Improving Walker's Algorithm to Run in Linear Time

Christoph Buchheim[1], Michael Jünger[1], and Sebastian Leipert[2]

[1] Universität zu Köln, Institut für Informatik,
Pohligstraße 1, 50969 Köln, Germany
{buchheim,mjuenger}@informatik.uni-koeln.de
[2] caesar research center,
Friedensplatz 16, 53111 Bonn, Germany
leipert@caesar.de

Abstract. The algorithm of Walker [3] is widely used for drawing trees of unbounded degree, and it is widely assumed to run in linear time, as the author claims in his article. But the presented algorithm obviously needs quadratic runtime. We explain the reasons for that and present a revised algorithm that creates the same layouts in linear time.

1 Introduction

Since Walker presented his article [3] on drawing rooted ordered trees of unbounded degree, this topic is considered as a solved problem of Automatic Graph Drawing. In 1979, Wetherell and Shannon [4] presented a linear time algorithm for drawing binary trees, satisfying the following aesthetic requirements: The y-coordinate of a node corresponds to its level, so that the hierarchical structure of the tree is displayed; the left child of a node is placed to the left of the right child, i.e., the order of the children is displayed; finally, each parent node is centered over its children. Nevertheless, this algorithm showed some deficiencies. In 1981, Reingold and Tilford [1] improved the Wetherell-Shannon algorithm by adding the following feature: Each pair of isomorphic subtrees is drawn identically up to translation, i.e., the drawing does not depend on the position of a subtree within the complete tree. They also made the algorithm symmetrical: If all orders of children in a tree are reversed, the computed drawing is the reflected original one. The width of the drawing is not always minimized subject to these conditions, but it is close to the minimum in general. The algorithm of Reingold and Tilford runs in linear time, too.

Extending this algorithm to rooted ordered trees of unbounded degree in a straightforward way produces layouts where some subtrees of the tree may get clustered on a small space, even if they could be dispersed much better. This problem was solved in 1990 by the algorithm of Walker [3], which spaces out subtrees whenever possible. Unfortunately, the runtime of the algorithm presented in [3] is quadratic, in contrary to the author's assertion. In the present article, we close this gap by giving an adjustment of Walker's algorithm that does not affect the computed layouts but yields linear runtime.

In the next section, we gather the basic notation about trees and state the aesthetic criteria guiding the algorithms to be dealt with. In Sect. 3, we explain the Reingold-Tilford algorithm. In Sect. 4, we describe the idea of Walker's algorithm and point out the non-linear parts. We improve these parts in order to get a linear time algorithm in Sect. 5. Finally, we state the complete revised algorithm in App. A.

2 Preliminaries

We define a *(rooted) tree* as a directed acyclic graph with a single source, called the *root* of the tree, such that there is a unique directed path from the root to any other node. The *level* of a node is the length of this path. For each edge (v, w), we call v the *parent* of w and w a *child* of v. If w_1 and w_2 are two different children of v, we say that w_1 and w_2 are *siblings*. Each node w on the path from the root to a node v is called an *ancestor* of v, while v is called a *descendant* of w. A *leaf* of the tree is a sink of the graph, i.e., a node without children. If v_- and v_+ are two nodes such that v_- is not an ancestor of v_+ and vice versa, the *greatest uncommon ancestors* of v_- and v_+ are defined as the unique ancestors w_- and w_+ of v_- and v_+, respectively, such that w_- and w_+ are siblings. Each node v of a rooted tree T induces a unique subtree of T with root v.

In a *binary* tree, each node has at most two children. In an *ordered* tree, a certain order of the children of each node is fixed. The first (last) child according to this order is called the *leftmost (rightmost)* child. The *left (right) sibling* of a node v is its predecessor (successor) in the list of children of the parent of v. The *leftmost (rightmost) descendant* of v on level l is the leftmost (rightmost) node on level l belonging to the subtree induced by v. Finally, if v_1 is the left sibling of v_2, w_1 is the rightmost descendant of v_1 on some level l, and w_2 is the leftmost descendant of v_2 on the same level l, we call w_1 the *left neighbor* of w_2 and w_2 the *right neighbor* of w_1.

