Tidier Drawings of Trees

EDWARD M. REINGOLD AND JOHN S. TILFORD

Abstract—Various algorithms have been proposed for producing tidy drawings of trees—drawings that are aesthetically pleasing and use minimum drawing space. We show that these algorithms contain some difficulties that lead to aesthetically unpleasing, wider than necessary drawings. We then present a new algorithm with comparable time and storage requirements that produces tidier drawings. Generalizations to forests and m-ary trees are discussed, as are some problems in discretization when alphanumeric output devices are used.

Index Terms—Data structures, trees, tree structures.

INTRODUCTION

In a recent article [6], Wetherell and Shannon presented algorithms for producing "tidy" drawings of trees—drawings that use as little space as possible while satisfying certain aesthetics. The basic task is the assignment of x and y coordinates to each node of a tree, after which a straightforward plotting or printing routine generates a drawing of the tree. Wetherell and Shannon give three aesthetics in an attempt to define a "tidy" drawing of a binary tree.

Aesthetic 1: Nodes at the same level of the tree should lie along a straight line, and the straight lines defining the levels should be parallel.

Aesthetic 2: A left son should be positioned to the left of its father and a right son to the right.

Aesthetic 3: A father should be centered over its sons.

Although not mentioned in [6], Aesthetic 1 was also meant to require that the relative order of nodes across any level be the same as in the level order traversal of the tree. This can be shown to guarantee that edges in the tree do not intersect except at nodes.

The algorithms presented in [6] try to achieve these aesthetics while at the same time minimizing width. Similar algorithms were developed by Sweet [3] for use in his thesis, but were never published. The basic algorithm of [6] proceeds as follows. First, store in each node its level in the tree; this is essentially its y coordinate. Then traverse the tree in postorder, pausing at each node to give it an x coordinate. Initially, a provisional x coordinate is assigned according to this rule: if the node is a leaf, give it the next available position on its level; if it has only a left son, give it a position one unit to the right of its son; if it has only a right son, give it a position one unit to the left of its son; otherwise (the node has two sons) give it the average of their positions. Meanwhile, keep track of the next available position on each level with an array NEXT_POS, indexed by level, in which each value is set to two greater than the coordinate of the last node assigned on the corresponding level.

Manuscript received April 10, 1980.

The authors are with the Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801.

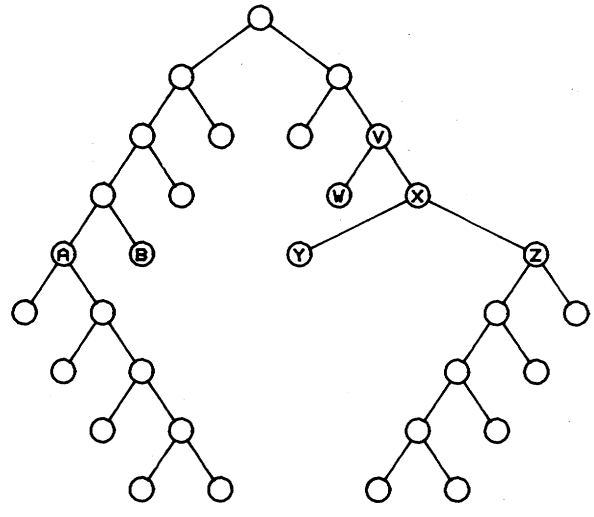


Fig. 1. Final positioning of example tree as drawn by Algorithm WS.

If a provisional position is less than the next available position on that level, the node is given the next available position, and its subtrees are shifted to the right so as to be properly positioned relative to it. Actually, the amount of the shift is just stored in the current node and applied with all the other shifts during a subsequent preorder traversal. Whenever a shift is applied to a node, all nonleaf nodes to its right on the same level must have at least the same shift applied to them and their subtrees (because the nodes in those subtrees were positioned without knowledge of shifts that would occur above them). This necessitates another array, indexed by level, containing the most recent shift applied on each level.

