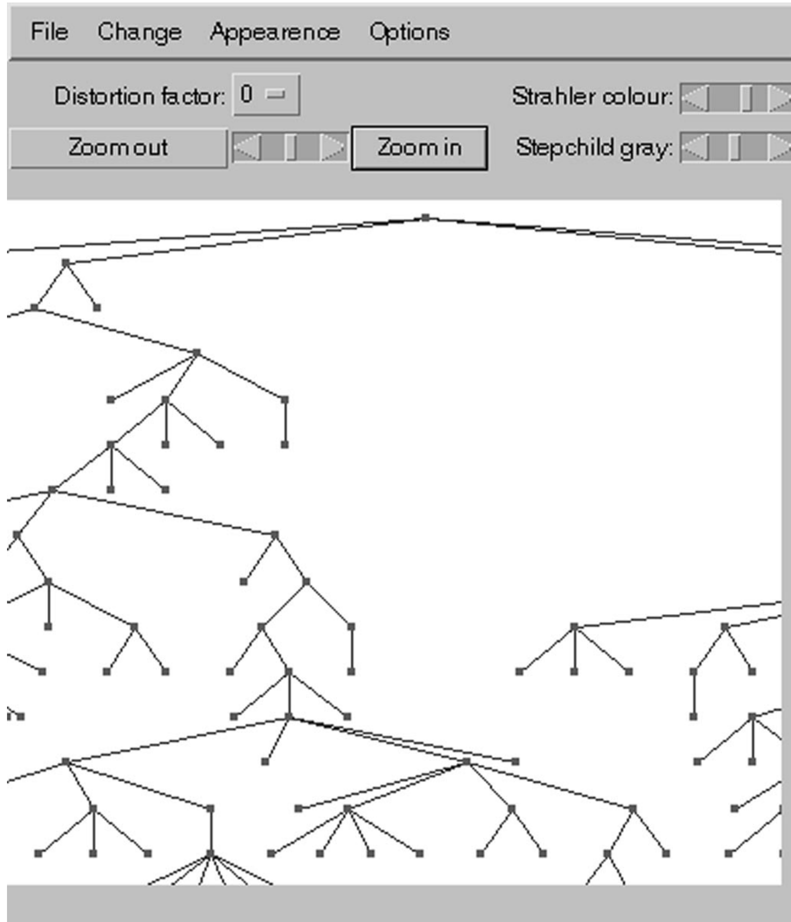

Figure 2: *Missing visual clues in a tree navigation*

been extensively used in such fields as hydrology (the original work of Strahler was related to the morphological structure of river networks), molecular biology, botany, and have also been used in computer graphics[9], although on rare occasions. They have also been the subject of active combinatorial research in mathematics. Practical experiences with Strahler number in various application areas have shown that they offer a useful characterisation of the structural complexity of trees, hence our choice to use these numbers for the purpose of information visualisation.

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 a node of the tree by N; if the tree is binary, the left and right children of the node will be denoted by N_l and N_r, respectively. In the case of general trees, the children will be denoted by $N_i, i = 1, \ldots, \rho_n$, where ρ_n denotes the number of children for the node N.

If S_N denotes the Strahler number of a node, the following formula defines 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 of the amount of "energy" which is forwarded through a node to its ancestors.

For our purposes, this binary tree version had to be generalised. The modification we have adopted takes the number of children of a node into consideration; obviously, this value is also a factor of complexity in general trees. The formula for this modified Strahler

© The Eurographics Association and Blackwell Publishers Ltd 1998

Figure 3: *Tree navigation with visual clues*

number is:

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

The two factors 1 and 2 are meant to provide a true generalisation of the binary Strahler numbers.

A further generalisation of the Strahler number is to combine these formula 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 formula by accumulating the weight, too:

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

where w_N denotes the weight associated with the node N. In what follows, we will refer to the Strahler numbers only, which can be either the weighted or the simple version depending on the application. 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.