To *draw* a tree into the plane means to assign x- and y-coordinates to its nodes and to represent each edge (v, w) by a straight line connecting the points corresponding to v and w. When drawing a rooted tree, one usually requires the following aesthetic properties:

- (A1) The layout displays the hierarchical structure of the tree, i.e., the y-coordinate of a node is given by its level.
- (A2) The edges do not cross each other.
- (A3) The drawing of a subtree does not depend on its position in the tree, i.e., isomorphic subtrees are drawn identically up to translation.

If the trees to be drawn are ordered, we additionally require the following:

- (A4) The order of the children of a node is displayed in the drawing.
- (A5) The algorithm works symmetrically, i.e., the drawing of the reflection of a tree is the reflected drawing of the original tree.

Here, the *reflection* of an ordered tree is the tree with reversed order of children for each parent node. Usually, one tries to find a layout satisfying (A1) to (A5) with a small width, i.e., with a small range of x-coordinates.

3 Reingold and Tilford's Algorithm

For ordered binary trees, the first linear time algorithm satisfying (A1) to (A5) was presented by Reingold and Tilford [1]. This algorithm is easy to describe informally: It draws the tree recursively in a bottom-up sweep. Leaves are placed to an arbitrary x-coordinate and to the y-coordinate given by their level. After drawing the subtrees induced by the children of a parent node independently, the right one is shifted so that it is placed as close to the right of the left subtree as possible.[1] Next, the parent is placed centrally above the children, this is, at the x-coordinate given by the average x-coordinate of the children, and at the y-coordinate given by its level. Finally, the edges are inserted.

The Reingold-Tilford algorithm obviously satisfies (A1) to (A5). The difficult task is how to perform the steps described above in linear time. The crucial part of the algorithm is the shifting of the second subtree; solving the following problems takes quadratic runtime in total, if a straightforward algorithm is used: First, the computation of the new position of this subtree, second, the shifting of the subtree itself.

For the first problem, define the *left (right) contour* of a tree as the sequence of leftmost (rightmost) nodes in each level, traversed from the root to the highest level. For illustration, see Fig. 1, where nodes belonging to the contours are shaded. To place the right subtree as close to the left one as possible, we have to compare the positions of the right contour of the left subtree with the positions of the left contour of the right subtree, for all levels occuring in both subtrees. Since each node belongs to the traversed part of the left contour of the right subtree at most for one subtree combination, the total number of such comparisons is linear for the complete tree. The runtime problem is how to traverse the contours without traversing (too many) nodes not belonging to the contours. To solve this problem, Reingold and Tilford introduce *threads*. For each leaf of the tree that has a successor in the same contour, the thread is a pointer to this successor. See Fig. 1 again, where the threads are represented by dotted arrows. For every node of the contour, we now have a pointer to its successor in the contour: Either it is the leftmost (rightmost) child, or it is given by the thread. Finally, to keep the threads up to date, one has to add a new thread whenever two subtrees of different height are combined.

For the second problem, the straightforward algorithm would shift all nodes of the right subtree by the same value. Since this needs quadratic time in total, Reingold and Tilford attach a new value $mod(v)$ to each node v, its *modifier* (this technique was presented by Wetherell and Shannon [4]). The position of each node is preliminary in the bottom-up traversal of the tree. When moving a subtree rooted at a node v, only $mod(v)$ and a preliminary x-coordinate $prelim(v)$ are adjusted by the amount of shifting. The modifier of a node v is interpreted as

[1] For simplicity, we assume throughout this paper that all nodes have the same dimensions and that the minimal distance required between neighbors is the same for each pair of neighbors. Both restrictions can be relaxed easily, since we will always compare a single pair of neighbors

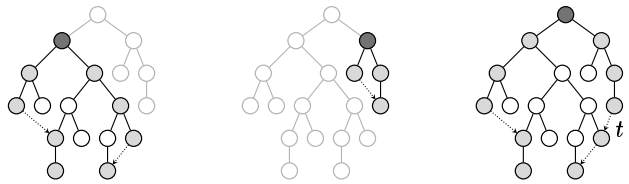


Fig. 1. Combining two subtrees and adding a new thread t

a value to be added to all preliminary x-coordinates in the subtree rooted at v, except for v itself. Thus, the real position of a node is its preliminary position plus the aggregated modifier $\mathit{modsum}(v)$ given by the sum of all modifiers on the path from the parent of v to the root. To compute all real positions in linear time, the tree is traversed in a top-down fashion at the end.

