Drawing Non-layered Tidy Trees in Linear Time

A.J. van der Ploeg

ploeg@cwi.nl
Centrum Wiskunde & Informatica, Amsterdam

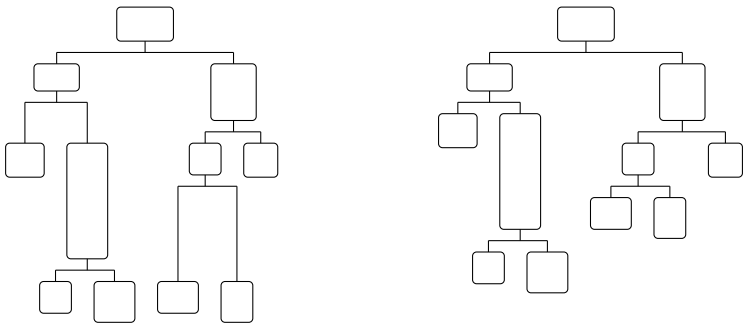
August 5, 2013

Abstract

The well-known Reingold-Tilford algorithm produces tidy *layered* drawings of trees: drawings where all nodes at the same depth are vertically aligned. However, when nodes have varying heights, layered drawing may use more vertical space than necessary. A *non-layered* drawing of a tree places children at a fixed distance from the parent, thereby giving a more vertically compact drawing. Moreover, non-layered drawings can also be used to draw trees where the vertical position of each node is given, by adding dummy nodes. In this paper we present the first linear time algorithm for producing non-layered drawings. Our algorithm is a modification of the Reingold-Tilford algorithm, but the original complexity proof of the Reingold-Tilford algorithm uses an invariant that does not hold for the non-layered case. We give an alternative proof of the algorithm and its extension to non-layered drawings. To improve drawings of trees of unbounded degree, extensions to the Reingold-Tilford algorithm have been proposed. These extensions also work in the non-layered case, but we show that they then cause a $O(n^2)$ run-time. We then propose a modification to these extensions that restores the $O(n)$ run-time.

1 Introduction

In many fields, trees are a much used abstraction. The understanding of trees is greatly improved by visualizing them and hence many types of tree drawings have been proposed [1, 2, 3]. In this paper, we focus on classical node-link diagrams, an example of which is shown in Figure 1. Usually, we are mainly interested in showing the structure of the tree and hence all the nodes can have the same width and height as shown in Figure 1. Sometimes, we also want to show some information inside each node or in the dimensions of each node. Examples of this are Tableau style proof trees, parse trees of (formal) languages, class diagrams in software engineering and Polymetric views [4]. The latter are inheritance diagrams of software systems where the width and height of each node signifies a software metric of the corresponding class, such as the lines of



(a) Layered. (b) Non-layered.

Figure 2: Layered and non-layered tidy drawings of the same tree.

code or number of methods. In these situations, the width and height of each node may vary.

Trees with nodes of varying dimensions can be drawn such that all nodes at the same depth are vertically aligned, which we call *layered* drawings, or nodes can be placed vertically at at fixed distance from each other, which we call *non-layered* drawings. The difference between a layered and a non-layered drawing can be seen in Figure 2. Both types of drawings have their own merits: layered drawings make it easy to compare the depth of nodes, whereas non-layered drawings are vertically more compact.

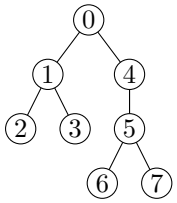


Figure 1: Example node-link diagram of a tree.

Using a simple trick which we introduce later, non-layered drawings can also be used to show an attribute of each node in its vertical coordinate. An example of this is a family tree diagram where the vertical coordinate top coordinate of a node signifies the birth year of the corresponding person, as shown in Figure 3. An example in biology is a diagram which shows evolutionary relationships between biological species and the time in which each species came into existence. Another example is a cell division diagram where the vertical coordinate of each cell indicates the time when its parent cell divided.

A layered drawing of a tree can be found in $O(n)$, where n is the number of nodes in the tree, using the well-known Reingold-Tilford [5] algorithm. Various algorithms [6, 7, 8, 9, 10] have been proposed for the non-layered case, but all of these either make simplifying assumptions or have not been proven to run in linear time.

In this paper, we extend the Reingold-Tiford algorithm such that it also works for non-layered drawings. The original complexity proof of the algorithm for layered trees makes use of an invariant that does not hold for the non-layered case. We give an alternative proof that does not use this invariant, and show how it is adopted to prove that the extended Reingold-Tilford algorithm for the

non-layered case also runs in $O(n)$.

To improve drawings of trees of unbounded degree, an extension to the Reingold-Tilford algorithm has been proposed [11, 12]. This extension also applies to non-layered trees, but we show that they then cause a $O(n^2)$ run-time. We then present a modification to these techniques and prove that this modification restores the $O(n)$ run-time.

Our contributions can be summarized as follows:

- An extension of the Reingold-Tilford algorithm such that it can also produce non-layered drawings.
- A proof of the linear run-time of the Reingold-Tilford algorithm with this extension.
- A proof that the extension for trees of unbounded degree causes a $O(n^2)$ run-time in the non-layered case.
- A modification to this extension and proof that it restores $O(n)$ run-time.

The rest of this paper is organized as follows; We first reformulate the tidy tree drawing problem to include non-layered drawings in Section 2. We then give an overview of the Reingold-Tilford algorithm and introduce its extension for non-layered drawings in Section 3. In Section 4, we prove that the extended Reingold-Tilford algorithm runs in linear time. We discuss the known extension (for layered trees) which improves the drawings of trees of unbounded degree in Section 5. Afterwards, we show that these techniques lead to a $O(n^2)$ run-time in the non-layered case and propose a modification to restore the $O(n)$ run-time in Section 6. We show measurements of the speed of this algorithm in Section 7. Finally, in Section 8 we discuss the history of this algorithm and related work. For sake of completeness, we discuss the techniques in the Reingold-Tilford algorithm which are also applicable in the non-layered case in Appendix A. In Appendix B, we discuss the details of the parts of the techniques to improve drawings of unbounded degree which are also applicable in the non-layered case. In the Appendix C we list the complete source code of the algorithm with the extensions discussed and proposed here.

2 Redefining the Tidy Tree Problem

In this section we reformulate the tidy tree drawing problem to include non-layered drawings. In order to reformulate cleanly, we abstract away from spacing between nodes and drawing connecting lines. The spacing between nodes is added by adding a gap to the widths and heights of the nodes. For example, the solid boxes in Figure 5(f) show the original widths and heights and the dashed boxes show the widths and heights after adding the gap.

Since we abstract away from spacing, in the layered setting, all nodes at the same depth can be considered to be of the same height. The input tree is then a

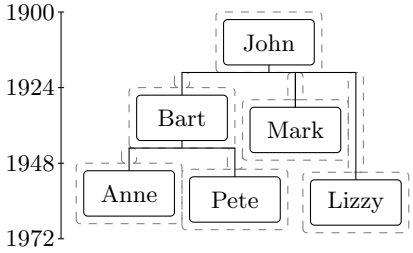


Figure 3: Descendants of John: layout with prescribed vertical positions (corresponding to birth year).

rooted, ordered tree with a width for each node and a height for each depth in the tree. The vertical coordinate of each node is simply the vertical coordinate of its depth. In our more general non-layered setting, an input tree is also a rooted, ordered tree, but with a width and height for each node. Since we abstract away from spacing, the vertical top position of a node is then the bottom coordinate of its parent, which in turn is its top coordinate plus its height. In the rest of this paper, we assume that the vertical positions of the input nodes have already been calculated in this way. Notice that if all the nodes at the same depth are of the same height, then a non-layered drawing is the same as a layered drawing.

