Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Circular Drawings of Rooted Trees

G. Melanc con and I. Herman

Information Systems (INS)

INS-R9817 December 1998

Report INS-R9817
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

Circular Drawings of Rooted Trees

G. Melançon and I. Herman
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{*Guy.Melancon, Ivan.Herman*}*@cwi.nl*

ABSTRACT
We describe an algorithm producing circular layouts for trees, that is drawings, where subtrees of a node lie within circles, and these circles are themselves placed on the circumference of a circle. The complexity and methodology of our algorithm compares to Reingold and Tilford's algorithm for trees [11]. Moreover, the algorithm naturally admits distortion transformations of the layout. This, added to its low complexity, makes it very well suited to be used in an interactive environment.

1991 Computing Reviews Classification System: D.2.2, E.1, G.2.1, G.2.2
Keywords and Phrases: Rooted trees, graph drawing, information visualization
Note: At CWI, the work was carried under the project INS3.2 "Information Visualization".

1. INTRODUCTION

Rooted trees are at the center of many problems and applications in computer science. Information systems, multimedia documents databases, or virtual reality scene descriptions are only a few examples in which they are used. Their widespread use is most probably the result of the fact that they capture and reflect the way humans often organize information. A visual representation of these structures is often a major tool to help the user finding his/her way in exploring data; hence the importance of graph drawing and exploration in information visualization.

Methods for drawing trees has received a significant attention for a long time and it still reappears as an intermediate task in many applications. The classic algorithm by Reingold and Tilford [11], improved later in [14], gives a very effective solution to produce a classical, top–down drawing of rooted trees. Eades [4] also described an alternative, so called *radial* display of a tree, based on earlier results described, for instance, in [2]. However, the need for suitable tree representations for large amounts of data still simulates work in finding alternative layouts. For example, there has been an increased interest recently in using non–euclidean geometries [6, 8, 1, 9, 10], or 3D representations [12, 7].

The classical, top–down drawing of trees has the advantage of being well–known and widely used in many applications. Its use for displaying hierarchies benefits from its natural interpretation. In some cases, however, one might need to deal with representations where the hierarchical organization of data is much less relevant. As an example, we encountered situations (e.g., in exploring keyword and thesauri data) where the hierarchy was not natural, although the underlying data structure was indeed a tree. Eades [4] cites some other examples for what he calls "free trees". Data in all these applications can of course be described through rooted trees, but only artificially; in fact, exploration in these applications consist in "changing" a root. In our case, our users felt that the classical top–down drawing strongly suggested a hierarchy, and this view interfered with mental model of the relevant data exploration.

Our search for an alternative tree representation originated from this user demand. The algorithm we present here yields a circular view of trees. The drawings are very intu-

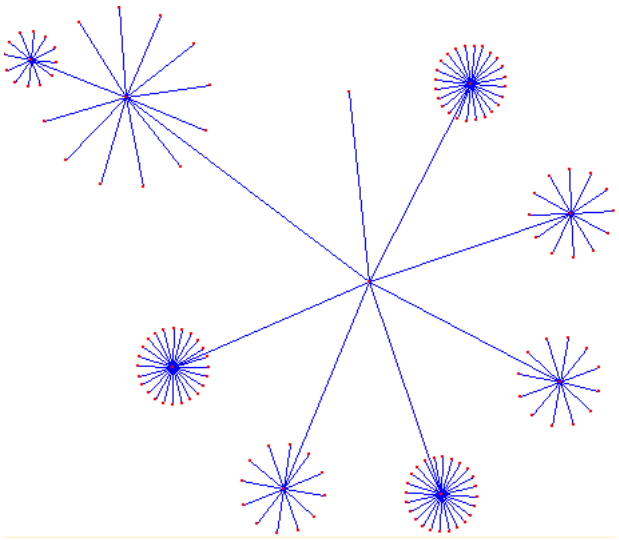

Figure 1: Circular positioning of a tree

itive: every node is placed at the center of a circle, and the subtrees attached to a node, themselves drawn into circles of smaller radii, are then placed on the circumference of the large circle. This process is repeated recursively. Although the idea is simple, we have found no description of similar algorithms in the literature. The drawings look familiar, though; indeed, we have observed circular displays of simple trees in some demonstration applications, but none mentioned the use of any general algorithms, capable of dealing with arbitrary trees. Another advantage of our algorithm is that it makes it easy to apply distortion on the drawing, offering a natural alternative to the well–known fish–eye view of graphs [13]. Indeed, each node stores a scaling factor to apply on its own radius; by changing this value interactively one may control a distortion on the subtree.