When comparing contour nodes to compute the new position of the right subtree, we need the real positions of these nodes, too. For runtime reasons, we may not sum up modifiers on the paths to the root. Therefore, modifiers are used in leafs as well. A modifier of a leaf v with a thread to a node w stores the difference between $\mathit{modsum}(w)$ and $\mathit{modsum}(v)$. Since new threads are added after combining two subtrees, these modifier sums can be computed while traversing the contours of the two subtrees. We have to traverse not only the inside contours but also the outside contours for computing the modifier sums, since v is a node of the outside contour. Now the aggregated modifiers can be computed as the sums of modifiers along the contours instead of the paths to the root.

4 Walker's Algorithm

For drawing trees of unbounded degree, the Reingold-Tilford algorithm could be adjusted easily by traversing the children from left to right, placing and shifting the corresponding subtrees one after another. However, this violates property (A5): The subtrees are placed as close to each other as possible and small subtrees between larger ones are piled to the left, see Fig. 2(a). A simple trick to avoid this effect is to add an analogous second traversal from right to left, see Fig. 2(b), and to take average positions after that. This algorithm satisfies (A1) to (A5), but smaller subtrees are usually clustered then, see Fig. 2(c).

To obtain a layout where smaller subtrees are spaced out evenly, as for example in Fig. 2(d), Walker [3] proposed the following proceeding, see Fig. 3: The subtrees of the current root are processed one after another from left to right. First, each child of the current root is placed as close to the right of its left sibling as possible. As in Reingold and Tilford's algorithm, the left contour of the current subtree is then traversed top down in order to compare the positions of its nodes to those of their left neighbors. Whenever two conflicting neighbors v_-

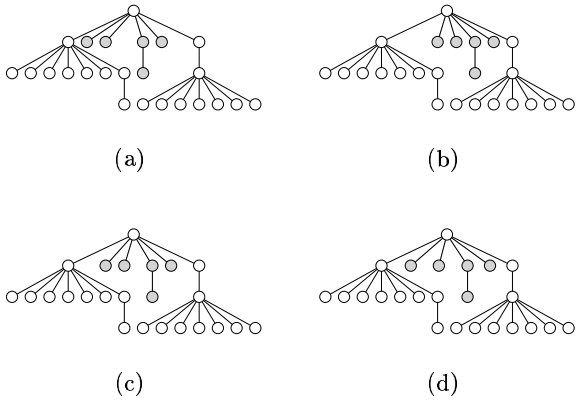


Fig. 2. Extending the Reingold-Tilford algorithm to trees of unbounded degree

and v_+ are detected, forcing v_+ to be shifted to the right by an amount of *shift*, we apply an appropriate shift to all smaller subtrees between the subtrees containing v_- and v_+. More precisely, let w_- and w_+ be the lowest uncommon ancestors of v_- and v_+. Notice that both w_- and w_+ are children of the current root. Let *subtrees* be the number of children of the current root between w_- and w_+ plus 1. Spacing out the subtrees is shifting the subtree rooted at the i-th child to the right of w_- by an amount of $i \cdot \mathit{shift}/\mathit{subtrees}$, for $i = 1, \ldots, \mathit{subtrees}$. Observe that subtrees may be shifted several times by this algorithm, even while adding a single subtree. It is easy to see that this algorithm satisfies (A5).