DIFFICULTIES

Algorithm WS works well in many cases; however, it contains an important deficiency. It can produce drawings that are not really pleasing and that can be made narrower within the constraints of the aesthetics. In Fig. 1, for example, nodes Y and Z are too far apart; instead, the tree ought to be drawn as shown in Fig. 2 because that tree is both narrower and aesthetically more pleasing, in fact, "tidier."

The problem of Algorithm WS in the drawing of Fig. 1 is the influence of the fixed left margin, defined by the values of the array NEXT_POS. Since node Y is a leaf, it receives the next available position on its level, 6. Now the lower part of the

0098-5589/81/0300-0223$00.75

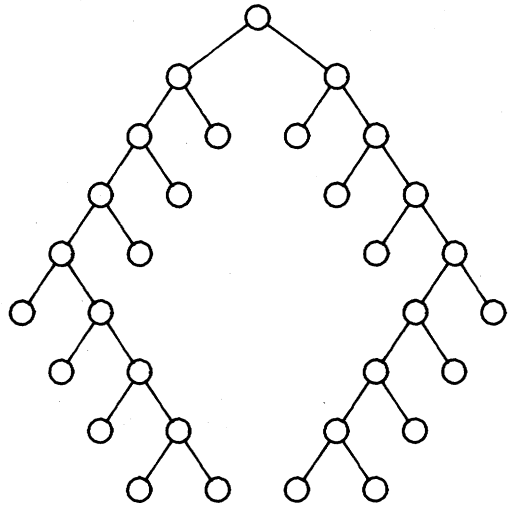


Fig. 2. Example tree as drawn by Algorithm TR.

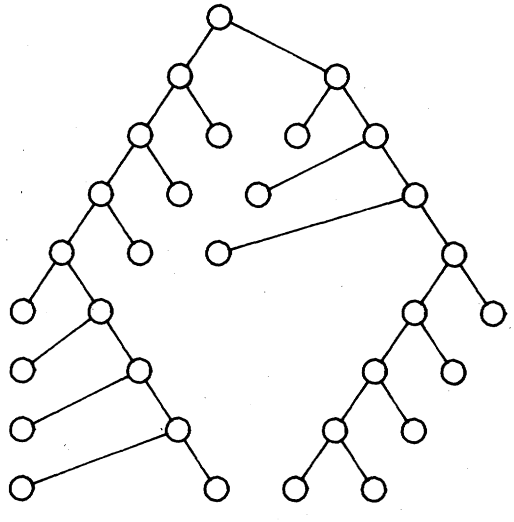


Fig. 3. Example tree drawn by a modified Algorithm WS.

right subtree is built as usual, with Z being placed at 12. X, the father of Y and Z, is given the average of their positions, i.e., 9, and V receives the average of the positions of W and X, which is 8. This is too far to the left according to NEXT_POS, so the subtree rooted at V is shifted two units to the right. The resulting tree is two units wider than necessary. The culprit is the empty space in the middle of the tree; it caused Y to be placed too far to the left when it should have been the minimum distance from Z (as A is from B). As the number of nodes increases, this anomalous behavior of Algorithm WS can worsen.

Wetherell and Shannon present a modification to Algorithm WS that guarantees minimum width drawings at the expense of Aesthetic 3. Although the drawing it produces for the sample tree (see Fig. 3) is not too wide, the drawing of Fig. 2 is much better. Vaucher [5], independently of [3] and [6], developed a tree printing algorithm that seems to avoid this problem but does not satisfy the additional aesthetic constraint introduced in the next section.

As our example illustrates, the difficulty with Algorithm WS

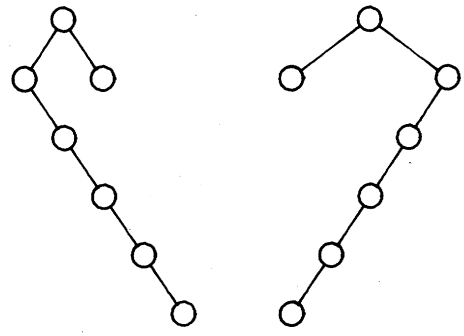


Fig. 4. A tree and its mirror image positioned by Algorithm WS.