The rest of the paper is organized as follows. The first section describes the main lines of our algorithm. For the sake of clarity, the description ignores certain details and postpones their discussion to later sections. The examples presented at first serve both for comparison with a classical top–down display of a tree, as well as a basis for the improvements we introduce in section 3. The drawings are computed in linear time; this, and some other issues in comparing our algorithm to the classical Reingold and Tilford's algorithm [11], are discussed in remark 2.4. A section describing the distortion effect obtained by changing a node's scaling factor concludes the paper.

2. Basic positioning algorithm

The drawings generated by our algorithm are very intuitive. Every node is placed at the center of a circle and the subtrees of the node, themselves drawn into circles of smaller radius, are placed on the circumference. Figure 1 shows a typical circular display for a simple tree.

As said before, there is no explicit mention for such layout algorithm in the literature, although the idea of such a display seems very natural. The reason is probably that the superficial description of the algorithm is very simple and, consequently, one might think that the implementation is not much more complicated either. This is misleading, though; in reality, the implementation of the algorithm becomes quite complex. Undoubtedly, attempts have been made to develop similar layout methods; we have observed circular displays of trees in various demo applications[1]. However, the usual examples show very well balanced trees for which the display is optimal. Some other examples use interaction to stick to

[1] See, for example, `http://www.merzcom.com` and

balanced trees, and/or they restrict the visible portion of the tree to a given depth.

Our algorithm uses recursive traversal of the tree to compute the position of a node; it proceeds in two steps. The first step is bottom–up and computes position of a node relative to its ancestor, along with scaling factors to be applied during the second pass. The second, top–down traversal cumulates these scaling factors and computes the absolute x-y coordinates of the nodes.

Let us first describe the parameters driving the behaviour of the algorithm.

Notation. Let n be a node. Denote by $k_1, \ldots, k_p$ the subtrees attached to the children nodes of n. Suppose, by induction, that each subtree k_j needs to be drawn in a circle C_j of radius r_j. The distance between n and the center of the circle C_j will be referred to as the distance from n to a circle C_j (i.e., the subtree k_j). Incidentally, we shall identify the node k_j with the subtree it induces, as well as with the circle C_j containing it.

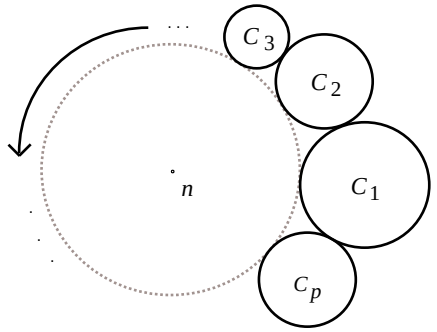

Figure 2: Positioning of children's circles along the circumference

The scheme we adopt is the following. Each circle C_j is drawn at a distance $d + r_j$ from n, where $d = d_n$ is constant for all $k_1, \ldots, k_p$ (see Figure 2). The value for d has to be decided based on some heuristics: our experimentation has shown that $d = \max\{r_1, \ldots, r_p\}$ suits well. Other choices for d give different displays; section 3 discusses a more dynamic choice for the value of d.

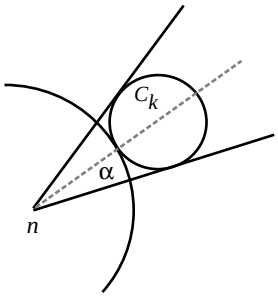

Figure 3: Halfsector α associated with a children node k

Definition 2.1 *Let us define the* sector *of a subtree C_j to be the angle formed by two rays originating at n and tangent to two diametrically opposed point on C_j (cf figure 4). That is, consider the angle $\alpha_j = \arctan(\frac{r_j}{d+r_j})$. Then, the sector associated with C_j is $2\alpha_j$. We shall refer to the angle α_j as the* halfsector *of C_j. Obviously, the sector (and halfsector) is uniquely defined.*

Hence, three values are associated with each node n:

1. a (constant) distance d used to position each child's circle;
2. the radius r of the circle containing the full subtree induced by n (i.e., all the recursive circles); and
3. an angle α, which is equal to the halfsector formed by the circle n, relative to its ancestor node.