Sometimes we have an input tree where the top coordinate of each child is not equal to the bottom coordinate of its parent, as is the case in the trees and in Figure 2(a) and Figure 3. We can transform such invalid trees to valid trees by adding thin "dummy" nodes between parent and child, as shown in Figure 4.

The tidy tree drawing problem is then reformulated as follows: given an input tree, produce a horizontal coordinate for each node such that drawing is compact[1] and the following aesthetic criteria are met [5, 12]:

1. Nodes do not overlap.
2. Children are positioned horizontally in the order given in the tree.
3. Parents are centered above their children.
4. The drawing of a subtree does not depend on its position in the tree, i.e., identical subtrees are drawn identically.
5. The drawing of the reflection of a tree, i.e. the order of the children of each parent is reversed,

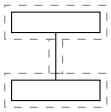


Figure 4: Dummy node

[1] Compact meaning here that if we draw a vertical line from the top of the drawing to the bottom of the drawing at any horizontal coordinate inside the drawing, we will cross at least one node of the tree.

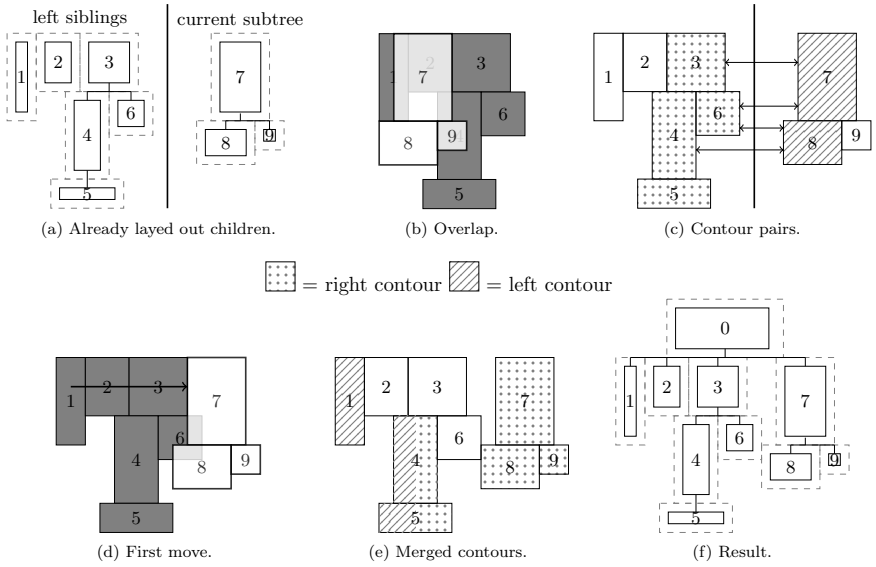


Figure 5: Moving a child subtree.

is the mirror image of the drawing of the original tree.

The rationale for these aesthetic criteria can be found in [5].

3 Overview of the extended Reingold-Tilford algorithm

We now give a high-level description of the Reingold-Tilford algorithm and introduce our extension for the non-layered case. A pseudo-algorithm for this high-level description is given in Algorithm 1. The Reingold-Tilford algorithm first recursively processes all the children of the tree, which produces a layout of each child as if it were the root, and hence the children overlap. Afterwards, each child subtree is moved to the right such that it does not overlap with its left siblings. After moving the children, the horizontal position of the root node is set such that it is centered above its children.

Moving the children works as follows: the algorithm iterates over the children from left to right, and moves each child subtree so that it does not overlap with any of its left siblings. Consider an example of such an iteration, shown in Figure 5. The start of the iteration is shown in Figure 5(a). Since the algorithm does not deal with spacing and connecting lines, we only show these at the start and at the end, i.e., in Figure 5(a) and Figure 5(f). In the left part of the Figure

```
1  Layout(root) begin
2  |  foreach Each child of root do
3  |  |  Layout(child);
4  |  |  Separate((left siblings,child)) ;
5  |  Set position of root;
6  Separate(left siblings, current subtree) begin
   |  /* The contour pair is the pair of these two variables.
   |     */
7  |  Current right contour node ← root of rightmost sibling;
8  |  Current left contour node ← root of current subtree;
9  |  while right contour node ≠ null ∧ left contour node ≠ null do
10 |  |  x_l ← horizontal position of the left side of the current left contour
   |  |  node;
11 |  |  x_r ← horizontal position of the right side of the current right
   |  |  contour node;
12 |  |  if x_l < x_r then
13 |  |  |  Move current subtree by x_r − x_l to the right;
14 |  |  y_l ← vertical position of the bottom of the current left contour
   |  |  node;
15 |  |  y_r ← vertical position of the bottom of the current right contour
   |  |  node;
   |  |  /* Coordinate system increases upwards.                  */
16 |  |  if y_l ≤ y_r then
17 |  |  |  Current left contour node ← next node of the left contour;
18 |  |  if y_l ≥ y_r then
19 |  |  |  Current right contour node ← next node of the right contour;
20 |  Merge contours;
```

Algorithm 1: High-level description of the extended Reingold-Tilford algorithm.

5(a) we see that three left sibling subtrees were already layed out and moved, namely the subtree consisting of the node 1, the subtree consisting of the node 2 and the subtree consisting of the nodes $3, 4, 5, 6$. The subtree that should now be moved, from now on referred to as the *current* subtree, is shown in the right part of Figure 5(a) and consists of the nodes $7, 8, 9$. The layout of the current subtree and the left siblings is already correct, due to recursion. Notice that the left siblings and the current subtree are actually in the same space, and thus overlap as shown in Figure 5(b).

To see how much the current subtree must be moved, we use the *contours* of the current subtree and its left siblings. The left contour is the list of the nodes, from top to bottom, that can be "seen" from the left. The right contour is defined symmetrically. The contours in the example are shown in Figure 5(c). In our example, the left siblings have a left contour, consisting of the nodes $[1, 4, 5]$, and a right contour, consisting of the nodes $[3, 6, 4, 5]$. The current subtree has of a left contour, $[7, 8]$, and a right contour, $[7, 9, 8]$.

To move the current subtree, only the right contour of the left siblings and the left contour of the current subtree are needed. The algorithm then processes all *contour pairs*: vertically overlapping nodes from both contours. The contour pairs in our example are also shown in Figure 5(c).

In the layered case, finding the contour pairs works as follows: The first contour pair consists of the first node of the right contour and the first node of the left contour. The next contour pair is found by advancing both nodes from the current pair to the next element of their contour. We iterate this process until one of the contours has no more elements. In the non-layered case, the bottom coordinates of the contour nodes do not have to be equal. Hence, in each iteration only the highest one will be advanced to the next node of its contour, or both if they have the same bottom coordinates. This is the only modification needed to make the Reingold-Tilford algorithm work for non-layered trees. This modification consists of the two tests in lines 17 and 19 in our high-level description of the algorithm in Algorithm 1. In our non-layered example in Figure 5(c), this process yields contour pairs $\langle 3, 7\rangle, \langle 6, 7\rangle, \langle 6, 8\rangle$ and $\langle 4, 8\rangle$.

For each contour pair we then check if the left side of the left contour node is to the left of the right side of the right contour node. If this is the case, then the current subtree overlaps with its left siblings, and we move the current subtree such that the horizontal position of the left side of the left contour node is the same as the horizontal position of the right side of the node in the right contour. In our example the first contour pair is $\langle 3, 7\rangle$, and the left side of 7 is indeed to the left of the right side of 3, as can be seen in 5(b). We then move the current subtree to the right such that the left side of 7 is the right side of 3, as shown in Figure 5(d). The current subtree is then moved again for contour pairs $\langle 6, 7\rangle$ and $\langle 6, 8\rangle$, after which the current subtree is positioned as shown in Figure 5(e). Notice that this is *not* the same as simply moving the current subtree by the distance between the left of its leftmost node and the right of the rightmost node of its left sibling.