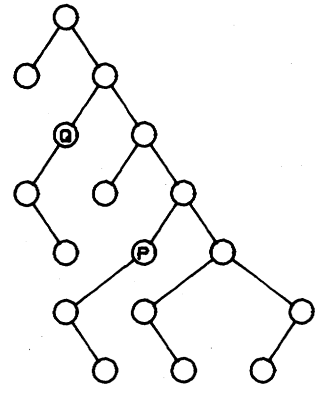


Fig. 5. A tree for which the narrowest drawing that satisfies Aesthetics 1-3 violates Aesthetic 4. The subtrees rooted at P and Q are isomorphic, but must be drawn nonisomorphically (as shown) to obtain a minimum width drawing.

stems from the fact that the shape of a subtree is influenced by the positioning of nodes outside that subtree; Sweet [3] made a similar observation. As a consequence, symmetric trees may be drawn asymmetrically, or more generally, a tree and its reflection will not always produce mirror image drawings; even the same subtree may appear differently in different parts of the tree. Fig. 4 shows a small tree and its reflection whose drawings by Algorithm WS are not mirror images.

A New Aesthetic and Algorithm

It is certainly desirable that a symmetric tree be drawn symmetrically; therefore, we introduce a new aesthetic that guarantees this (along with a somewhat stronger requirement).

Aesthetic 4: A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree.

We pay a price for this aesthetic in terms of the width of the tree. Fig. 5 illustrates a tree for which the narrowest drawing that satisfies Aesthetics 1-3 violates Aesthetic 4. Nevertheless, we consider Aesthetic 4 to be more important than minimum width since the shape of the printed tree and its reflection ought to be independent of its surroundings to aid in human perception. In any case, with the exception of the theoretically interesting but impractical linear programming technique of [2], the published tree printing algorithms all fail to produce minimum width placements, even without the stricture of Aesthetic 4.

Satisfying Aesthetic 4 requires an algorithm in which nodes outside a subtree do not interfere with the relative positionings of nodes in the subtree, so that the inherent asymmetry of the postorder traversal will not be manifested in the drawing. To this end we propose Algorithm TR, based on the following heuristic: two subtrees of a node should be formed independently, and then placed as close together as possible. By requiring that the subtrees be rigid at the time they are put together, we avoid the undesirable effects that can accrue from positioning nodes rather than subtrees.

The above heuristic is applied during a postorder traversal as follows. At each node T, imagine that its two subtrees have been drawn and cut out of paper along their contours. We then superimpose the two subtrees at their roots and move them apart until no two points are touching. Initially their roots are separated by some agreed upon minimum distance; then at the next lower level, we push them apart until minimum separation is established there. This process continues at successively lower levels until we get to the bottom of the shorter subtree. At some levels no movement may be necessary; but at no level are the two subtrees moved closer together. When the process is complete, we fix the position of the subtrees relative to their father, which is centered over them. Assured that the subtrees cannot be placed closer together, we continue the postorder traversal. After this traversal, a preorder traversal converts the relative positionings into absolute coordinates.

This technique is similar to Algorithm WS in several ways: Aesthetics 1, 2, and 3 are satisfied; fathers are visited after sons so that they may be centered over them; and empty subtrees are handled as in Algorithm WS. The essential difference is that mirror image trees produce mirror image drawings and a subtree is drawn identically wherever it occurs.

Although the idea behind Algorithm TR is simple, implementation is somewhat tricky, because we need an efficient way to follow the contour of a subtree as it is compared to its brother subtree. As Fig. 2 illustrates, simply following left and right links does not suffice. Note, however, that when the next node on the contour is not a son of the current node, the current node must be a leaf; in many applications this means that it contains two unused pointer fields. We can use one of these fields to store a temporary pointer to the next node on the contour, as long as we reserve an additional bit to distinguish it from a regular pointer. This idea is reminiscent of threaded binary trees [1], and so we call these distinguished pointers "threads." Fortunately, the maintenance of these threads is not difficult and can be done at the same time the subtrees are positioned.