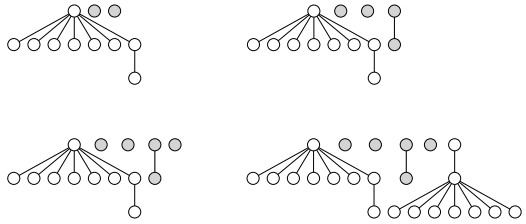


Fig. 3. Spacing out the smaller subtrees

Unfortunately, many parts of the algorithm presented in [3] do not run in linear time. Some of them are easy to improve, for example by using Reingold and Tilford's ideas, and some afford new ideas. All problems concern Walker's procedure APPORTION, see pages 695 and 696 in [3]. In the following, we list the critical parts. In the next section, we will explain how to change these in order to obtain linear runtime.

Traversing the right contour: A recursive function GETLEFTMOST is used to find the leftmost descendant of a given node v on a given level l. If the level of v is l, the algorithm returns v. Otherwise, GETLEFTMOST is applied recursively to all children of v, from left to right. The aggregated runtime of GETLEFTMOST is not linear in general. To prove that, we present a series of trees T_k such that the number of nodes in T_k is $n \in \Theta(k^2)$, but the total number of GETLEFTMOST calls is $\Theta(k^3)$. Since $k \in \Theta(n^{1/2})$, this shows that the total runtime of GETLEFTMOST is $\Omega(k^3) = \Omega(n^{3/2})$. The tree T_k is defined as follows (see Fig. 4(a) for $k = 3$): Beginning at the root, there is a chain of $2k$ nodes, each of the last $2k-1$ being the right or only child of its predecessor. For $i = 1, \ldots, k$, the i-th node in this chain has another child to the left; this child is the first node of a chain of $2(k-i)+1$ nodes. The number of nodes in T_k is

$$2k + \sum_{i=1}^{k} (2(k-i)+1) = 2k + k(k-1) + k \in \Theta(k^2) \ .$$

Now for each $i = 0, \ldots, k-1$, we have to combine two subtrees when visiting the node on the right contour of T_k on level i. In this combination, the highest common level of the subtrees is $2k-i-1$, and by construction of T_k we always have to apply GETLEFTMOST to every node of the right subtree up to this level. The number of these nodes is

$$k - i + \sum_{j=0}^{k-i-1} (2j) = (k-i) + (k-i)(k-i-1) = (k-i)^2 \ ,$$

hence the total number of GETLEFTMOST calls for all combinations is

$$\sum_{i=0}^{k-1} (k-i)^2 = \sum_{i=1}^{k} i^2 = k(k+1)(2k+1)/6 \in \Theta(k^3) \ .$$

Finding the ancestors and summing up modifiers: This part of the algorithm is obviously quadratic. When adjusting the current subtree to the left subforest, the greatest uncommon ancestors of the possibly conflicting neighbors are computed for each level by traversing the graph up to the current root, at the same time computing the modifier sums. Since the distance of the levels grows linearly, the total number of steps is in $\Omega(n^2)$.

Counting and shifting the smaller subtrees: When shifting the current subtree to the right because of a conflict with a subtree to the left, the procedure APPORTION also shifts all smaller subtrees in between immediately. Furthermore, the number of these subtrees is computed by counting them one by one. Both has an aggregated runtime of $\Omega(n^{3/2})$, as the following example shows. Let the tree T^k be constructed as follows (see Fig. 4(b) for $k = 3$): Add k children to the root. The i-th child, counted $i = 1, \ldots, k$ from left to right, is root of a chain of i nodes. Between each pair of these children, add k children being leafs. The leftmost child of the root has $2k+5$ children, and up to level $k-1$, every

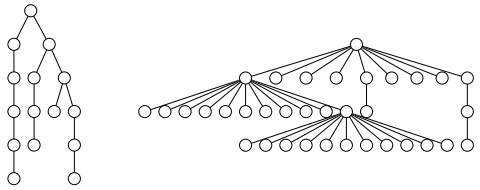
rightmost child of the $2k+5$ children has again $2k+5$ children. The number of nodes of T^k is

$$1 + \sum_{i=1}^{k} i + (k-1)k + (k-1)(2k+5) \in \Theta(k^2) \ .$$

Furthermore, by construction of the left subtree, adding the i-th subtree chain for $i = 2, \ldots, k$ results in a conflict with the left subtree on level i. Hence all $(i-1)(k+1)-1$ smaller subtrees between the two conflicting ones are counted and shifted. Thus, the total number of counting and shifting steps is

$$\sum_{i=2}^{k}((i-1)(k+1)-1) = (k+1)k(k-1)/2 - k + 1 \in \Theta(k^3) \ .$$

As in the last example, we derive that counting and shifting needs $\Omega(n^{3/2})$ time in total.