Afterwards the contours of the left siblings and the current subtree are

merged into a new left and right contour, so that these are available later. In our example, the left siblings were taller than the current subtree. The left merged contour is then just the left contour of the left siblings, as shown in Figure 5(e). The merged right contour is the right contour of the current subtree, followed by the remainder of the right contour of the left siblings. More precisely, the merged left contour consists of the nodes $[1, 4, 5]$, and the merged right contour consists of the nodes $[7, 9, 8, 4, 5]$. After merging the contours the iteration ends and the algorithm starts moving the next child subtree. In our example, the subtree rooted at 7 was the last child, so the algorithm positions the root node using the positions of its children. The result can be seen in Figure 5(f).

A naïve implementation of the above algorithm will not run in linear time. In order to get a linear run-time the Reingold-Tilford algorithm uses techniques to do both of the following in $O(1)$:

- Getting the next element of a contour.
- Moving a subtree horizontally.

For the former, the algorithm maintains two fields called the left and right *threads* for each node, which contain a reference to the next node in the left or right contour respectively. Getting the next element of a contour is then simply using this reference. For the latter, the algorithm makes use of *relative coordinates*. The details of these techniques are given in Appendix A. In the rest of this paper, it suffices to know that these operations can be done in $O(1)$. It does not matter how this is achieved.

It should be clear that this algorithm satisfies aesthetic criteria 1-3, as listed on page 4. See [13] for a more in depth discussion of this. Aesthetic criteria 4 also holds, since the algorithm is a simple recursive algorithm that lays out each subtree in the same fashion, without taking into account where in the tree the subtree is located. Aesthetic criteria 5 does *not* hold, and we will see later how this can be fixed.

4 Complexity Proof

The original complexity proof of the Reingold-Tilford algorithm uses an invariant which does not hold in the non-layered case: the number of nodes in the left or right contour is equal to the depth of the tree, i.e. the length of the longest path from root to leaf. We will now give an alternative complexity proof, which does not use this invariant and then generalize this proof to the non-layered case.

The running time of the Reingold-Tilford algorithm depends on the total amount of contour pairs considered to move all subtrees. The total number of contour pairs is the same as the total number of times that the program executes the body of the while loop in Algorithm 1. The centering of a root above its children costs constant time per node, as we only need the positions of the leftmost and rightmost child. The moving of a subtree and the obtaining of the next node of a contour costs constant time per contour pair, by using the

techniques explained in Appendix A. Hence, if the total number of contour pairs processed during the layout of the entire tree is linear in the size of the tree, then the Reingold-Tilford algorithm runs in linear time.

4.1 Layered case

Let us first assume that the input tree is *layered*. The key insight for this complexity proof is the following: if a node in the left contour of a subtree was processed to move that subtree, then it *cannot* be part of the left contour of another subtree. As an example, consider Figure 1. Here the left contour of the right child consisted of $4, 5$ and 6. The left contour nodes considered when moving the right child are 4 and 5, which thus *cannot* reoccur in another left contour. The reason for this is that after moving the current subtree, the left contour nodes that were processed will all have a node in the left siblings to the left of them. In our example, the processed left contour nodes 4 and 5 have the nodes 1 and 3 to the left of the them respectively. In other words, since the left contour is all the nodes that can be "seen" from the left, the processed left contour nodes of the current subtree cannot be part of the merged contour, because they are occluded by nodes in the left siblings. An analogous insight holds for the right contour nodes.

More formally, the input tree consists of n nodes, $v_1 \dots v_n$. Let $f_l(v_i)$ be the set of left contour nodes processed to move the subtree with root v_i. Due to the above insight, we know that if a node v_i is in a set $f_l(v_j)$, it cannot be a part of any other set $f_l(v_z)$, with $z \neq j$. Because of this, we know that the total number of left contour nodes processed to layout the entire tree is less than or equal to n, i.e.:

$$\sum_{i=1}^{n} |f_l(v_i)| \leq n$$

Where $|x|$ denotes the number of elements in the set x. The equivalent holds for $f_r(v_i)$, the set of right contour nodes that where processed to move the subtree with root v_i.

Let $f(v_i)$ be the set of contour pairs processed to move the subtree with root v_i. In the layered case, nodes are aligned vertically, and hence we know that $|f_l(v_i)| = |f_r(v_i)| = |f(v_i)|$, where $|x|$ denotes the size of the set x. Hence, the amount of contour pairs processed during the entire algorithm is less than or equal to n, which means that the Reingold-Tilford algorithm runs in linear time.

4.2 Non-layered case

In the non-layered case, nodes are not necessarily vertically aligned. Hence, the amount of contour pairs processed to move a child is no longer the same as the number of left contour elements processed. In the worst case the nodes from the right and left contours are never aligned, i.e., their bottom coordinates are never the same. After processing a contour pair, we will advance either along the left

or the right contour or along both. This gives us an upper bound on the number of contour pairs:

$$|f(v_i)| \leq |f_l(v_i)| + |f_r(v_i)|$$

Another difference to the layered case is that a node in a left contour now *can* be processed to move the subtree *as well as* being included in another left contour. Again, the same holds for right contours. As an example of a right contour node that is also included in another right contour, consider the tree in Figure 5(c). In this example, right contour node 4 is processed when moving the subtree rooted at 7 and it is also in the right contour of the entire tree, rooted at 0, as shown in Figure 5(d).

However, this can *only* happen if the node is the *last* right contour node that was processed to move a subtree. The reason for this is that the top part of the last node that was considered is occluded by nodes to the right while other nodes that were considered must be *totally* occluded by nodes to the right. Again, as an example, consider Figure 5(c). The last considered right contour node of the left siblings, node 4, is partially occluded by the nodes 7 and 8 to the right in the merged contour in Figure 5(d). However the other right contour nodes that were considered , namely 3 and 6, are totally occluded by nodes to the right, namely 7 and 8. The same reasoning holds for the last left contour node that was processed to move a subtree.

Let $f_l^p(v_i)$ be the set of left contour nodes that where processed to move the subtree with root v_i, *except* the last left contour node that was processed. More formally $f_l^p(v_i) = f_l(v_i) - \{l_l(v_i)\}$, where $l_l(v_i)$ is the last node of the left contour that is processed to move the subtree with root v_i. Since only last elements of a contour can reappear, we know that:

$$\sum_{i=1}^{n} |f_l^p(v_i)| \leq n$$

When moving each subtree, there will be only one last left contour element considered, and since there are n subtrees, we know that:

$$\sum_{i=1}^{n} |f_l(v_i)| = \sum_{i=1}^{n} [|f_l^p(v_i)| + |\{l_l(v_i)\}|] = \sum_{i=1}^{n} |f_l^p(v_i)| + n \leq 2n$$

The same argument can be made for the right contour, and hence we know that:

$$\sum_{i=1}^{n} |f(v_i)| \leq \sum_{i=1}^{n} |f_l(v_i)| + \sum_{i=1}^{n} |f_r(v_i)| \leq 4n = O(n)$$

Which proves that the extension of the Reingold-Tilford algorithm for non-layered trees also runs in linear time.