We have implemented Algorithm TR in Pascal; the bulk of the work is done by the recursive postorder traversal procedure shown in Fig. 6. During the traversal the procedure performs three tasks at each node T: first it determines how close together the subtrees of T can be placed; second, it keeps track of nodes that may need to be threaded later; and third, it inserts a thread if one is required.

Since Aesthetic 3 requires that fathers be centered over sons, it suffices to include a single integer field in each node that specifies the horizontal offset between the node and each of its sons. We store this relative distance rather than the absolute location so that subtrees can be pushed apart easily. Storing this offset field in the father rather than sons may not be a good idea for m-ary trees and forests, but for reasons

```
PROCEDURE SETUP ( T : LINK;                    (* ROOT OF SUBTREE          *)
                  LEVEL : INTEGER;             (* CURRENT OVERALL LEVEL    *)
                  VAR RMOST,
                      LMOST : EXTREME );       (* EXTREME DESCENDANTS      *)

(* THIS PROCEDURE IMPLEMENTS ALGORITHM TR, ASSIGNING RELATIVE             *)
(* POSITIONINGS TO ALL NODES IN THE TREE POINTED TO BY PARAMETER T.       *)

VAR
   L,R : LINK;                                 (* LEFT AND RIGHT SONS      *)
   LR, LL, RR, RL : EXTREME;                   (* LR = RIGHTMOST NODE ON   *)
                                               (* LOWEST LEVEL OF LEFT SUBTREE *)
                                               (* AND SO ON                *)


   CURSEP,                                     (* SEPARATION ON CURRENT LEVEL  *)
   ROOTSEP,                                    (* CURRENT SEPARATION AT NODE T *)
   LOFFSUM, ROFFSUM : INTEGER;                 (* OFFSET FROM L & R TO T   *)

BEGIN (* SETUP *)
   IF T = NIL THEN BEGIN                       (* AVOID SELECTING AS EXTREME   *)
      LMOST.LEV := -1;
      RMOST.LEV := -1
   END ELSE BEGIN
      T^.YCOORD := LEVEL;
      L := T^.LLINK;                           (* FOLLOWS CONTOUR OF LEFT SUBTREE  *)
      R := T^.RLINK;                           (* FOLLOWS CONTOUR OF RIGHT SUBTREE *)
      SETUP ( L, LEVEL+1, LR, LL );            (* POSITION SUBTREES RECURSIVELY    *)
      SETUP ( R, LEVEL+1, RR, RL );
      IF (R=NIL) AND (L=NIL) THEN BEGIN  (* LEAF *)
         RMOST.ADDR := T;                      (* A LEAF IS BOTH THE LEFTMOST      *)
         LMOST.ADDR := T;                      (* AND RIGHTMOST NODE ON THE        *)
         RMOST.LEV := LEVEL;                   (* LOWEST LEVEL OF THE SUBTREE      *)
         LMOST.LEV := LEVEL;                   (* CONSISTING OF ITSELF             *)
         RMOST.OFF := 0;
         LMOST.OFF := 0;
         T^.OFFSET := 0
      END ELSE BEGIN                           (* T NOT A LEAF *)

         (* SET UP FOR SUBTREE PUSHING.  PLACE ROOTS OF *)
         (* SUBTREES MINIMUM DISTANCE APART.            *)

         CURSEP := MINSEP;
         ROOTSEP := MINSEP;
         LOFFSUM := 0;
         ROFFSUM := 0;

         (* NOW CONSIDER EACH LEVEL IN TURN UNTIL ONE   *)
         (* SUBTREE IS EXHAUSTED, PUSHING THE SUBTREES  *)
         (* APART WHEN NECESSARY.                       *)

         WHILE (L<>NIL) AND (R<>NIL) DO BEGIN
            IF CURSEP < MINSEP THEN BEGIN (* PUSH ? *)
               ROOTSEP := ROOTSEP + (MINSEP - CURSEP);
               CURSEP := MINSEP
            END; (* IF CURSEP < MINSEP *)

            (* ADVANCE L & R *)
            IF L^.RLINK <> NIL THEN BEGIN
               LOFFSUM := LOFFSUM + L^.OFFSET;
               CURSEP := CURSEP - L^.OFFSET;
               L := L^.RLINK
            END ELSE BEGIN
               LOFFSUM := LOFFSUM - L^.OFFSET;
               CURSEP := CURSEP + L^.OFFSET;
               L := L^.LLINK
            END;
            IF R^.LLINK <> NIL THEN BEGIN
               ROFFSUM := ROFFSUM - R^.OFFSET;
               CURSEP := CURSEP - R^.OFFSET;
               R := R^.LLINK
            END ELSE BEGIN
               ROFFSUM := ROFFSUM + R^.OFFSET;
               CURSEP := CURSEP + R^.OFFSET;
               R := R^.RLINK
            END; (* ELSE *)
         END; (* WHILE *)

         (* SET THE OFFSET IN NODE T, AND INCLUDE IT IN *)
         (* ACCUMULATED OFFSETS FOR L AND R             *)

         T^.OFFSET := (ROOTSEP + 1) DIV 2;
         LOFFSUM := LOFFSUM - T^.OFFSET;
         ROFFSUM := ROFFSUM + T^.OFFSET;

         (* UPDATE EXTREME DESCENDANTS INFORMATION *)

         IF (RL.LEV > LL.LEV) OR (T^.LLINK = NIL)  THEN BEGIN
            LMOST := RL;
            LMOST.OFF := LMOST.OFF + T^.OFFSET
         END ELSE BEGIN
            LMOST := LL;
            LMOST.OFF := LMOST.OFF - T^.OFFSET
         END;
         IF (LR.LEV > RR.LEV) OR (T^.RLINK = NIL) THEN BEGIN
            RMOST := LR;
            RMOST.OFF := RMOST.OFF - T^.OFFSET
         END ELSE BEGIN
            RMOST := RR;
            RMOST.OFF := RMOST.OFF + T^.OFFSET
         END;

         (* IF SUBTREES OF T WERE OF UNEVEN HEIGHTS, CHECK *)
         (* TO SEE IF THREADING IS NECESSARY.  AT MOST ONE *)
         (* THREAD NEEDS TO BE INSERTED.                   *)

         IF (L <> NIL) AND (L <> T^.LLINK) THEN BEGIN
            RR.ADDR^.THREAD := TRUE;
            RR.ADDR^.OFFSET := ABS(( RR.OFF + T^.OFFSET) - LOFFSUM );
            IF LOFFSUM - T^.OFFSET <= RR.OFF THEN
               RR.ADDR^.LLINK := L
            ELSE
               RR.ADDR^.RLINK := L
         END ELSE IF (R <> NIL) AND (R <> T^.RLINK)  THEN BEGIN
            LL.ADDR^.THREAD := TRUE;
            LL.ADDR^.OFFSET := ABS(( LL.OFF - T^.OFFSET ) - ROFFSUM );
            IF ROFFSUM + T^.OFFSET >= LL.OFF THEN
               LL.ADDR^.RLINK := R
            ELSE
               LL.ADDR^.LLINK := R
         END

      END; (* OF IF NOT LEAF *)

   END; (* OF T <> NIL *)

END; (* PROCEDURE SETUP *)
```