(a) T_3 (b) T^3

Fig. 4. Examples proving the non-linear runtime of Walker's algorithm

5 Improving Walker's Algorithm

The algorithm of Walker can be improved to linear runtime without affecting the computed layouts. In this section, we explain how to do that. For a closer look see App. A, where we state the complete algorithm in a pseudocode style.

Traversing the contours and summing up modifiers: This can be done exactly as in the case of binary trees by using threads, see Sect. 3. The fact that the left subforest is no tree in general does not create any additional difficulty.

Finding the ancestors: The problem of finding the greatest uncommon ancestors w_- and w_+ of two nodes v_- and v_+ can be solved by the algorithm of Schieber and Vishkin [2]. For each pair of nodes, this algorithm can determine

the greatest uncommon ancestors in constant time, after an $O(n)$ preprocessing step. However, in our application, a much simpler algorithm can be applied. First observe that we know the right ancestor w_+ anyway; it is just the root of the current subtree. Furthermore, as v_+ is always the right neighbor of v_- in our algorithm, the left one of the greatest uncommon ancestors only depends on v_-. Thus we may shortly call it *the ancestor* of v_- in the following. We use a node pointer $ancestor(x)$ for each node x to save its ancestor and initialize it to x itself. Observe that this value is not correct for rightmost children, but we do not need the correct value w_- of $ancestor(v_-)$ until the right neighbor v_+ of v_- is added, i.e., until the current root is the parent node of w_-. Hence assume that we are placing the subtrees rooted at the children of v from left to right. Since tracing all $ancestor(x)$ consumes too much time, we use another node pointer *defaultAncestor*. Our aim is to have the following property (*) for all nodes v_- on the right contour of the left subforest after each subtree addition: If $ancestor(v_-)$ is up to date, i.e., is a child of v, then it points to the correct ancestor w_- of v_-; otherwise, the correct ancestor is *defaultAncestor*. We start with placing the first subtree, rooted at w, which does not require any ancestor calculations. After that, we set *defaultAncestor* to w. Since all pointers $ancestor(x)$ of the left subtree either point to w or to a node of a higher level, the desired property (*) holds, see Fig. 5(a). After placing the subtree rooted at another child w' of v, we distinguish two cases: If the subtree rooted at w' is smaller than the left subforest, we can actualize $ancestor(x)$ for all nodes x on its right contour by setting it to w'. By this, we obviously keep (*), see Fig. 5(b). Otherwise, if the new subtree is larger than the left subforest, we may not do the same because of runtime. But now it suffices to set *defaultAncestor* to w', since again all pointers $ancestor(x)$ of the subtree induced by w' either point to w' or to a node of a higher level, and all other subtrees in the left subforest are hidden. Hence we have (*) again, see Fig. 5(c).

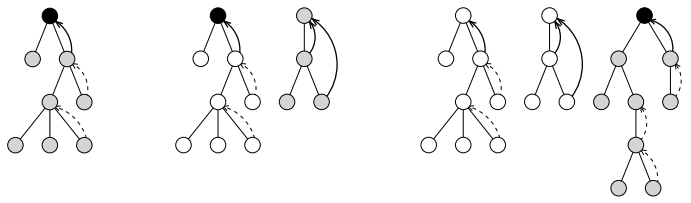


Fig. 5. Adjusting ancestor pointers when adding new subtrees: The pointer $ancestor(x)$ is represented by a solid arrow if up to date and by a dashed arrow if expired. In the latter case, the *defaultAncestor* is used, it is drawn black. When adding a small subtree, all ancestor pointers $ancestor(x)$ of its right contour are updated. When adding a large subtree, only *defaultAncestor* is updated

Counting the smaller subtrees: For that, we just have to number the children of each node consecutively; then the number of smaller subtrees between the two greatest uncommon ancestors w_- and w_+ is the number of w_+ minus the number of w_- minus 1. Hence it can be computed in constant time (after a linear time preprocessing step to compute all child numbers).