If the halfsector value for each child is already calculated, one may compute the total sum of these halfsectors, i.e., $\sum_j \alpha_j$, to evaluate whether all C_j circles fit within the 2π angle which is at disposal for n. There are obviously, two cases:

- $\sum_j \alpha_j \leq \pi$ and
- $\sum_j \alpha_j > \pi$

(Note that we use halfsectors, hence the comparison with π instead of 2π.) In the first case, we may decide to distribute the free space left between each circles evenly; this is the simple case. The second case necessitates some more work, because each circle has to be scaled down in order to fit them into the available space. I.e., we must compute a scaling factor for each circle C_j, to readjust the total $\sum_j \alpha_j$ to π. This can be acheived by computing a common scaling factor $c = c_n$; choosing $c = \pi/(\sum_j \alpha_j)$, and taking $\alpha'_j = c \cdot \alpha_j$ for each halfsector will fulfill the conditions. The new radii for the circles will be:

$$r'_j = d \cdot \frac{\tan(\alpha'_j)}{1 - \tan(\alpha'_j)} \tag{2.1}$$

With these formulae at hand, the first (simple) version of the positioning algorithm is as follows (for sake of simplicity, we shall refer to the values associated to the node n using the dot notation as: $n.d$, $n.\alpha$ and $n.r$). The algorithm itself consist of two distinct steps:

```
algorithm circularDisplay(node n) {
    firstWalk(n)
    secondWalk(n, ... )
}
```

(The meaning of the empty arguments " ... " will become clear as the discussion evolves.) The first step proceeds bottom up, computes a value for $d = d_n$, possibly calculates a scaling factor for the children of a node in function of the value of $\sum_j \alpha_j$, and concludes by computing the radius of the circle C_n so as to contain the full subtree at n:

```
method firstWalk(node n) {
    n.d = 0
    loop over children k of n {
       firstWalk(k)
       n.d = max(n.d, k.r)
    }
    adjustChildren(n)
    setRadius(n)
}
```

Note that the value d need not be computed for leaves and that its zero value shall not affect the positioning of the nodes.

The work carried out by `adjustChildren` is the one described before, depending on the comparison of $\sum_j \alpha_j$ and π. It results in the scaling factor $n.c$, which is to applied to every children's radius, and a value $n.f$ measuring the free angle to be eventually distributed among all children.

```
method adjustChildren(node n) {
    compute s = ∑_j α_j
    if (s > π) {
       set n.c = π/s and n.f = 0
    }
    else {
       set n.c = 1 and n.f = π − s
    }
}
```

The factor $n.c$ will be used in the second pass to adjust the radii and the halfsectors of the descendents' circles. Finally, the value $n.f$ will be used when computing the absolute x–y coordinates of the node.

This scheme makes the `setRadius` prodedure rather simple. Indeed, the node's radius is three times the maximum value of the children's radii, except when a node is a leaf in which case we must assign it a minimum radius m. This maximum value has already been computed in `firstWalk`, i.e.:

```
method setRadius(node n) {
    n.r = max(n.d, m) + 2 n.d
}
```

(Actually, we may want to slightly augment this value in order to show neighbour circles apart by a minimal distance.)

The method `secondWalk` computes the absolute x–y coordinates for each node. This method is invoked with a number of parameters, which are defined as follows. Because we want to place the root of the tree at the origin, the convention is that the node n is assigned its (x, y) coordinates by the caller, i.e., when `secondWalk` is invoked. The node should also receive a scaling factor λ which it must apply to its radius; this is achieved by applying it to $n.d$ and, recursively, to all its descendents. Finally, remember that data calculated during `firstWalk` are all relative to local x-y axis; the node must receive an angle θ which is its angle with the absolute x–axis. All points lying inside its circle must be rotated accordingly. Hence, the method `secondWalk` has the prototype:

```
secondWalk(node n, double x, double y, double c, double θ)
```

The call to `secondWalk` on the root of a tree uses the values $x = y = 0$, $c = 1$, and $\theta = 0$, i.e., is positioned at the origin and its children are placed counterclockwise starting from the positive x–axis. Remember that the scaling factor $n.c$ must be applied to every children's halfsectors, their radii must be adjusted accordingly, and that the free angle $n.f$ has to be evenly distributed among all children nodes.

```
method secondWalk(node n, double x, double y, double λ, double θ) {
    store x and y for node n, i.e.  n.x = x, n.y = y
    set d′ to λ · n.d

    set φ = θ + π
    set freespace = n.f / (number of kids+1)
    set previous = 0
    loop over all children k {
        set α′ to c · k.α
        set r′ to d · tan(α′)/(1−tan(α′))  (cf Eq. (2.1))
        φ = φ + previous + k.α + freespace
        k.x = (λ · r′ + d′) · cos(φ)
        k.y = (λ · r′ + d′) · sin(φ)
        previous = k.α

        secondWalk(k, k.x + n.x, k.y + n.y, λ · r′/k.r, φ)
    }
}
```