Fig. 6. Procedure SETUP assigns relative x coordinates to all nodes during a postorder traversal.

that will become apparent this method is preferable for binary trees. A node of the tree, then, has the following format (for the sake of clarity we have reserved space for both OFFSET and XCOORD, although clearly these can share the same space in a real implementation):

```
NODE = RECORD
    INFO    : ;              (* whatever *)
    LLINK,
    RLINK   : LINK;          (* pointers to subtrees *)
    XCOORD,
    YCOORD  : INTEGER;       (* coordinates of this node *)
    OFFSET  : INTEGER;       (* distance to each son *)
    THREAD  : BOOLEAN
END;
```

The first task is determining the separation of the subtrees. This is accomplished by the **while** loop in Fig. 6. It scans down the right contour of the left subtree and the left contour of the right subtree, computing the distance between them, and pushing them apart when necessary. Variables L and R are initially set to point to the left and right son, respectively, of T. MINSEP is a parameter that gives the minimum separation allowed between two nodes on a level; CURSEP holds the separation at the current level. ROOTSEP accumulates the required separation distance for the sons of T so that their subtrees will have a separation of at least MINSEP at all levels. LOFFSUM and ROFFSUM accumulate the total offset of the current L and R from the root; this information is needed to compute offsets of threaded nodes.