5 Improving layouts

The above Reingold-Tilford algorithm for non-layered trees satisfies aesthetics 1-4, but not aesthetic 5 [11]: the drawing of the reflection of a tree is not the

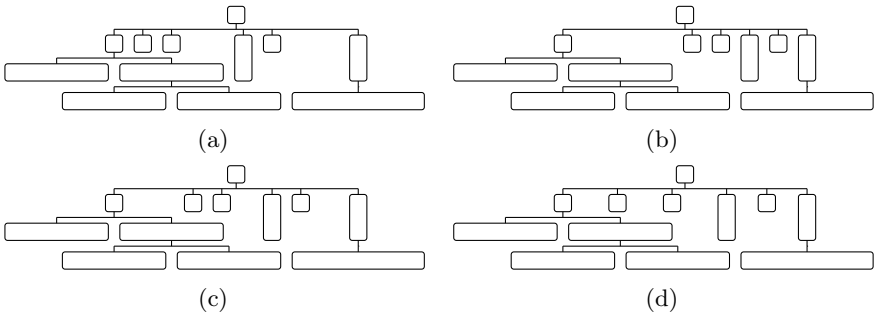


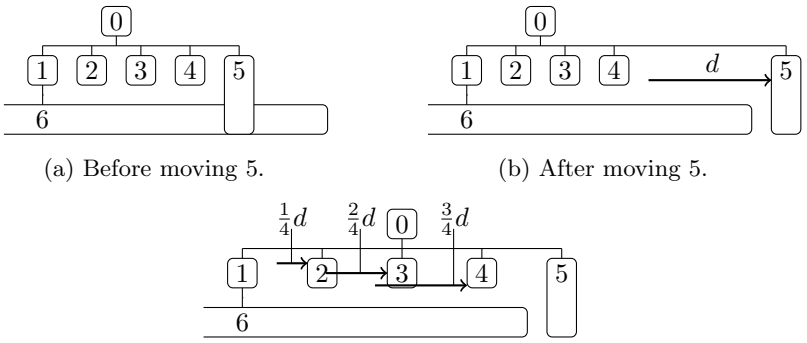
Figure 6: Layouts not satisfying aesthetic 5 (a,b) and satisfying aesthetic 5(c,d). Inspired by a figure in [12].

mirror image of the drawing of the original tree. An example of this is shown in Figure 6(a). When subtrees are enclosed by larger siblings they will be piled to the left. If we draw the reflected tree and then mirror the layout, the subtrees are piled to the right as shown in Figure 6(b). A simple trick to satisfy aesthetic 5 is to take the average of the horizontal position of the each node in the original and mirrored, reflected drawing, which results in a layout as shown in Figure 6(c). However, this tends to cluster smaller subtrees, which is less aesthetically pleasing. Walker [11] noticed this problem and proposed extensions to the Reingold-Tilford algorithm to produce aesthetically more pleasing layouts, such as the one shown in Figure 6(d). Buchheim, Jünger and Leipert [12] then showed that the Walker algorithm runs in $O(n^2)$ and provided techniques to restore the linear running time.

To see how such layouts are achieved, consider the example shown in Figure 7(a). Here we see that we have already layed out and moved children 1-4 and we are currently moving child 5 to the right. We already processed the first node of the right contour of the left siblings, namely 4, and hence the left side of node 5 is no longer to the left of the right side of node 4. We now move on to the next node of the right contour, 6. As shown in Figure 7(b), we need to move node 5 by a distance d to the right. In the Reingold-Tilford algorithm as described before this would have been the only thing we would have done. To satisfy aesthetic 5, we notice that the current node in the right contour, 6, which caused the move by d, is in the sibling subtree with root 1.

If we move 5 by d, then there is d space between 5 and its left sibling, 4, as shown in Figure 7(b). We can then distribute this extra space over the gaps between the intermediate siblings, the siblings between the sibling that caused the move and the current subtree, namely 1 through 4. Since there are 4 gaps, we move the first intermediate sibling node by a distance $\frac{1}{4}d$, the second by $\frac{2}{4}d$, and the third by $\frac{3}{4}d$, as shown in Figure 7(c).

In general, suppose a node in the current subtree is a distance d to the left of a node v in the right contour of its left siblings. After moving the current subtree by d, we then see which left sibling is the ancestor of v. Let i be the



(a) Before moving 5.

(b) After moving 5.

(c) Distributing the space over intermediate siblings.

Figure 7: Shifting intermediate children

index of this left sibling and j be the index of the current subtree. We then move each intermediate sibling, with an index z the range $[i+1 \ldots j-1]$, by a distance $\frac{z-i}{j-i}d$.

To do the above modification to the Reingold-Tilford algorithm while retaining the running time of $O(n)$, Buchheim et al. introduce techniques to do both of the following in $O(1)$:

- Move the intermediate siblings as described above.
- Given a node in the right contour, get the index of the sibling subtree which contains that node.

For the first point, Buchheim et al. propose a technique which is also applicable in the non-layered case. Its details are not important in this paper, but it is explained in Appendix B for sake of completeness.

For the second point, the Buchheim et al. propose a technique which requires updating all the nodes in the right contour of a subtree after moving that subtree, but only if the subtree is less tall than its left siblings. If a subtree is less tall than its left siblings, then all its left contour nodes are considered to move that subtree. In the layered case, the left and right contours must have exactly as many elements. Therefore, the number of right contour nodes of a subtree that is less tall than its left siblings is the same as the number of contour pairs considered to move that subtree. Hence, updating the right contour of subtrees that are less tall than its left siblings does not modify the $O(n)$ run-time in the layered case. In the non-layered case this technique causes a run-time of $O(n^2)$, which we will show and remedy in the next section.

6 Improving layouts in linear time

In the non-layered case, the left and right contours do not have to have the same number of elements. Because of this, updating the right contour nodes of

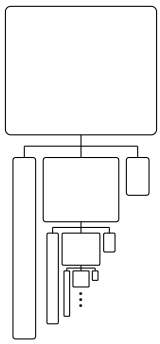


Figure 8: A tree construct where the sibling index lookup technique of Buchheim et al. gives $O(n^2)$ run-time.

subtrees that are less tall then their left siblings leads to an $O(n^2)$ run-time. As an example of this, consider the tree construct shown in Figure 8. Formally, we construct a tree given a parameter k: the root node has width and height 2^k, and has three children:

- A child consisting of a single node of width $\frac{1}{4}2^k$ and height $\frac{5}{4}2^k$.
- A child subtree constructed in the same way, with $k = k - 1$.
- A child consisting of a single node of width $\frac{1}{4}2^k$ and height $\frac{1}{4}2^k$.

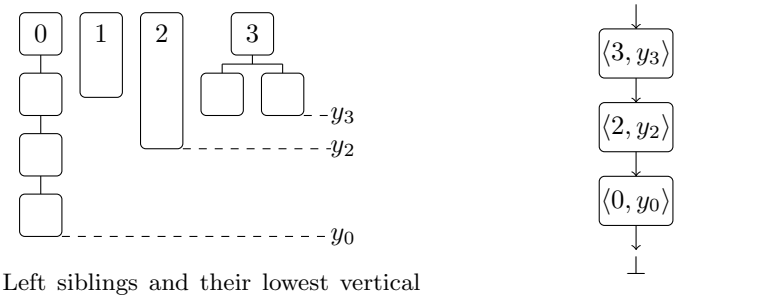
When $k = 1$, the tree is constructed in the same way, but the middle child is instead a single node with width and height 1.

For every k, the middle child subtree is less tall then its left sibling. Hence, when using Buchheim et al.'s technique we must always update the nodes in the right contour of the middle child after moving it to the right. For each k, a tree constructed in this way has $3k + 1$ nodes. The right contour of the middle child consists of all nodes in its subtree, i.e. $3(k - 1) + 1$ nodes. Updating the right contour *for all* middle children in in such a tree then takes:

$$\sum_{i=1}^{k-1}[3i + 1] = 3\sum_{i=1}^{k-1} i + k - 1 = O(k^2)$$

Due to the well-know equality $\sum_{i=1}^{k} i = k(k + 1)/2$. Since there is a linear relation between n and k, this means that the algorithm runs in $O(n^2)$.