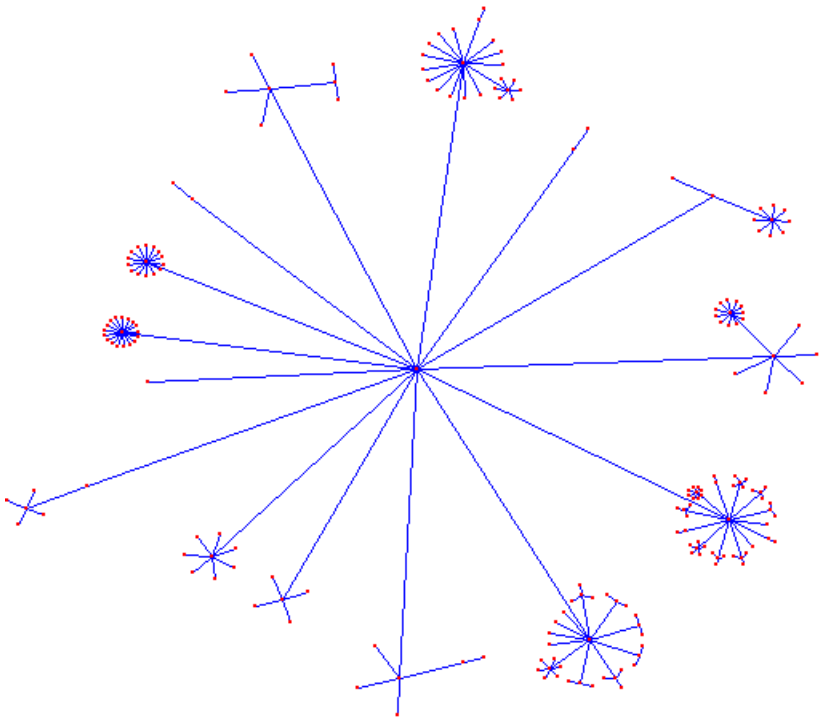

Figure 4: Circular positioning of a more complex tree

Figure 4 shows another example of a circular display for a more complex tree.

2.1 Discussions on the simple algorithm

The algorithm, as described above, calls for a certain number of remarks.

Remark 2.2 There are several points in the algorithm where somewhat arbitrary choices have been made. For example, a slightly more elaborate way of solving the $s = \sum_j \alpha_j < \pi$ case could have been chosen, by using values proportional to the children's sizes; the

choice of the distance between a node and its child's circle could be slightly different, etc. Although these are all possible variants of the algorithm, their overall effect is not significant, and it is not of a real interest to go into all these kinds of details here.

Remark 2.3 The edge length decreases exponentially when going deeper into the tree. This can already be observed in Figures 1 and 4. As a consequence, a deeply hidden subtree will look as a dense set of points and will not clearly show its structure. This phenomenon had to be expected since the area of a circle only offers linear space (with respect to the circle's radius) where a (potentially) exponential number of informations must be placed. This problem also arises when drawing in non–euclidean geometries using the Poincaré model, for example[2]. An alternative, when dealing with deeper subtrees, could be to show only its points without drawing the edges. This has already been implemented elsewhere [15] in a different context, where it seems to give satifying results. Section 4 proposes another technique to overcome this problem by interacting with the algorithm directly.

Remark 2.4 The complexity of the algorithm is linear in the number of nodes of the tree, just like the classical, hierarchical placement algorithms. There is, however, a closer parallel to be drawn between this circular layout and Reingold and Tilford's (R&T) algorithm [11]; the structure of both methods are indeed quite similar.

As in our case, R&T computes the positions of the nodes in two passes. In the first, bottom-up pass, our algorithm calculates and stores the fields $n.c$ and $n.f$; these values are used in the second, top–down traversal, as a planar transformation on the subtree, resulting in the absolute x–y coordinates of the nodes. In the case of R&T, the positioning of a node boils down to the computation of its x value, since the y ordinate is given by the node's depth. On the first traversal, R&T computes the relative position of a node (as the root of a subtree) starting from the leaves. On its way up, the algorithm might deduce that a child's subtree will have to be translated horizontally. It does not perform this translation at that moment since it might well have to be composed with translations still to be discovered; the value is stored in a field and used during the second traversal of the tree. This "lazy transformation" is precisely what is done by our algorithm, too: a child k might have to adjust its radius if the $\sum_j \alpha_j > \pi$ occurs. However, instead of performing the transformation, the scaling factor is stored, and is combined later with its parent's scaling factor λ.