The second task is to keep track of the leftmost and rightmost nodes on the lowest level of the subtree; only these nodes could ever be threaded. Information concerning these extreme descendants is stored in the following record format:

```
EXTREME = RECORD
    ADDR : LINK;          (* address *)
    OFF  : INTEGER;       (* offset from root of subtree *)
    LEV  : INTEGER        (* tree level *)
END;
```

If the current subtree is a single node, then that node is trivially both the leftmost and rightmost node on the lowest level of the subtree; otherwise, its extreme descendants can easily be chosen from among the extreme descendants of its sons.

Threading, the third task, is needed only if the subtrees joined at the current node have different heights and neither is empty. If the left subtree, say, is taller, a thread must be inserted from the rightmost node on the lowest level of the right subtree to the rightmost node on the next lower level of the left subtree (Fig. 7). The former node is an extreme descendant that has been determined by the second task in a previous activation of SETUP, and the variable L is currently pointing to the latter node, so inserting the thread with the appropriate offset is a simple matter. The thread is stored in the left link field if the node to which it points is to its left, and in the right link field otherwise; this makes the threads transparent as the subtrees are pushed apart. This convenience is the motivation for storing offsets in fathers rather than sons. Although a node can be threaded to at most one other node, a node might be pointed to by two active threads; thus, if offsets were stored in sons, extra space would have to be reserved in each node for two thread offsets, complicating both the data structure and the algorithm.

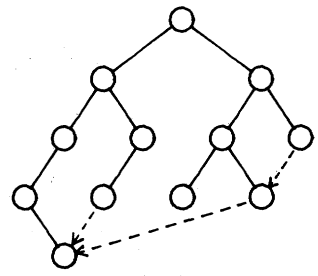


Fig. 7. The dashed lines represent threads used to follow the contours of subtrees.

```
PROCEDURE PETRIFY ( T : LINK; XPOS : COLUMN );

(* THIS PROCEDURE PERFORMS A PREORDER TRAVERSAL OF THE TREE,       *)
(* CONVERTING THE RELATIVE OFFSETS TO ABSOLUTE COORDINATES.        *)


BEGIN
   IF T <> NIL THEN BEGIN
      T^.XCOORD := XPOS;
      IF T^.THREAD THEN BEGIN
         T^.THREAD := FALSE;
         T^.RLINK := NIL;          (* THREADED NODE MUST BE A LEAF *)
         T^.LLINK := NIL
      END;
      PETRIFY ( T^.LLINK, XPOS - T^.OFFSET );
      PETRIFY ( T^.RLINK, XPOS + T^.OFFSET )
   END (* IF T <> NIL *)
END; (* PETRIFY *)
```

Fig. 8. Procedure PETRIFY converts relative positionings to absolute coordinates.

The relative positionings need to be converted to absolute coordinates after procedure SETUP is finished. A simple routine can scan the left contour of the tree to find the offset from the leftmost node to the root. Lastly, procedure PETRIFY (shown in Fig. 8) computes the final position of each node, and erases all the threads as well.

Mathematical Analysis

The algorithm does not appear to be cost effective because it must examine the contour of the subtrees of every node in the tree. However, the requirement that scanning must proceed only to the depth of the shorter subtree of each node makes the running time linear in the number of nodes in the tree and hence comparable to that of Algorithm WS.

The time required by Algorithm TR is completely determined by the **while** loop because SETUP is executed exactly once per node of the tree. To study the behavior of the **while** loop, we need a few definitions. A *binary tree* is either empty or consists of a node called the root and two subtrees T_l and T_r. The *size* n(T) of a binary tree T is the number of nodes it contains; its *height* h(T) is the number of nodes on the longest path from the root to a leaf. Thus the tree of Fig. 4 has height 6 (although by the usual definition it has height 5).