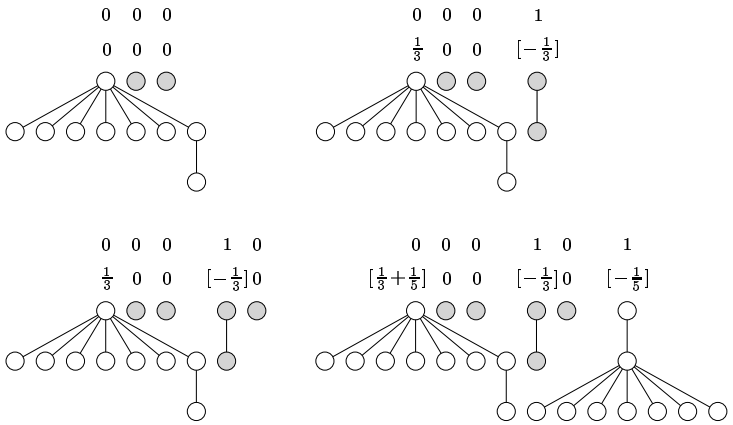


Fig. 6. Aggregating the shifts: The top number at node x indicates the value of $\mathit{shift}(x)$, the bottom number indicates the value of $\mathit{change}(x)$

Shifting the smaller subtrees: In order to get a linear runtime, we will shift each subtree at most once while it is not the currently added subtree. The currently added subtree, however, may be shifted whenever it conflicts with a subtree to the left, using the fact that shifting a single subtree is done in constant time (recall that we only have to adjust $\mathit{prelim}(w_+)$ and $\mathit{mod}(w_+)$). Furthermore, shifting the current subtree immediately is necessary to keep the right contour of the left subforest up to date. All shiftings of non-current subtrees are performed in a single traversal after all subtrees of the current root have been placed. To memorize the shiftings at the moment they arise, we use real numbers $\mathit{shift}(x)$ and $\mathit{change}(x)$ for each node x, both set to zero at the beginning. Assume that the subtree rooted at w_+ is the subtree currently placed, and that a conflict with the subtree rooted at w_- forces the current subtree to move to the right by an amount of *shift*. Let *subtrees* be the number of subtrees between w_- and w_+ plus 1. According to Walker's idea, the i-th of these subtrees has to be moved by $i \cdot \mathit{shift}/\mathit{subtrees}$. We save this by increasing $\mathit{shift}(w_+)$ by *shift*, decreasing $\mathit{change}(w_+)$ by $\mathit{shift}/\mathit{subtrees}$, and increasing $\mathit{change}(w_-)$ by $\mathit{shift}/\mathit{subtrees}$. The interpretation of this is the following: To the left of node w_+, the nodes are shifted by an amount initialized to *shift*, but this amount starts decreasing by $\mathit{shift}/\mathit{subtrees}$ per subtree at node w_+ and ends decreasing at w_-, where it is

zero. The trick is to aggregate the shifts: Since the decrease in the amount of shifting is linear, we can add all these decreases in one array, see Fig. 6 for an example. Finally, we execute all shifts in a single traversal of the children of the current root as follows, see Fig. 7: We use two real values *shift* and *change* to store the shifts and the decreases of shift per subtree, respectively, both set to zero at the beginning. Then we traverse the children from right to left. When visiting child v, we move v to the right by *shift* (i.e., we increase $prelim(v)$ and $mod(v)$ by *shift*), increase *change* by $change(v)$, and increase *shift* by $shift(v)$ and by *change*. Then we go on to the left sibling of v. It is easy to see that this algorithm shifts each subtree by the correct amount.