The main difference with the R&T algorithm is the following. In R&T, the range of x values for nodes has no constraint except for a minimum x distance between nodes. Our case is conditionned by the fact that the sum must satisfy $\sum_j \alpha_j \leq \pi$. This would be equivalent in R&T to ask the x values to lie within a given interval. The algorithm would then have to adjust the width of subtrees to make sure the whole tree fits in the given interval.

3. TIDIER CIRCULAR POSITIONING

This section discusses an improvement of the algorithm described in the preceding section; the goal is to make a more optimal use of the available space, while still keeping the complexity within linear bounds (staying within linear complexity was a critical aspect since we use the circular layout within an interactive application).

To understand the problem, let us first go back to the original algorithm. A node n is positioned along a ray stemming from its father. Its first child, k_1, is placed tangentially to this line. This first child generates another ray, defining the sector rooted at n and tangent to C_1. The second child, k_2, is then placed tangent to that ray associated with k_1, and so on (see Figure 5). This observation might have escaped the reader's attention, since one would naturally expect to place a child next to its left brother, letting it "slide" until both

[2] See, for example, the hyperbolic tree visualizer at http://www.inxight.com/

circumferences touch at a single tangent point. This is *not* what happens in our case, since a sector's ray might well keep two neighbour circles from touching.

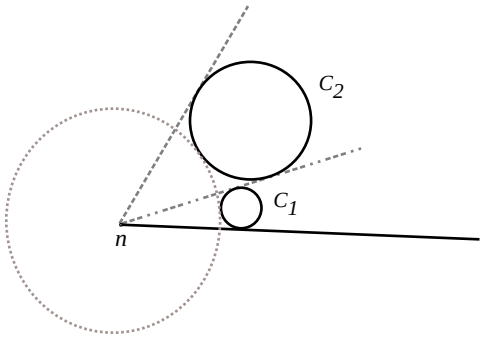

Figure 5: Neighbour circles rest on halfsector rays

Obviously, the use of rays to order and separate neighbouring circles does not optimize the use of space. The optimal algorithm would be to let a circle "slide" in the direction of the circles which are already positioned, until it "bumps" into another circle, or into the ray connecting a circle to its father. Observe that the circle being bumped into would not necessarily be the immediate left brother: it would be possible to have any number of circles "hidden" below two circles with larger radii (Figure 6). Furthermore, the search for optimums should not only consider the option of effectively placing circles as close as possible to one another, but should also allow the distance between a node and its children to vary. Indeed, one could, for example, expect leaves to be closer to their father.

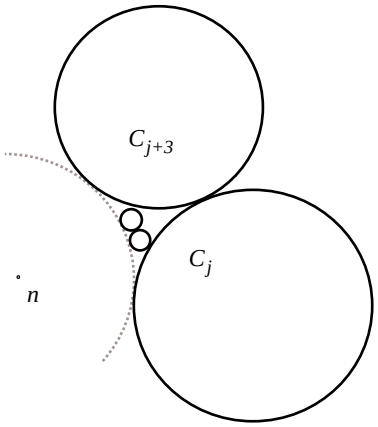

Figure 6: When sliding, smaller circles can be hidden by larger ones

Our analysis and experiments lead us to the conclusion that any optimal solution along these lines would be too complex for our purposes. Even naive attempts to compute a less than optimal positioning turn into optimization problems dealing with huge sets of constraints. A solution could be to use a force–based method built on top of a properly defined physical model. The solution could then be an equilibrium point, reached by a set of circles "fighting" against one another. Similar, optimization based algorithms for graph layout do exist (see, e.g., the overview of di Battista et al. [3]), but they are rarely used for trees; their complexities clearly go beyond linearity. However, there is a possibility to improve the simple algorithm, while still remaining within the linear complexity domain; this will be presented in the rest of this section.

The solution described in the simple version of the algorithm is very conservative. When the children's circles have been positioned, it places the node n at the center of its circle and the radius is set to a large enough value so that the circle would contain all the circles associated to the children. (This is achieved by defining the radius to be three times that of the children's maximal radius, cf p. 5.) Observe however that, for example, when a node has a single child, this step produces a circle that is about twice as big as necessary: a circle containing both the n and its child could well be centered at their midpoint (see Figure 7). This will have a cumulative effect on the tree layout as a whole, because the size of the circles around the leaves will influence, recursively, the circular arrangement of all subtrees.