Hence, we need a different technique to find the index of the sibling subtree that contains a given node in the right contour. As noted by Buchheim et al., it is also possible to adopt the lowest common ancestor algorithm of Schrieber and Vishkin [14] to find the index of the sibling subtree in $O(1)$, after an $O(n)$ preprocessing step. This is indeed possible, but not trivial, and Buchheim et al. do not describe the details.



(a) Left siblings and their lowest vertical coordinates.

(b) Left sibling lookup list.

Figure 9: A tree layout with the corresponding linked list.

We propose a different, much simpler technique: during the moving of the children we maintain a linked list of the siblings that currently have a node in the right contour. Each node in this linked list is a pair of the index of the corresponding sibling and its lowest vertical bottom coordinate. This list is always sorted in descending order of the siblings indices. An example of a tree with the corresponding list is shown in Figure 9. When moving a child subtree we advance this list if the current right contour node has a lower vertical coordinate than the pair at the head of the list, adding only $O(1)$ operations per contour pair. The index of the sibling subtree that contains the current right contour node is then always given in the pair at the head of the list.

After moving a current subtree to the right, we need the pair for the current subtree, i.e. the subtree that was just moved. This pair consists of the sibling index of the current subtree, which we already know, and the lowest vertical bottom coordinate of the current subtree. The latter is easily found when using the techniques in the Reingold-Tilford algorithm as described in Appendix A. More precisely, the Reingold-Tilford algorithm keeps track of the *extreme nodes* of each subtree, i.e. the lowest nodes that can be "seen" from the left or right. The lowest vertical bottom coordinate of a subtree is then the bottom coordinate of either of its extreme nodes, and can hence be found in $O(1)$.

To update the list, we then remove elements at the head of the list that have a higher lowest vertical coordinate than the new pair. This removes siblings from the list that had nodes in the right contour, but these nodes are now occluded by the current subtree. Afterwards, we prepend the new pair to the list. In this way, the list always corresponds to the siblings that currently have a node in the right contour.

The total number of operations needed for updating the list when moving all the children of a node is at most $2m(v_i)$, where $m(v_i)$ is the number of children of a node v_i. This can be seen as follows: we update the list $m(v_i)$ times during the moving of the children, namely after moving each child. Each time we remove some number of elements from the head of the list and an element is prepended to the list. Since we add $m(v_i)$ elements to the list, and an element

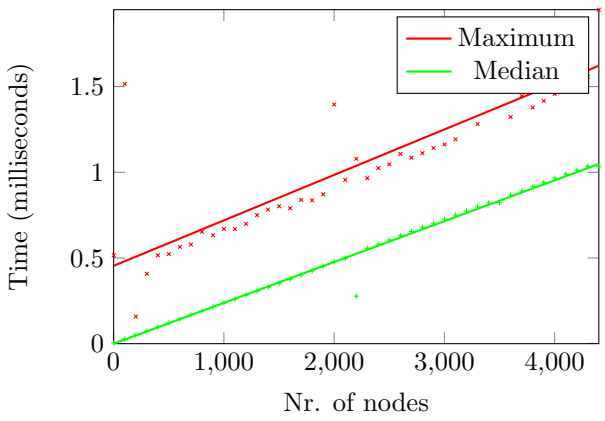


Figure 10: Measurements on random trees. The minimum is not shown, as it coincides with the x-axis.

can only be removed once, the total number of elements dropped is at most $m(v_i)$. Hence, together the prepending and dropping of elements cost at most $2m(v_i)$ operations per node.

For the entire tree this will add at most $\sum_{i=1}^{n} 2m(v_i)$ operations. Every node is a child of exactly one other node, except for the root who is a child of no one. Hence, the sum of the number of children of all nodes, $\sum_{i=1}^{n} m(v_i)$, is equal to $n-1$. Because of this we know that the updating of the lists cost

$$\sum_{i=1}^{n} 2m(v_i) = 2\sum_{i=1}^{n} m(v_i) = 2(n-1) = O(n)$$

extra operations for the entire tree. Since the updating of the place in the list during the moving of a subtree cost $O(1)$ per contour pair, this will also add $O(n)$ extra operation to the algorithm. Hence, the running time of the algorithm with this extension is linear in the size of the tree.

7 Empirical results

In practice the algorithm presented in this paper is very fast, and should hence be applicable in any computing environment. In Figure 10 we show the results of measuring the time the algorithm took to lay out randomly generated trees. We used the following procedure to generate a tree with n nodes: Start with a tree with a single node. Then, for each other node, add it as a leaf in the current tree in a random position. This random position is determined by descending the current tree as follows: Starting with the root node as the current node, we generate a random integer between zero and the number of children of the current node. If the random number is zero, then new node

is added as a new child of the current node. Otherwise, we descend into the child with as index the random number and repeat the procedure. Each node has a random width and height uniformly distributed between 1.0 and 10.0. As there are more possible trees as the number of nodes goes up, we generated $200 \times n$ tests, where n is the number of nodes, for each multiple of 100 in the range [0,4400]. These results were obtained using OpenJDK java runtime version 2.3.9.8 on an Intel i7 2.8 GHz CPU running Fedora Linux 3.9.4-200.fc18.x86_64. This experiment can be repeated by downloading the source code from `http://github.com/cwi-swat/non-layered-tidy-trees`.

8 Related work

8.1 History

The history of the Reingold-Tilford algorithm is quite long: In 1979, Wetherell and Shannon [15] presented the first $O(n)$ algorithm that produces drawing satisfying aesthetics 1-3, that was inspired by a tree drawing algorithm presented by Knuth in 1971 [16]. Two years later, Reingold and Tilford [5] gave an algorithm, inspired by the Wetherell and Shannon algorithm, that also satisfied aesthetic 4. Then in 1990, Walker [11] presented an improvement such that aesthetic 5 is also satisfied for trees of unbounded degree. In 2002 Buchheim, Jünger and Leipert showed that Walker's algorithm ran in $O(n^2)$, in contrast to what the author claimed. They presented improvements to the Reingold and Tilford algorithm inspired by Walker's work that did run in $O(n)$. All these versions of the Reingold-Tilford algorithm, and their proofs assume layered trees.

8.2 Algorithms for non-layered trees

There have been several previous efforts to produce non-layered drawings of trees. All of these either make simplifying assumptions and/or have not been proven to linear time.

Miyadera et al. present an $O(n^2)$ algorithm [8] for non-layered trees that horizontally positions a parent at a fixed offset from its first child, instead of centered above the children. This greatly simplifies things, as only the first child needs to be layed out before positioning the root node, allowing a simple depth-first solution. A proof that such a $O(n)$ depth-first solution exists was given by Hasan and Radwan [7]. This type of layout can also be easily handled by the extended Reingold-Tilford algorithm, by simply modifying the computation of the position of the root node. This will break aesthetic 5, as the Hasan and Miyadera algorithms also do.

Bloesch gives two algorithms [6] for non-layered trees that are intended to satisfy aesthetics 1-5. Both algorithms follow the same idea: discretize the drawing vertically. The first is then a variant of an algorithm for layered trees by Vaucher [17] and the second is a variant of the original Reingold-Tilford algorithm. This variant does not use threads (described in Appendix A) like the

original Reingold-Tilford algorithm, as the author states that this is impossible in a non-layered setting, which is obviously false. It is unclear to us how the drawing generated by these algorithms satisfy aesthetics 4 and 5. Bloesch reports that these algorithms run in $O(nh)$, where h is the number of elements in the discretization of the height. It is unclear to us whether this is true, as no proof is given.