Note that the **while** loop is executed only so long as both L and R are non-**nil**, that is, only until we have dropped off the contour of the shorter subtree. The test of the loop fails once for each node, i.e., n(T) times. Let F(T) be the number of times the test of the loop succeeds, i.e., the number of times the body of the loop is executed for a tree T.

$$F(\textbf{nil}) = F(\bullet) = 0,$$

because the loop is not executed at all for empty trees or leaves. Also,

$$F\left(\begin{array}{c}\bullet\\ T_l \quad T_r\end{array}\right) = F(T_l) + F(T_r) + \min[h(T_l), h(T_r)],$$

since the number of times the loop is executed is equal to the sum of the executions for each subtree, plus one iteration at each level in the shorter of its subtrees as they are positioned.

We claim that $F(T) = n(T) - h(T)$. The proof is by induction on N, the number of nodes in T. In light of our definition of height, the truth of the claim at $N = 0$ and $N = 1$ is obvious. Suppose the claim is true for trees of less than N nodes; we show that it is then true for trees containing N nodes. Let T_l contain $k < N$ nodes; then by the inductive hypothesis,

$$\begin{aligned} F(T) &= [k - h(T_l)] + [N - k - 1 - h(T_r)] \\ &\quad + \min[h(T_l), h(T_r)] \\ &= N - 1 - h(T_l) - h(T_r) + \min[h(T_l), h(T_r)] \\ &= N - (\max[h(T_l), h(T_r)] + 1). \end{aligned}$$

But the parenthesized expression is exactly h(T), so that $F(T) = n(T) - h(T)$ and the claim holds for trees with N nodes.

In retrospect, it is easy to see why the body of the **while** loop is executed $n(T) - h(T)$ times. The code there "glues" two subtrees together; looking across levels of nodes, if there are k nodes in a level then there are $k - 1$ gluings that occur at that level. Summing over all levels except the root (there is no gluing at the root level) gives $[n(T)-1] - [h(T)-1]$ since the total number of nodes except the root is $n(T) - 1$ and the total number of levels except the root level is $h(T) - 1$.

Thus, the body of the loop is executed $n(T) - h(T)$ times and the loop test is made $2n(T) - h(T)$ times. The procedure is therefore linear in n(T). The best case is a degenerate tree in which no node has two sons; then $n(T) = h(T)$ and the loop body is never executed. The worst case is a complete binary tree for which the loop is executed about $n(T) - \lg n(T)$ times.

Generalization to m-ary Trees and Forests

Algorithm TR can easily be extended to handle m-ary trees without affecting its linear performance. Instead of making one scan at each node to the depth of the shorter of two subtrees, we make $m - 1$ scans, one for each adjacent pair of subtrees. As subtrees are combined into "clumps" (the order in which this is done is unimportant), we scan to the depth of the shorter clump in each case—thus, it is only the tallest among the m subtrees whose height is not included in the count of levels scanned. Let F(T) be the number of times the body of the generalized **while** loop is executed for an m-ary tree T; again $F(T) = n(T) - h(T)$ by a simple extension of either of the above proofs. Similarly, the number of times that the test of the **while** loop fails is $(m - 1)n(T)$.

For m-ary trees and forests we would probably not want to insist that the separations of the roots of adjacent subtrees of a given node be equal, so that if the separations were stored in the father, $m - 1$ offset fields would be required per node in m-ary trees and arbitrarily many offset fields in forests. If we store the offsets in sons instead, we will need only three offset fields per node; the extra two are needed to store the distance from nodes whose threads point to the current node.

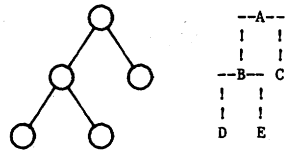


Fig. 9. A tree and its most natural discrete representation.

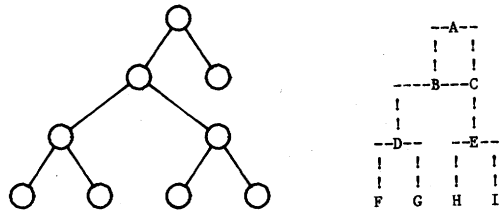


Fig. 10. A tree for which the straightforward discrete representation is inadequate.