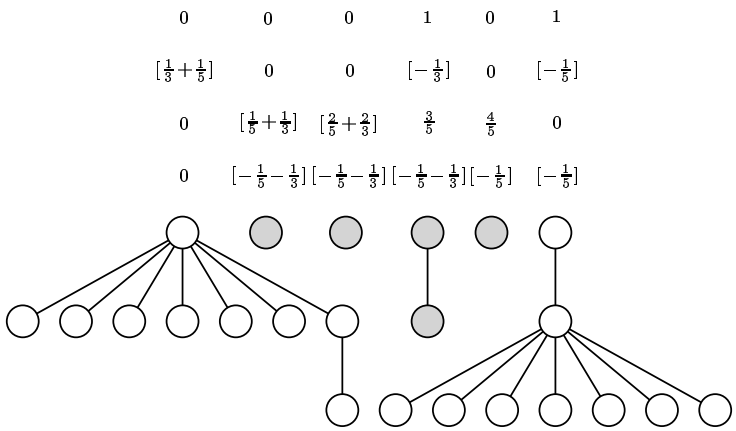


Fig. 7. Executing the shifts: The new numbers at node x indicate the values of *shift* and *change* before shifting x, respectively

References

1. Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.
2. Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. In *Proceedings of the Third Aegean Workshop on Computing*, volume 319 of *Lecture Notes in Computer Science*, pages 111–123. Springer-Verlag, 1988.
3. John Q. Walker II. A node-positioning algorithm for general trees. *Software – Practice and Experience*, 20(7):685–705, 1990.
4. Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, 5(5):514–520, 1979.

A The Complete Revised Algorithm

In this appendix, we list the complete revised algorithm described in the preceding sections. The algorithm TREELAYOUT first initializes modifiers, threads, and ancestors, then it starts the bottom-up and top-down traversals of the tree (called FIRSTWALK and SECONDWALK following Walker).

```
TREELAYOUT(T)

for all nodes v of T
    let mod(v) = thread(v) = 0
    let ancestor(v) = v
let r be the root of T
FIRSTWALK(r)
SECONDWALK(r, −prelim(r))
```

Calling FIRSTWALK(v) computes a preliminary x-coordinate for v. Before that, FIRSTWALK is applied recursively to the children of v, as well as the function APPORTION. After spacing out the children by calling EXECUTESHIFTS, the node v is placed to the midpoint of its outermost children.

```
FIRSTWALK(v)

if v is a leaf
    let prelim(v) = 0
else
    let defaultAncestor be the leftmost child of v
    for all children w of v from left to right
        FIRSTWALK(w)
        APPORTION(w, defaultAncestor)
    EXECUTESHIFTS(v)
    let midpoint =
        ½(prelim(leftmost child of v) + prelim(rightmost child of v))
    if v has a left sibling w
        let prelim(v) = prelim(w) + distance
        let mod(v) = prelim(v) − midpoint
    else
        let prelim(v) = midpoint
```

The procedure APPORTION (again following Walker's notation) is the core of the algorithm. Here, a new subtree is combined with the previous subtrees. As explained in Sect. 3, threads are used to traverse the inside and outside contours of the left and right subtree up to the highest common level. The vertices used for the traversals are v^i_+, v^i_-, v^o_-, and v^o_+, where the superscript o means outside and i means inside, the subscript $-$ means left subtree and $+$ means right subtree. For summing up the modifiers along the contour (see Sect. 3 again), we use respective variables s^i_+, s^i_-, s^o_-, and s^o_+. Whenever two nodes of the inside contours conflict, we compute the left one of the greatest uncommon ancestors using the function