In general, space can be saved by a modification of the original algorithm, placing a subtree into a circle centered at the *barycenter* of all children of node n. The radius of this circle can then be computed so that it contains node n as well as all circles associated to its children nodes.

The barycenter of points $P_q = (x_1, y_1), \ldots, P_q = (x_q, y_q)$ is $B = (\sum_j x_j/q, \sum y_j/q)$. The method `setRadius` should be modified as follows:

```
method setRadius(node n) {
    compute local x-y coordinates for all children
       (ignoring scaling factor n.c
       but taking freespace n.f into account)
    compute the barycenter B = (b_x, b_y) of those points
    store the node's relative coordinates with origin placed at B
       i.e.  n.rel_x = −b_x, n.rel_y = −b_y
    set radius to max_j d(B, k_j) + r_j
}
```

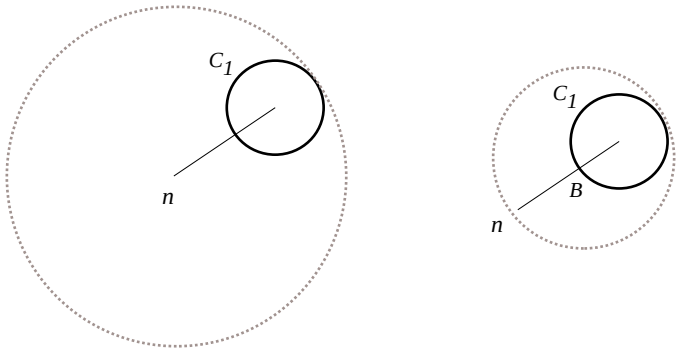

Figure 7: Containing circles: basic algorithm (left); tidier algorithm (right)

The coordinates of the barycenter B are computed with the node n being placed at the origin of the co–ordinate system. However, to have a more natural look, it is the barycenter which should be placed into the origin, and not n. This is achieved by computing the coordinates of n relative to B before leaving the method.

In the basic version of the algorithm, there was also a rotation step around n. This was taken care of by a simple instruction (in `secondWalk`), namely:

$$\texttt{set } \varphi = \theta + \pi$$

In this new version of the algorithm, however, the rotation should be performed around the barycenter, rather than around n. This means that the children's positions, computed relative to n, should be translated before applying a rotation around B (this also applies to the node n). This leads to a modification of `secondWalk`, too. The input arguments of the modified version should receive the barycenter's absolute x–y coordinates; the relative

coordinates $n.rel_x$ and $n.rel_y$ (computed in the modified version of `setRadious`), as well as the angle θ, must then be used to compute the absolute x–y values. Here is the new version of the method `secondWalk`:

```
method secondWalk(node n, double b_x, double b_y, double λ, double θ) {
    store x and y for node n, i.e.
        n.x = b_x + λ · (n.rel_x cos(θ) − n.rel_y sin(θ)),
        n.y = b_y + λ · (n.rel_x sin(θ) + n.rel_y cos(θ))
    set φ = π
    set freespace = n.f / (number of kids+1)
    set previous = 0
    loop over all children k {
        set α′ to c · k.α
        and r′ to d · tan(α′)/(1−tan(α′))
        φ = φ + previous + k.α + freespace
        # compute coordinates for k relative to n
        k.x = (λ · r′ + d′) · cos(φ)
        k.y = (λ · r′ + d′) · sin(φ)

        # translate to B
        [ k.x ]   [ k.x ]   [ n.rel_x ]
        [ k.y ] = [ k.y ] + [ n.rel_y ]

        # and then rotate by θ
        [ k.x ]   [ cos(θ)  −sin(θ) ] [ k.x ]
        [ k.y ] = [ sin(θ)   cos(θ) ] [ k.y ]

        previous = k.α
        secondWalk(k, k.x + b_x, k.y + b_y, λ · r′/k.r, φ)
    }
}
```

Obviously, the modified version of the algorithm is still linear; the added complexity, caused by the calculation of the barycenter, and the translations, are negligible. Figure 7 shows the same tree as in Figure 4, but positioned by the modified, tidier version of our algorithm.

4. Interacting with the algorithm

When dealing with a large number of nodes, any layout algorithm will need to be complemented by navigation techniques to help the user in understanding and/or exploring the data set. One basic technique that has been successfully used is the fish–eye transformation [5], originally introduced in [13]. The idea is to give a more detailed view of a small part of a graph while maintaining the entire data set in sight, too (as opposed, for example, to a zooming effect which displays a part of the graph only). The more general term "focus+context" has also been used in the literature to describe these techniques [8].