Stein and Benteler [9] propose a similar technique: A non-layered tree is converted into a layered tree by discretizing horizontally and vertically. Afterwards, an algorithm for layered trees can be applied and the results can be translated back. This approach runs in $O(f(n)wh)$, where w and h are the number of elements in the horizontal and vertical discretization respectively and $f(n)$ is the running time of the algorithm for layered trees.

Xiaohong and Jingwei [10] present an algorithm for non-layered trees satisfying aesthetic criteria 1-4. The algorithm is presented as a complete novelty, but it is the Reingold-Tilford algorithm with the small extension that we introduced in Section 3, which amounts to three extra lines in the algorithm as shown in Appendix C. In contrast to our paper, no proof is given of the time complexity nor do the layouts by algorithm satisfy aesthetic criteria 5.

Marriot, Sbarski, Van Gelder, Prager and Bulka [18] presented, among other things, a technique to produce a more vertically compact drawing of a tree where the nodes have different heights. This technique works by first pre-processing the tree and then using the Reingold-Tilford algorithm extended with the techniques of Walker and Buchheim. In the pre-processing step, the tree is "re-layered": if the distance between a parent and a child is too large, the layer is split in two. In this way, a more vertically compact layout can be obtained in $O(n \log n)$. With the algorithm presented in this paper we can achieve a drawing with *minimal* height in $O(n)$. A non-layered drawing has minimal height, since the top-coordinate of a node in the bottom coordinate of its parent (abstracting away from spacing).

8.3 Related work on node-link drawings on trees

Apart from algorithms for drawing node-link diagrams of trees, various results have been published on other aspects of node-link diagrams of trees: Gibbons [13] derives the Reingold-Tilford algorithm for binary trees from the aesthetic criteria. His derivation of the algorithm shows that the Reingold-Tilford is the only reasonable algorithm that satisfies the aesthetic criteria.

Kennedy [19] shows how to implement a variant of the Reingold-Tilford algorithm in a purely functional setting, with time complexity $O(n^2)$. We note that it should be possible to implement the Reingold-Tilford algorithm with its extensions in a purely functional setting while retaining the $O(n)$ run-time. This could work by separating the contours from the tree itself, i.e. maintaining a separate list representing the contour instead of reusing the tree structure for this. This would get rid of the need for mutability. If we then also choose a purely functional data structure for such a contour list with $O(1)$ first and last element access and $O(1)$ list concatenation, such as the data structure given by

Kaplan and Tarjan [20], we can implement a purely functional version running in $O(n)$.

Suppowit and Reingold [21] investigated the complexity of drawing trees nicely. They found that a drawing with global minimum width may have subtrees that are much wider than necessary. For this reason the Reingold-Tilford algorithm does not promise minimal width. They also found that the tidy tree problem can be reduced to a linear programming problem and that it is NP-hard if the horizontal coordinates are restricted to integers.

Moen [22] shows a variant of the Reingold-Tilford algorithm that works in approximately the same way, the main difference being that he uses a separate data structure for the contour instead of reusing the tree itself. He then shows how to keep the layout of the tree up-to-date when there are insertions and deletions in the tree.

Marriot and Sbarski [23] relax the requirement that a parent must be placed exactly between the children, making it a preference that may be violated if it yields a tree with a smaller width. Their approach is to first find a initial layout using the (extended) Reingold-Tilford algorithm and then solve a kind of quadratic programming problem to see where the preference should be violated to produce a more narrow drawing. This technique requires that the drawing is divided into layers. However, in another publication [18], Marriot and Sbarski showed how to this technique can be applied in a setting where a node can span *multiple* layers. Hence, if we introduce a layer for each unique vertical position and then assign nodes to layers, then this technique can also be used together with the algorithm presented in this paper.

8.4 Other ways of drawing trees

We will now give a short and by no means complete overview of other tree visualization methods, for a more complete overview see [24, 25].

There are many variations on the basic node-link diagram. One way to adapt node-link diagrams is to change the coordinate system in which they are drawn. Radial trees draw node-link diagrams in a polar coordinate system, where the root is displayed at the origin. Balloon trees are similar, but the children of *each* node are in a circle around the node instead. The hyperbolic browser[26] also changes the coordinate system in which a node-link diagram is draw, namely to a hyperbolic plane.

Another way to adapt node-link diagrams is to move from 2D to 3D. Cone trees [27] are a 3D generalization of balloon trees: viewed from the top the diagram is a balloon tree, while viewed from the side we see a cone from each root node to its children. Another 3D generalization of node-link diagrams is visualize a hierarchy as as a botanical tree [3].

Instead of node-link diagram, the parent-child relationship can also be visualized by *containment*: the children are drawn inside the parent. Treemaps [2] draw nodes as rectangles, and each rectangle is subdivided into the rectangles of the children. The area of each rectangle signifies the size of the node, for

example the size of a file or the total size of a directory when visualizing a file system.

Hi-trees [18] combine containment and node-link diagrams: the parent-child relationship is visualized by either a link or containment, depending on the type of the relationship between parent and child. For example, arguments in a discussion can have sub-arguments (containment) and supporting and opposing arguments (links).

Acknowledgements

We thank Leen Torenvliet, Menno van der Ploeg, Anastasia Izmaylova, Paul Klint and the anonymous reviewers for their constructive comments on this paper.

References

[1] Tollis IG, Di Battista G, Eades P, Tamassia R. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[2] Johnson B, Shneiderman B. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. *Proceedings of the '91 IEEE Conference on Visualization*, 1991; 284 –291.

[3] Kleiberg E, van de Wetering H, Van Wijk JJ. Botanical visualization of huge hierarchies. *Proceedings of the IEEE Symposium on Information Visualization 2001*, 2001; 87–94.

[4] Lanza M, Ducasse S. Polymetric views - a lightweight visual approach to reverse engineering. *Transactions on Software Engineering* 2003; **29**:782–795.

[5] Reingold EM, Tilford JS. Tidier drawings of trees. *IEEE Transactions on Software Engineering* 1981; **7**(2):223–228.

[6] Bloesch A. Aesthetic layout of generalized trees. *Software: Practice and Experience* August 1993; **23**:817–827.

[7] Hasan M, Rahman MS, Nishizeki T. A linear algorithm for compact box-drawings of trees. *Networks* 2003; **42**(3):160–164.

[8] Miyadera Y, Anzai K, Unno H, Yaku T. Depth-first layout algorithm for trees. *Information Processing Letters* May 1998; **66**:187–194.

[9] Stein B, Benteler F. On the generalized box-drawing of trees: Survey and new technology. *Proceeding of I-KNOW '07*, 2007.

[10] Xiaohong L, Jingwei H. An improved generalized tree layout algorithm. *Proceedings of the 2nd international Asia conference on Informatics in control, automation and robotics - Volume 2*, CAR'10, IEEE Press, 2010; 163–166.

[11] Walker JQ II. A node-positioning algorithm for general trees. *Software: Practice and Experience* 1990; **20**:685–705.

[12] Buchheim C, Jünger M, Leipert S. Drawing rooted trees in linear time. *Software: Practice and Experience* 2006; **36**(6):651–665.

[13] Gibbons J. Deriving tidy drawings of trees. *Journal of Functional Programming* 1996; **6**(3):535–562.

[14] Schieber B, Vishkin U. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing* 1988; **17**(6):1253–1262.

[15] Wetherell C, Shannon A. Tidy drawings of trees. *IEEE Transactions on Software Engineering* 1979; **5**(5):514–520.

[16] Knuth DE. Optimum binary search trees. *Acta Informatica* 1971; **1**:14–25.

[17] Vaucher JG. Pretty-printing of trees. *Software: Practice and Experience* 1980; **10**(7):553–561.