The Strahler number describes the structural complexity of the subtree at a given node N. The word "structural" is important, i.e., the values are independent of the tree layout algorithm used to display the graph and reflect 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 4 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 sat-

© The Eurographics Association and Blackwell Publishers Ltd 1998

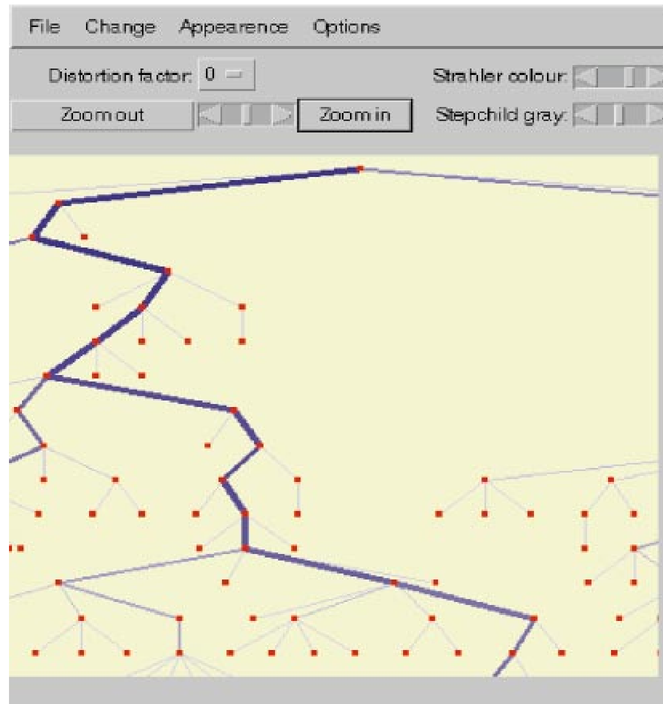

Figure 4: *Coloured Strahler values on a zoomed portion of a tree*

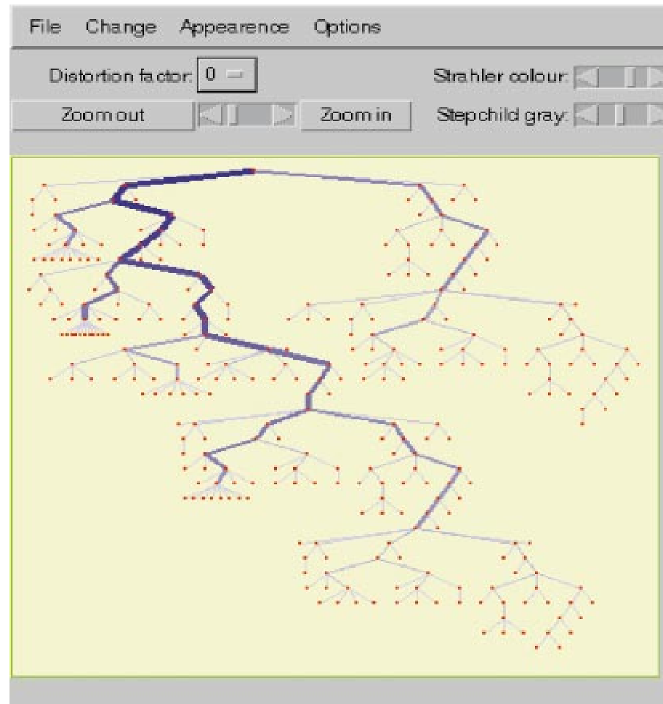

Figure 5: *A full tree coloured with Strahler numbers (301 nodes)*

© The Eurographics Association and Blackwell Publishers Ltd 1998

uration and the available Strahler number ranges‖. (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 4 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 in which direction this should occur (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 the lower half of the picture; which direction should we choose in this subtree? The answers to these questions are apparent if the Strahler based visual clues are used, see Figures 3 and 4. When experimenting with the results, 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 (the tool was used, e.g., to navigate in a tree representing hierarchical file systems, or through internal data structures of compilers); users found it to be a very useful aid for navigation.

Note that, at this stage, we deliberately tried to use simple graphics in defining the visual clues; the goal was information visualisation, the possible target applications may have to rely on simple graphics facilities only such as Java's AWT package. More complex visual clues could be defined and implemented if more elaborate tools were used, such as Java2D, or other, more powerful graphics hardware and packages.