ANCESTOR and call MOVESUBTREE to shift the subtree and prepare the shifts of smaller subtrees. Finally, we add a new thread (if necessary) as explained in Sect. 3. Observe that we have to adjust $ancestor(v^o_+)$ or *defaultAncestor* to keep property (*), see Sect. 5. For function NEXTLEFT and NEXTRIGHT see below.

```
APPORTION(v,defaultAncestor)

if v has a left sibling w
    let v^i_+ = v^o_+ = v
    let v^i_- = w
    let v^o_- be the leftmost sibling of v^i_+
    let s^i_+ = mod(v^i_+)
    let s^o_+ = mod(v^o_+)
    let s^i_- = mod(v^i_-)
    let s^o_- = mod(v^o_-)
    while NEXTRIGHT(v^i_-) ≠ 0 and NEXTLEFT(v^i_+) ≠ 0
        let v^i_- = NEXTRIGHT(v^i_-)
        let v^i_+ = NEXTLEFT(v^i_+)
        let v^o_- = NEXTLEFT(v^o_-)
        let v^o_+ = NEXTRIGHT(v^o_+)
        let ancestor(v^o_+) = v
        let shift = (prelim(v^i_-) + s^i_-) - (prelim(v^i_+) + s^i_+) + distance
        if shift > 0
            MOVESUBTREE(ANCESTOR(v^i_-,v,defaultAncestor),v,shift)
            let s^i_+ = s^i_+ + shift
            let s^o_+ = s^o_+ + shift
        let s^i_- = s^i_- + mod(v^i_-)
      let s^i_+ = s^i_+ + mod(v^i_+)
      let s^o_- = s^o_- + mod(v^o_-)
      let s^o_+ = s^o_+ + mod(v^o_+)
if NEXTRIGHT(v^i_-) ≠ 0 and NEXTRIGHT(v^o_+) = 0
    let thread(v^o_+) = NEXTRIGHT(v^i_-)
    let mod(v^o_+) = mod(v^o_+) + s^i_- - s^o_+
if NEXTLEFT(v^i_+) ≠ 0 and NEXTLEFT(v^o_-) = 0
    let thread(v^o_-) = NEXTLEFT(v^i_+)
    let mod(v^o_-) = mod(v^o_-) + s^i_+ - s^o_-
    let defaultAncestor = v
```

The function NEXTLEFT(v) is used to traverse the left contour of a subtree (or subforest), see Sect. 3. It returns the successor of v on this contour. This successor is either given by the leftmost child of v or by the thread of v. The function returns 0 if and only if v is on the highest level of its subtree.

```
NEXTLEFT(v)

if v has a child
    return the leftmost child of v
else
    return thread(v)
```

The function NEXTRIGHT(v) works analogously.

```
NextRight(v)

if v has a child
    return the rightmost child of v
else
    return thread(v)
```

Shifting a subtree can be done in linear time if performed as explained in Sect. 5. Calling MOVESUBTREE(w_-,w_+,*shift*) first shifts the current subtree, rooted at w_+. This is done by increasing $prelim(w_+)$ and $mod(w_+)$ by *shift*. All other shifts, applied to the smaller subtrees between w_- and w_+, are performed later by EXECUTESHIFTS. To prepare this, we have to adjust $change(w_+)$, $shift(w_+)$, and $change(w_-)$.

```
MoveSubtree(w_-,w_+,shift)

let subtrees = number(w_+) − number(w_-)
let change(w_+) = change(w_+) − shift / subtrees
let shift(w_+) = shift(w_+) + shift
let change(w_-) = change(w_-) + shift / subtrees
let prelim(w_+) = prelim(w_+) + shift
let mod(w_+) = mod(w_+) + shift
```

Now EXECUTESHIFTS(v) only needs one traversal of the children of v to execute all shifts computed and memorized in MOVESUBTREE. This is explained in Sect. 5 again.

```
ExecuteShifts(v)

let shift = 0
let change = 0
for all children w of v from right to left
    let prelim(w) = prelim(w) + shift
    let mod(w) = mod(w) + shift
    let change = change + change(w)
    let shift = shift + shift(w) + change
```

The function ANCESTOR returns the left one of the greatest uncommon ancestors of v^i_- and its right neighbor, as described in Sect. 5.

```
Ancestor(v^i_-,v,defaultAncestor)

if ancestor(v^i_-) is a sibling of v
    return ancestor(v^i_-)
else
    return defaultAncestor
```

Finally, the SECONDWALK is used to compute all real x-coordinates by summing up the modifiers recursively.

```
SECONDWALK(v, m)

let x(v) = prelim(v) + m
let y(v) be the level of v
for all children w of v
    SECONDWALK(w, m + mod(v))
```