[18] Marriott K, Sbarski P, van Gelder T, Prager D, Bulka A. Hi-trees and their layout. *IEEE Transactions on Visualization and Computer Graphics* 2011; **17**(3):290–304.

[19] Kennedy A. Drawing trees. *Journal of Functional Programming* 1996; **6**(3):527–534.

[20] Kaplan H, Tarjan RE. Purely functional, real-time deques with catenation. *Journal of the ACM* 1999; **46**(5):577–603.

[21] Supowit K, Reingold E. The complexity of drawing trees nicely. *Acta Informatica* 1983; **18**:377–392.

[22] Moen S. Drawing dynamic trees. *IEEE Software* 1990; **7**:21–28.

[23] Marriott K, Sbarski P. Compact layout of layered trees. *Proceedings of the 13th Australasian conference on Computer science - Volume 62*, ACSC '07, Australian Computer Society, Inc., 2007; 7–14.

[24] Katifori A, Halatsis C, Lepouras G, Vassilakis C, Giannopoulou E. Ontology visualization methodsa survey. *ACM Computing Surveys (CSUR)* 2007; **39**(4):10.

[25] Nguyen QV, Huang ML. A space-optimized tree visualization. *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, IEEE, 2002; 85–92.

[26] Lamping J, Rao R, Pirolli P. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 1995; 401–408.

[27] Robertson GG, Mackinlay JD, Card SK. Cone trees: animated 3d visualizations of hierarchical information. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 1991; 189–194.

A Techniques in the Reingold-Tilford algorithm

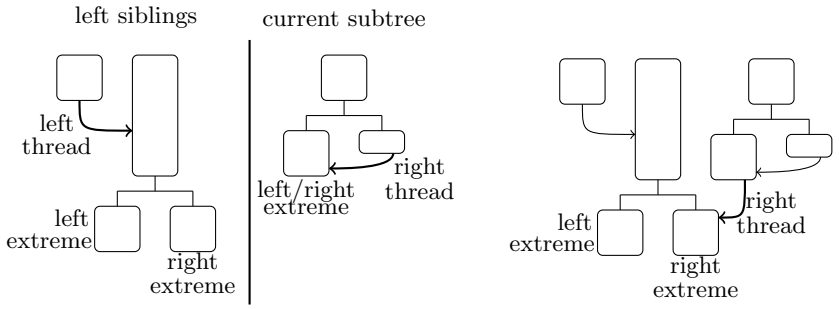
In order to get a linear run-time the Reingold-Tilford algorithm [5] uses techniques to do both of the following in $O(1)$:

- Getting the next element of a contour.
- Moving a subtree horizontally.

The operation for the first item, getting the next node of a contour, depends on whether the current node is a leaf or not. If the node is not a leaf, then the next element of the left contour is its leftmost child, and the next element of the right contour is its rightmost child. For leafs, the next element of the left and right contours are stored in two fields of each leaf, called the left and right *threads*. To keep the threads up-to-date, the algorithm has two additional fields per node: the left and right *extreme nodes*. The left and right extreme nodes of a set of siblings is the lowest node in the subtree the can be "seen" from the left and right respectively. For an example of threads and extreme nodes, see Figure 11(a). Before moving the current subtree, its left and right extreme nodes point to the extreme nodes in the current subtree. After moving the current subtree, its right extreme node points to the extreme node of the current subtree *and* its left siblings. The left extreme node of the first sibling always points to the left extreme node of the siblings that are already moved. Thus, the left extreme of a set of siblings that is already moved is a field of the leftmost sibling, and the right extreme is a field of the rightmost sibling.

After moving a current subtree, the threads and extreme nodes may be updated. If the current subtree was less tall than its left siblings, as is the case in Figure 11, the right thread of the extreme right node of the current subtree is set. Afterwards, the extreme right node of the root of the current subtree is set to the extreme right node of its left siblings. The resulting situation in our example is shown in Figure 11(b). If the current subtree was taller than its left siblings, the operation is symmetrical. If the current subtree is as tall as its left siblings no threads are set and no extreme nodes are updated.

To achieve moving a subtree horizontally in $O(1)$, the Reingold-Tilford algorithm uses a field named *mod*, for modifier, to store for each node how much its *entire* subtree should be moved horizontally. Moving a subtree is then done by simply updating this modifier. Another field, *prelim* is used to remember the *preliminary* horizontal coordinate of the node. This is set when we position the root after moving its children and represents the distance that the left side of the root is positioned relative to the left side of its first child. After laying out the entire tree, a single extra pass over the tree to computes the actual horizontal coordinate of each node. This use of relative coordinates requires some changes



(a) Before (b) After

Figure 11: Before and after settings threads and updating extreme nodes.

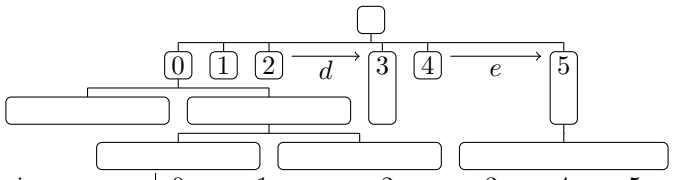
to the rest of the algorithm: during the moving of a subtree, we must maintain and take into account the sum of modifiers along the left and right contours to compute the horizontal positions of the contour nodes.

When setting a thread, we must ensure that if we follow the thread to a node, the sum of the modifiers along the contour is the same as the sum of the modifiers along the route without threads from the root to that node. We will only set threads of extreme nodes, which must be leafs. Hence if we adjust the modifier of an extreme node and adjust its preliminary horizontal position by an opposite amount, we will not actually change the position of any node. In this way we can adjust the modifier of the extreme node such that the sum of modifiers after following the thread is equal to the sum of modifiers when following the route without threads.

B Moving intermediate siblings in $O(1)$

To move an arbitrary number of intermediate siblings in $O(1)$, Buchheim et al. [12] propose to add two fields to each node, namely *shift* and *change*. Suppose, like in Section 5, that i is the index of the sibling which is an ancestor of the current node in the right contour, j is the index of the current subtree, and we move the current child by a distance d. We then add $\frac{1}{j-i}d$ to the *shift* of the $i+1$th child, subtract $\frac{1}{j-i}d$ from *shift* of jth child and subtract $\frac{j-i-1}{j-i}d$ from the *change* of the jth child, as shown in the method `distributeExtra` in Appendix C. Together, these operations cost only constant time.

After laying out the entire tree, we do a constant number of extra operations per node to compute the actual offset of each child, using these *shift* and *change* fields. This can be done during the post-processing phase that produces the absolute horizontal coordinates. Before descending into the children, the *shift* and *change* fields are used to calculate the change to *mod*, Δmod, to each child, as shown in method `addChildSpacing` in Appendix C. An example is shown in Figure 12. The example shows the tree from Figure 6, and adding Δmod to the



i	0	1	2	3	4	5
$shift(T_i)$	0	$\frac{1}{3}d + \frac{1}{5}e$	0	$-\frac{1}{3}d$	0	$-\frac{1}{5}e$
$change(T_i)$	0	0	0	$-\frac{2}{3}d$	0	$-\frac{4}{5}e$
d	0	$\frac{1}{3}d + \frac{1}{5}e$	$\frac{1}{3}d + \frac{1}{5}e$	$\frac{1}{5}e$	$\frac{1}{5}e$	0
Δmod	0	$\frac{1}{3}d + \frac{1}{5}e$	$\frac{2}{3}d + \frac{2}{5}e$	$\frac{3}{5}e$	$\frac{4}{5}e$	0

Figure 12: Example of the post-processing of the spacing of intermediate children.

mod of each child will transform 6(a) to 6(d).