Though being very useful, fish–eye views have also their drawbacks. The essence of a fish–eye view is to distort the position of each node, using a concave function applied on the distance between a local point and the node's position. If the distortion were to be applied faithfully, the edges connecting the nodes should be distorted, too. This would result, in general, in a complicated curve, whose display on the screen might be prohibitively slow if a large number of nodes are involved. Consequently, implementers are forced to transform the nodes' positions only, and connect these transformed nodes with line segments again.

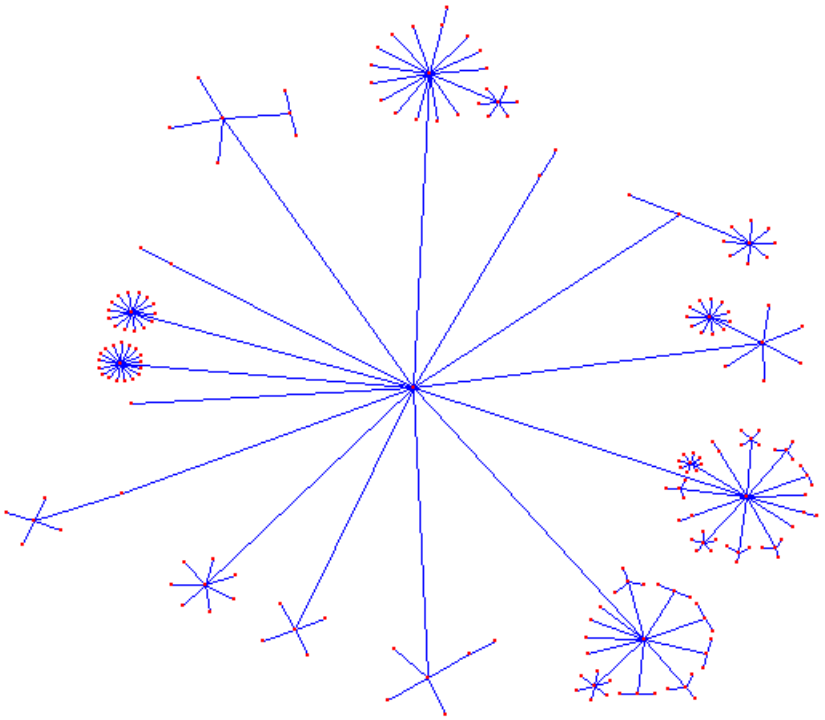

Figure 8: Modified circular positioning of the same tree as in Figure 4

The problem is that this may result in intersecting edges, thereby reducing the quality of display of the tree.

The advantage of our circular layout algorithm is that it offers a natural exploration view without the drawbacks described above. Indeed, an obvious focus+context approach in a circular layout is to simply *inflate* the circle assigned to a node; this inflation can be done under user control. Our algorithm is very well adapted for such control. By assigning a scaling factor to each node, the radius for the circle can be trivially modified; the effect of this modification will influence the rest of the layout automatically. It is easy to add an interactive control to any application which would allow the user to change this scaling factor (either by inflating or by deflating the corresponding circle); by simply re–running the layout algorithm the new view can be generated easily. Furthermore, and in contrast to the usual fish–eye approaches, it does not make it any more complicated to offer a multi–focus exploration possibility, too: simply allow for the modification of the scaling factors for more than one nodes. Figure 9 shows the effect of expanding a subtree (on the bottom right) to get a more detailed view of it.

Another navigation technique we implemented, which was also part of our initial motivations for the circular layout, is the re–root facility. This means that the user can interactively select a node, the internal structure of the graph is re–arranged so that this node becomes the new root, and the layout algorithm is re–run. The transition between the old and the new "view" is performed through a smooth animation. As predicted, the usage of a circular view gives a very natural setting for such re–root operation; the mental model of the underlying data–set is well preserved. In our experience, the combinations of the inflating/deflating actions on a subtree and the re–rooting facility offer a powerful tools to explore large amount of data.