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 (see Figure 5: the tree is not really large, but the size of this page 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 describes two other means to influence the outlook of a tree.

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

- 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 positions, 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 layout of the parent using the weighted average of the children's Strahler numbers.
  Figure 6 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 dependent and should be given as an option to the end–user.
- If 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 arranging the one with a maximum Strahler in the middle, and children with decreasing values symmetric on both sides. Note, however, that some applications may not allow a reordering of the nodes; indeed, the order may contain a 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, hierarchical views are unavoidable as the size grows. 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 detail 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 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 apparently ignore it).

© The Eurographics Association and Blackwell Publishers Ltd 1998

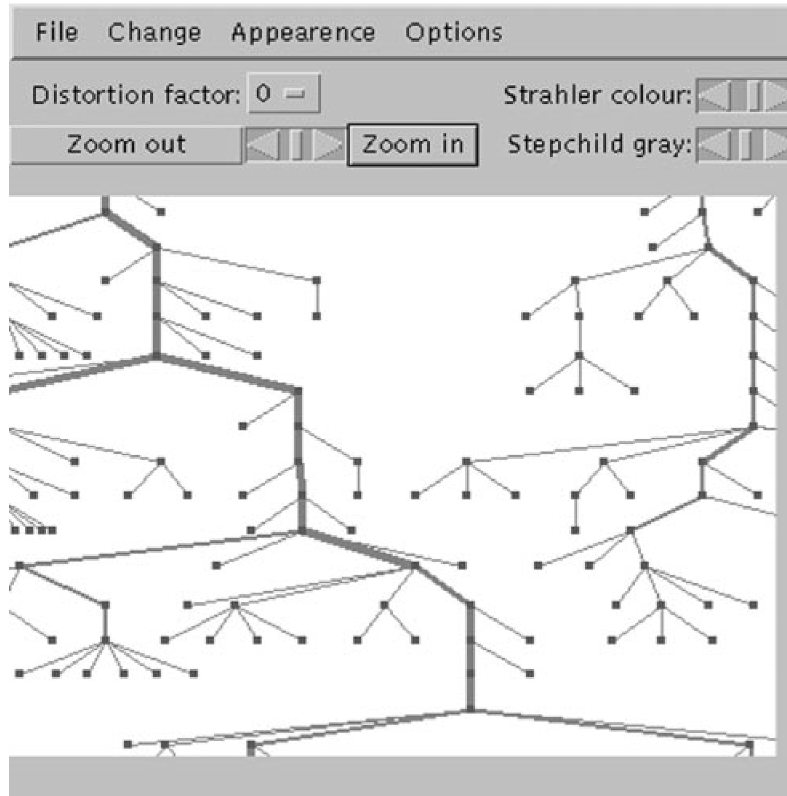

Figure 6: *Tree layout with Strahler skew*

- Although it should be possible to do folding interactively, this should not be the only way to accomplish folding: choosing a series of "foldable" subtrees manually is 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 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, meaning that if the tree is (interactively) modified, the visual effect on the tree itself is local.

Furthermore, folding a subtree (or, later, unfolding 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, propagating this change toward the root. In other words, the recursive layout 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 it is enough to propagate the changes toward the root, instead of reordering the full tree.

Note that, in our interface, the folding/unfolding step (which is accomplished by the user through mouse clicks) is "animated", i.e., the change between the old and the new views is not done instantaneously but through linearly interpolating the positions of all nodes which appear both on the old and the new view of the tree. This very simple trick has proven to be very important in helping the user keeping his/her mental model of the full tree on the screen.