C The complete revised algorithm

Below is the full Java code for the Reingold-Tilford algorithm with its extension. This implementation has undergone rigorous testing, including checking for overlapping nodes on random input trees. The extensions are indicated by a symbol in the right margin.

(nothing) = The original Reingold-Tilford algorithm (Section 3 and Appendix A).
⋆ = Our extension for non-layered trees (Section 3).
Extensions for satifying aesthetic 5 (Section 5)
∘ = Buchheim et al.'s extension to move intermediate siblings (Appendix B).
• = Our extension to look up the sibling index of a right contour node (Section 6).

```
1  class Tree {
2     double w, h;           // Width and height.
3     double x, y, prelim, mod, shift, change;
4     Tree tl, tr;           // Left and right thread.
5     Tree el, er;           // Extreme left and right nodes.
6     double msel, mser;     // Sum of modifiers at the extreme nodes.
7     Tree[] c; int cs;      // Array of children and number of children.
8
9     Tree(double w, double h, double y,Tree... c) {
10       this.w = w; this.h = h; this.y = y; this.c = c;
11       this.cs = c.length;
12    }
13 }
14 void layout(Tree t){ firstWalk(t); secondWalk(t,0); }
15
16 void firstWalk(Tree t){
17   if(t.cs == 0){ setExtremes(t); return; }
18   firstWalk(t.c[0]);
19   // Create siblings in contour minimal vertical coordinate and index list.
20   IYL ih =  updateIYL(bottom(t.c[0].el),0,null);                    •
21   for(int i = 1; i < t.cs; i++){
```

```
22        firstWalk(t.c[i]);
23        //Store lowest vertical coordinate while extreme nodes still point in current subtree.
24        double minY = bottom(t.c[i].er);                                    •
25        separate(t,i,ih);
26        ih = updateIYL(minY,i,ih);                                          •
27     }
28     positionRoot(t);
29     setExtremes(t);
30  }
31
32  void setExtremes(Tree t) {
33     if(t.cs == 0){
34        t.el = t; t.er = t;
35        t.msel = t.mser =0;
36     } else {
37        t.el = t.c[0].el; t.msel = t.c[0].msel;
38        t.er = t.c[t.cs-1].er; t.mser = t.c[t.cs-1].mser;
39     }
40  }
41
42  void separate(Tree t,int i,  IYL ih ){
43     // Right contour node of left siblings and its sum of modfiers.
44     Tree sr = t.c[i-1]; double mssr = sr.mod;
45     // Left contour node of current subtree and its sum of modfiers.
46     Tree cl = t.c[i]  ; double mscl = cl.mod;
47     while(sr != null && cl != null){
48        if(bottom(sr) > ih.lowY) ih = ih.nxt;                               •
49        // How far to the left of the right side of sr is the left side of cl?
50        double dist = (mssr + sr.prelim + sr.w) - (mscl + cl.prelim);
51        if(dist > 0){
52           mscl+=dist;
53           moveSubtree(t,i,ih.index,dist);
54        }
55        double sy = bottom(sr), cy = bottom(cl);
56        // Advance highest node(s) and sum(s) of modifiers (Coordinate system increases
  downwards)
57        if(sy <= cy){                                                       ⋆
58           sr = nextRightContour(sr);
59           if(sr!=null) mssr+=sr.mod;
60        }                                                                   ⋆
61        if(sy >= cy){                                                       ⋆
62           cl = nextLeftContour(cl);
63           if(cl!=null) mscl+=cl.mod;
64        }                                                                   ⋆
65     }
66     // Set threads and update extreme nodes.
67     // In the first case, the current subtree must be taller than the left siblings.
68     if(sr == null && cl != null) setLeftThread(t,i,cl, mscl);
69     // In this case, the left siblings must be taller than the current subtree.
70     else if(sr != null && cl == null) setRightThread(t,i,sr,mssr);
71  }
72
73  void moveSubtree(Tree t, int i, int si, double dist) {
74     // Move subtree by changing mod.
75     t.c[i].mod+=dist; t.c[i].msel+=dist; t.c[i].mser+=dist;
76     distributeExtra(t, i, si, dist);                                       ○
77  }
```

```
78
79  Tree nextLeftContour(Tree t) {return t.cs==0 ? t.tl : t.c[0];}
80  Tree nextRightContour(Tree t){return t.cs==0 ? t.tr : t.c[t.cs-1];}
81  double bottom(Tree t) { return t.y + t.h;  }
82
83  void setLeftThread(Tree t, int i, Tree cl, double modsumcl) {
84     Tree li = t.c[0].el;
85     li.tl = cl;
86     // Change mod so that the sum of modifier after following thread is correct.
87     double diff = (modsumcl - cl.mod) - t.c[0].msel ;
88     li.mod += diff;
89     // Change preliminary x coordinate so that the node does not move.
90     li.prelim-=diff;
91     // Update extreme node and its sum of modifiers.
92     t.c[0].el = t.c[i].el; t.c[0].msel = t.c[i].msel;
93  }
94
95  // Symmetrical to setLeftThread.
96  void setRightThread(Tree t, int i, Tree sr, double modsumsr) {
97     Tree ri = t.c[i].er;
98     ri.tr = sr;
99     double diff = (modsumsr - sr.mod) - t.c[i].mser ;
100    ri.mod += diff;
101    ri.prelim-=diff;
102    t.c[i].er = t.c[i-1].er; t.c[i].mser = t.c[i-1].mser;
103 }
104
105 void positionRoot(Tree t) {
106    // Position root between children, taking into account their mod.
107    t.prelim = (t.c[0].prelim + t.c[0].mod + t.c[t.cs-1].mod +
108                t.c[t.cs-1].prelim +  t.c[t.cs-1].w)/2 - t.w/2;
109 }
110
111 void secondWalk(Tree t, double modsum) {
112    modsum+=t.mod;
113    // Set absolute (non-relative) horizontal coordinate.
114    t.x = t.prelim + modsum;
115    addChildSpacing(t);                                                    ○
116    for(int i = 0 ; i < t.cs ; i++) secondWalk(t.c[i],modsum);
117 }
118
119 void distributeExtra(Tree t, int i, int si, double dist) {             ○
120    // Are there intermediate children?
121    if(si != i-1){                                                      ○
122       double nr = i - si;                                              ○
123       t.c[si +1].shift+=dist/nr;                                       ○
124       t.c[i].shift-=dist/nr;                                           ○
125       t.c[i].change-=dist - dist/nr;                                   ○
126    }                                                                   ○
127 }                                                                      ○
128
129 // Process change and shift to add intermediate spacing to mod.
130 void addChildSpacing(Tree t){
131    double d = 0, modsumdelta = 0;                                      ○
132    for(int i = 0 ; i < t.cs ; i++){                                    ○
133       d+=t.c[i].shift;                                                 ○
134       modsumdelta+=d + t.c[i].change;                                  ○
```

```
135         t.c[i].mod+=modsumdelta;                                    ○
136      }                                                              ○
137   }                                                                 ○
138
139    // A linked list of the indexes of left siblings and their lowest vertical coordinate.
140   class IYL{                                                        •
141      double lowY; int index; IYL nxt;                               •
142      public IYL(double lowY, int index, IYL nxt) {                  •
143           this.lowY = lowY; this.index = index; this.nxt = nxt;     •
144      }                                                              •
145    }                                                                •
146
147   IYL updateIYL(double minY, int i, IYL ih) {                       •
148      // Remove siblings that are hidden by the new subtree.
149      while(ih != null && minY >= ih.lowY) ih = ih.nxt;              •
150      // Prepend the new subtree.
151      return new IYL(minY,i,ih);                                     •
152   }                                                                 •
```