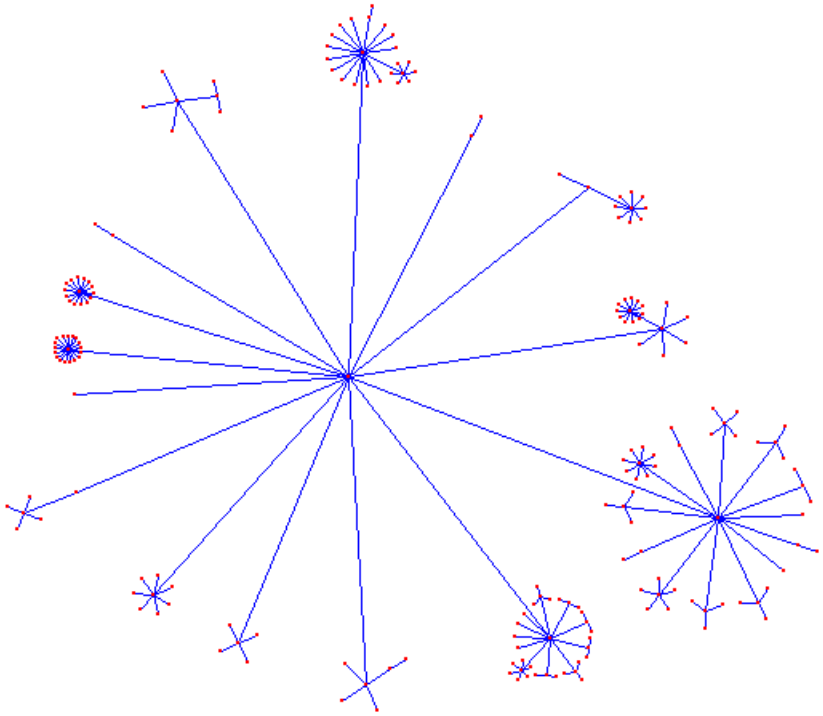

Figure 9: Expanding a subtree (increasing its radius by a factor $\lambda = 2$). Compare with Figure 8.

References

1. Burchard P. and Munzner T. Visualizing the structure of the World Wide Web in 3D Hyperbolic Space. In *Proceedings of the VRML'95 Symposium, ACM Siggraph*. ACM Press, 1995.
2. Read R. C. Methods for computer display and manipulation of graphs, and the corresponding algorithms. *Congressus Numerantium*, 63:49 – 88, 1988.
3. Di Battista G., Eades P., Tamassia R., and Tollis I.G. Algorithms for Drawing Graphs: an Annotated Bibliography. *Computational Geometry*, 4:235 – 282, 1994.
4. Eades P. Drawing Free Trees. *Bulletin of the Institute for Combinatorics and its Applications*, 5:10 – 36, 1992.
5. Formella A. and Keller J. Generalized fisheye views. In *Graph drawing '95*, volume 1027 of *Lecture Notes on Computer Science*, pages 243 – 253. Springer Verlag, 1995.
6. Gunn C. *Computer Graphics and Mathematics*, chapter Visualizing Hyperbolic Space. Focus on Computer Graphics series. Springer-Verlag, Berlin, Heidelberg, New York, 1992.
7. Jeong C.-S. and Pang A. Reconfigurable disc trees for visualizing large hierarchical information space. In G. Wills, editor, *Proceedings of the IEEE Symposium on Information Visualization (InfoVis'98)*. IEEE CS Press, 1998.
8. Lamping J., Rao R., and Pirolli P. A focus+context technique based on hyperbolic geometry for viewing large hierarchies. In *ACM CHI'95*. ACM Press, 1995.
9. Munzner T. H3: Laying out Large Directed Graphs in 3D Hyperbolic Space. In *Proceedings of the 1997 IEEE Symposium on Information Visualization*. IEEE CS Press, 1997.
10. Munzner T. Exploring Large Graphs in 3D Hyperbolic Space. *IEEE Computer Graphics & Applications*, pages 18 – 23, July/August 1998.
11. Reingold E.M. and Tilford J.S. Tidier Drawing of Trees. *IEEE Transactions on Software Engineering*, 7(2):223 – 228, 1981.
12. Robertson G., Mackinlay J., and Card S. Cone Trees: Animated 3D Visualization of Hierarchical Information. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 189 – 194. ACM, April 1991.
13. Sarkar M and Brown M.H. Graphical fisheye views. *Communication of the ACM*, 37(12):73–84, 1994.
14. Walker J.Q. A Node-Positioning Algorithm for General Trees. *Software: Practice and Experience*, 20(7):685 – 705, 1990.
15. Wills G. J. Nicheworks - interactive visualization of large graphs. In *Graph drawing '97*, volume 1353 of *Lecture Notes on Computer Science*, pages 403 – 414. Springer Verlag, 1997.