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:

© The Eurographics Association and Blackwell Publishers Ltd 1998

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 area" 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 distribution or, at least, its mean and standard deviation.
   Using classical methods in statistics, we may then calculate the so-called confidence limits α_n and β_n (of the confidence interval $[\alpha_n, \beta_n]$) which define the "normality" range for the value of $f_n(T)$. In other words, if the condition
   $$\alpha_n \leq f_n(T) \leq \beta_n \qquad \text{(EQ 1)}$$
   holds, the tree T has a "normal" geometry (since the probability of $f_n(T)$ of falling within the confidence interval is significantly high), whereas it is abnormal otherwise.
3. 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:
   a. The function $f_n(T)$ is evaluated for each node of the tree.
   b. The tree is traversed in post-order, i.e., the children of each node are examined first. If a child is "abnormal", it is folded and the values of $f_n(T)$ are re-evaluated for the full tree, with the folded subtree now considered 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 during traversal.

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 for 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[7]. 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 layout 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 space on the screen. Consequently, we concentrated on counting the number of leaves in a tree, and we have defined the folding function as (also known in the literature as the "width" of a subtree):

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

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[10]: 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[11]:

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

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

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

This means that if, for a given tree, the value of $f_n(T)$ is unusually large or unusually small the number of leaves is unusually large or, respectively, small, i.e., the (sub)tree has a distorting effect on the display of the tree. What "unusually" large or small must 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 unacceptably 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 approximations:

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

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

give a 95% confidence interval (for large n values). This approximation is based on a much more general (and highly non-trivial) theorem described in a paper by M. Drmota[12]. This paper contains a single theorem; by lucky coincidence, Drmota uses exactly the same function (i.e., the width of a tree) as an example of how his

© The Eurographics Association and Blackwell Publishers Ltd 1998

theorem can be used to deduce that normal distribution approaches f_n (when n goes to infinity). The values for smaller n's can be calculated off-line once, and stored in a table; this completes the algorithm. Having done these calculations in Maple, it turns out that the approximation holds already for n greater than 10 (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.

The code below shows the details of the algorithm in Java. The call to `isNormal` (details of this procedure are not shown) takes care of the statistical evaluation of the n and k values, as well as the halting condition of the algorithm:

```
static class FoldData {
    int n;
    int k;
}
FoldData boxANode(Node nd) {
    FoldData retval = new FoldData();
    if( nd.children.size() == 0 ) {
      // This is a leaf!
      retval.n = 1;retval.k = 1;
    } else {
        // Not a leaf; a recursive call to
        // all children has to be done:
        retval.n = 1;retval.k = 0;
        Enumeration e = nd.children.elements();
        while( e.hasMoreElement() ) {
          FoldData kidData;
          kid = boxANode((Node)e.nextElement());
          retval.n += kidData.n;
          retval.k += kidData.k;
        }
        // This is the real folding step for
        // the subtree. Note that the root of
        // the full tree is treated separately.
        if( nd.parent != null&&
            isNormal(retval) == false ) {
             // Fold the subtee; details
             // are not of interest here
             // (it is an administrative
             // procedure on the tree data):
             nd.foldSubtree();
             // Act as if it were a leaf:
             retval.n = 1;retval.k = 1;
        }
    }
    return retval;
}
```

Figure 7 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 results in very dense 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 light grey nodes (as opposed to the dark grey ones) represent the "folded" subtrees. Note that these light grey 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.) For our Java implementation on an average PC or UNIX Workstation, the folding step for this size is accomplished instantaneously.

The upper right hand corner image is somewhat artificial, because the tree has not yet been "reordered", i.e., the layout algorithm has not yet been used to produce a new image of the tree, so there are large, blank areas on the image. 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 has something been folded, but also whether the folded subtree is complex or not, i.e., whether it is worth further exploration.[**] In other words, the use of the Strahler numbers in the image partially compensates for the loss of information induced by folding, and it helps the user to keep a proper mental image of the full tree.

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 represents a significant change in 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, an 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

[**] 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.