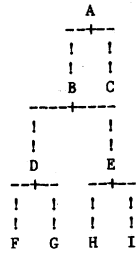


Fig. 11. A revised discrete drawing for the tree of Fig. 10.

Generalizing Algorithm TR for forests is less straightforward than for m-ary trees and depends heavily on the representation used. If a forest is represented as the binary tree that arises from the "natural correspondence" (see [1]), then the required postorder traversal of the forest amounts to an inorder traversal of the corresponding binary tree. Maintaining the threads and following the contours between adjacent pairs of subtrees is made complicated because there is no direct access from a father to its rightmost son. Following the right contour requires going through all of the sons of a node to reach the next node on the contour. Fortunately, the resulting algorithm remains linear in the number of nodes in the forest. Details of the algorithm and its analysis can be found in [4].

Discretization Problems

Wetherell and Shannon note that some difficulties arise when alphanumeric output devices are used to produce the tree drawings. On line printer devices the natural way to draw the tree of Fig. 9 is as shown. But both Algorithm WS and Algorithm TR run into a problem for many trees in which one subtree is wide and the other consists of just a few nodes, as can be seen in Fig. 10. The simple solution is to draw the tree in the way shown in Fig. 11; if we insist that branches be in the style of Fig. 9, the drawings must be made considerably wider.

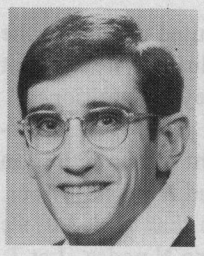
A second issue in discretization is that coordinates must be given integer values. The positioning of trees by Algorithm WS, as implemented in [6], frequently fails to satisfy Aesthetic 3 by a small amount. When the distance between brothers is an odd number, Algorithm WS truncates the computed x coordinate of the father, leaving it uncentered. By forcing a separation of even length between brothers, Algorithm TR avoids this pitfall. Algorithm WS could be similarly modified.

Conclusions

Algorithm TR produces very pleasing drawings in general. Unfortunately, the drawings can be wider than the minimum possible under the four aesthetics; this is unavoidable, however, because Supowit and Reingold [2] have shown that determining the minimum width under these aesthetics is NP-hard.

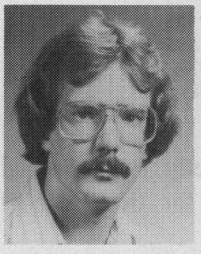
References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 2nd ed. Reading, MA: Addison-Wesley, 1972.
[2] K. J. Supowit and E. M. Reingold, "The complexity of drawing trees nicely," to be published.
[3] R. E. Sweet, "Empirical estimates of program entropy," Dep. Comput. Sci., Stanford Univ., Stanford, CA, Rep. STAN-CS-78-698, Nov. 1978; also issued as Rep. CSL-78-3, Xerox PARC, Palo Alto, CA, Sept. 1978.
[4] J. S. Tilford, "Tree drawing algorithms," M.S. thesis, Dep. Comput. Sci., Univ. Illinois, Urbana, IL, Rep. UIUC DCS-R-81-1055, 1981.
[5] J. G. Vaucher, "Pretty-Printing of Trees," *Software–Practice and Experience*, vol. 10, pp. 553–561, 1980.
[6] C. Wetherell and A. Shannon, "Tidy drawings of trees," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 514–520, 1979.



Edward M. Reingold received the B.S. degree from the Illinois Institute of Technology, Chicago, IL, and the M.S. and Ph.D. degrees from Cornell University, Ithaca, NY.

He is currently an Associate Professor of Computer Science at the University of Illinois, Urbana-Champaign. His areas of specialization are data structures and the analysis of algorithms.



John S. Tilford received the B.A. degree from DePauw University, Greencastle, IN, in 1977.

He is currently a Ph.D. candidate in computer science at the University of Illinois, Urbana-Champaign. His interests include data structures and relational database design theory.