© The Eurographics Association and Blackwell Publishers Ltd 1998

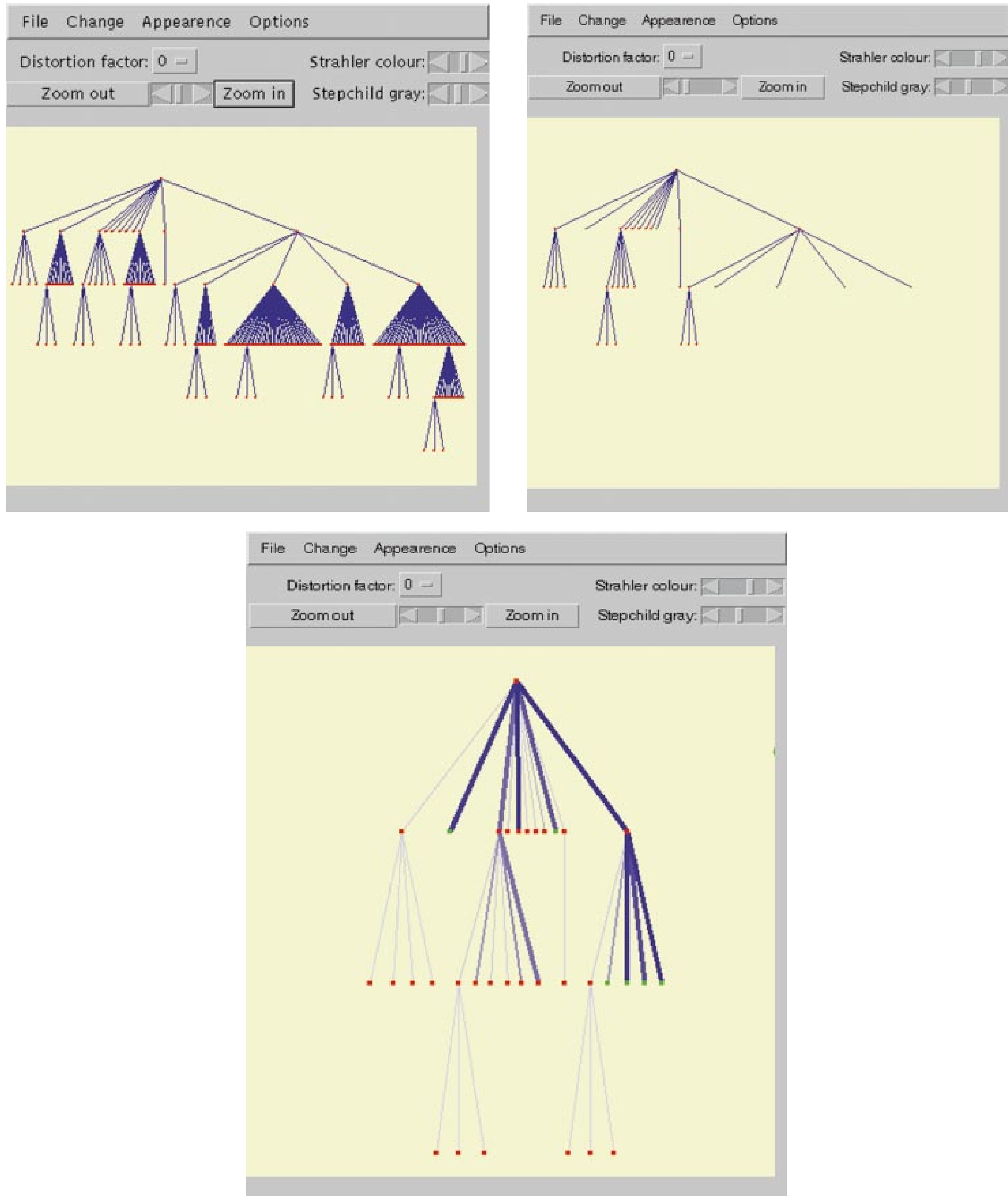

Figure 7: *The effect of the automatic folding algorithm (before and after the folding step; zoom in on the simplified tree)*

© The Eurographics Association and Blackwell Publishers Ltd 1998

added to each node, and the algorithm should take these flags into account when deciding on the folding step.

- The two inequalities in (EQ 1) are not considered separately in the algorithm. In other words, whether abnormality occurs because the value of $f_n(T)$ is smaller than α_n or whether it is greater than β_n does not make any difference for the algorithm. However, in some cases, differentiating between these two cases can be useful. For example, if the window in which the tree display occurs is unusually wide, then 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 Work

The framework using the results described in this paper is being successfully used for various research and development projects. A system has been developed for the company called ACE (Associated Computer Experts), in Amsterdam. It is used to visualise the internal data structures of compilers which have been 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 interesting for the combinatorics community.

There are a number of 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 deduce 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. 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 starting point for deriving good clues for DAGs.
- We emphasised in Section 3 that the visualisation clues are implemented using relatively simple graphics, to abide with 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. The linewidth can be used in such a case to convey some other information.
- The benefit of using 3D techniques in tree/graph visualisation is still an open issue. Robertson et al.[13] have experimented with 3D techniques before and 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[14] 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 interface around the algorithms, including zoom, pan, fish-eye, and other interactive facilities. His remarks throughout the development are very well appreciated. Discussion with Jean-Marc Fedou (Universite de Nice), Jean-Philippe Domenger (LaBRI), and Farhad Arbab (CWI) gave us extremely valuable feedback to our ideas. We are also grateful to the anonymous referees who commented on the first draft of this paper and who, with their comments, helped us to improve its final version.

Part of the work has been financed by a grant of the Franco–Dutch research cooperation programme "Van Gogh".

References

1. di Battista, G., Eades, P., Tamassia, R. and Tollis, I. G. "Algorithms for drawing graphs: an annotated bibliography". In: *Computational Geometry: Theory and Applications*, **4**(5), pp. 235–282, (1994).
2. Misue, K., Eades, P., Lai, W. and Sugiyama, K. "Layout adjustment and the mental map", *Journal*

© The Eurographics Association and Blackwell Publishers Ltd 1998

of Visual Languages and Computing, **6**, pp. 183–210, (1995).

3. Lin Huang, M., Eades, P. and Cohen, R. F. “WebOFDAV – navigating and visualizing the Web on-line with animated context swapping”, *Computer Networks and ISDN Systems (Proceedings of the 7th World Wide Web Conference)*, **30**, pp. 638–642, (1998).
4. Reingold, E. M. and Tilford, J. S. “Tidier drawing of trees”. In: *IEEE Transaction on Software Engineering*, **7**(2), pp. 223–228, (1981).
5. Walker II, J. Q. “A node-positioning algorithm for general trees”. In: *Software – Practice and Experience*, **20**(7), pp. 685–705, (1990).
6. Sarkar, M. and Brown, M. H. “Graphical Fish-eye views of graphs”. In: *Proceedings of the CHI’92 conference*, ACM Press, (1992).
7. Flajolet, P., Raoult, J. C. and Vuillemin, J. “The number of registers required for evaluating arithmetic expressions”. In: *Theor. Comput. Science*, **9**, pp. 99–125, (1979).
8. Viennot, X. G. “Trees”. In: *Melanges offerts a M.P. Schutzenberger*, eds. A. Lascoux and D. Perrin, Hermes, Paris, (1990).
9. Viennot, X. G., Eyrolles, G., Janey, N. and Arquès, D. “Combinatorial analysis of ramified patterns and computer imagery of trees”. In: *Computer Graphics (SIGGRAPH’89)*, **23**, pp. 31–40, (1989).
10. Kreweras, G. and Moszkowski, P. “A new enumerative property for the Narayana numbers”. In: *J. Stat. Plann. Inference*, **14**, pp. 63–67, (1986).
11. Wilf, H. S. *Generating functionology*. Academic Press, Boston – San Diego – New York – London, (1994).
12. Drmota, M. “Systems of functional equations”. In: *J. Random Structures and Algorithms*, **10**(1-2), pp. 103–124, (1997).
13. Robertson, G. G., Card, S. K. and Mackinlay, J. D. “Information visualization using 3D interactive animation”. In: *Communication of the ACM*, **36**(4), pp. 57–71, (1993).
14. Ware, C. and Franck, G. “Evaluation of stereo and motion cues for visualising information nets in three dimensions”. In: *ACM Transaction on Graphics*, **15**(2), pp. 121–140, (1996).

© The Eurographics Association and Blackwell Publishers Ltd 1998