[18] G. A. Miller, "The magical number seven, plus or minus two," *Psychological Rev.*, pp. 311-329, 1956.

[19] R. W. Motley and W. D. Brooks, "Statistical prediction of programming errors," Rome Air Develop., Final Tech. Rep., RADC-TR-77-175, May 1977.

[20] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 312-327, Sept. 1975.

[21] —, "Measuring software reliability," in *Proc. TIMS-ORSA Joint Nat. Meeting*, San Francisco, May 9-11, 1977.

[22] G. J. Myers, "An extension to the cyclomatic measure of program complexity," *ACM SIGPLAN Notices*, vol. 12, pp. 61-64, Oct. 1977.

[23] L. M. Ottenstein, "Predicting parameters of the software validation effort," Ph.D. dissertation, Purdue Univ., Aug. 1978.

[24] N. F. Schneidewind, "Analysis of error processes in computer software," in *Proc. 1975 Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 21-23, 1975; also, *ACM SIGPLAN Notices*, vol. 10, pp. 337-346, June 1975.

[25] N. F. Schneidewind and H. M. Hoffman, "Software structure and error properties: Models vs. real programs," in *Proc. TIMS-ORSA Joint Nat. Meeting*, San Francisco, May 9-11, 1977.

[26] —, "An experiment in software error data collection and analysis," in *Proc. 6th Texas Conf. Computing Systems*, Nov. 14-15, 1977.

[27] M. L. Shooman, "Operational testing and software reliability estimation during program development," in *Proc. IEEE Symp. Computer Software Reliability*, New York, Apr. 30-May 2, 1973, pp. 51-57.

[28] —, "Structured models for software reliability prediction," in *Proc. 1976 IEEE 2nd Int. Conf. Software Eng.*, San Francisco, Oct. 13-15, 1976, pp. 268-280.

[29] M. L. Shooman and M. I. Bolsky, "Types, distributions and test and correction times for programming errors," in *Proc. 1975 Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 21-23, 1975; also, *ACM SIGPLAN Notices*, vol. 10, pp. 347-357, June 1975.

[30] H. A. Simon, "How big is a chunk?" *Science*, vol. 183 (4124), pp. 482-488, Feb. 1974.

[31] A. N. Sukert, "A software reliability modeling study," Rome Air Develop., RADC-TR-76-247, Aug. 1976.

[32] J. E. Sullivan, "Measuring the complexity of computer software," MITRE MTR-2648, vol. V, June 25, 1973.

[33] T. A. Thayer, "Understanding software through empirical reliability analysis," in *AFIPS Conf. Proc.*, vol. 44, 1975, pp. 335-341.

[34] T. A. Thayer *et al.*, "Software reliability study," Rome Air Develop., Final Tech. Rep., RADC-TR-76-238, Aug. 1976.

[35] R. W. Wolverton, "The cost of developing large-scale software," *IEEE Trans. Comput.*, vol. C-28, pp. 615-636, June 1974.

[36] M. V. Zelkowitz, "Perspectives on software engineering," *ACM Computing Surveys*, vol. 10, no. 2, June 1978.

Linda M. Ottenstein was born in Sheboygan, WI, on December 22, 1950. She received the B.S., M.S., and Ph.D. degrees all in computer science from Purdue University, Lafayette, IN, in 1972, 1974, and 1978, respectively.

She is currently an Assistant Professor of Computer Science at Michigan Technological University, Houghton, MI. Her research interests include software science, software reliability, and programming methodologies.

Dr. Ottenstein is a member of the Association for Computing Machinery and the IEEE Computer Society.

Tidy Drawings of Trees

CHARLES WETHERELL AND ALFRED SHANNON

Abstract—**Trees are extremely common data structures, both as internal objects and as models for program output. But it is unusual to see a program actually draw trees for visual inspection. Although part of the difficulty lies in programming graphics devices, most of the problem arises because naive algorithms to draw trees use too much drawing space and sophisticated algorithms are not obvious. We survey two naive tree drawers, formalize aesthetics for tidy trees, and describe two algorithms which draw tidy trees. One of the algorithms may be shown to require the minimum possible paper width. Along with the algorithms proper, we discuss the reasoning behind the algorithm development.**

Index Terms—**Aesthetics, binary trees, computer graphics, drawing methods, trees.**

Manuscript received May 15, 1978; revised February 26, 1979.

C. Wetherell is with the Computing Science Group, Department of Applied Science, University of California at Davis, and the Lawrence Livermore Laboratory, Livermore, CA 94550.

A. Shannon is with the Lawrence Livermore Laboratory, Livermore, CA 94550.

COMPUTER programmers know well that trees are extremely common data structures. Trees model many real-world problems and a host of efficient and useful tree-based algorithms exist. Equally important, a good drawing of a tree is often a powerful intuitive guide to a modeled problem; indeed, some real problems consist of little more than finding and drawing a particular tree. But programmers who use trees seldom provide pretty graphic output. Users commonly tolerate listings of trees rather than demanding pictures. We attribute the lack of pictures to a dearth of published techniques for tree drawing. In this paper, we present some algorithms and heuristics for drawing tidy trees.

What, exactly, are the difficulties of drawing a tree? First, of course, each node of the tree must be assigned a position on the *drawing surface*. We assume that the drawing surface is always a flat sheet (e.g., of paper) and we will make use of no expedients such as twisting a rubber surface. Positions will be assigned using a convenient coordinate scheme, usually rectangular coordinates. Nodes are assumed to have very little extent. Once a *positioning* is known for all nodes, the tree can be drawn by a simple drafting routine conversant with the vagaries of local output devices. With this view of the problem, drawing a tree is exactly the same as assigning coordinates to the tree nodes.

But the nodes cannot be thrown willy-nilly onto the drawing surface. In due course, we shall consider aesthetics; for the moment, we shall discuss *physical limits.* No drawing surface is actually an unbounded plane, and even though a plane can be pieced together from smaller sheets, such subterfuges are ruled out in many practical cases. Computer graphics devices are *bounded* in one dimension (e.g., line printer, incremental plotter) or in both (e.g., CRT, microfiche camera). A doubly bounded sheet obviously limits the total size of potential drawings, but even a singly bounded surface may require that a drawing be wrenched about to fit properly. The assumption that nodes are (almost) geometric points also bears examination. Nodes are often labeled, sometimes elaborately, and labels may vary in size dramatically. Nodes must be positioned so that labels do not overlap and so that edges avoid nodes. Some devices (e.g., line printers) have difficulty drawing edges other than parallel to the axes; others have discretization problems severe enough that edges may not be drawn between arbitrary points.

Of the physical limits discussed, we will be most concerned with the effects of boundedness. We shall also consider problems caused by nodes with shape and size. However, we will leave problems caused by a limited drawing repertoire and by low resolution to be solved by the reader. Most installations have libraries of graphics routines which cope with such difficulties; any problems not so solved are probably so special that we could not discuss them adequately.

Aesthetics pose the other great difficulty when making tidy drawings of trees; indeed, the very notion of *tidiness* suggests that a drawing should possess a visual structure which reflects the properties of the underlying tree. Some properties which a tidy drawing might have include the following.

- Trees are planar graphs; edges should not cross.
- Trees impose a distance on the nodes; no node should be closer to the root than any of its ancestors.
- In a binary tree, the left son of a node should lie physically to the left of its parent, and similarly, right sons should lie to the right of their parents.

Naturally enough, aesthetic requirements often war against physical limits of drawing devices. Were it not so, very simple positionings would suffice for drawings even on severely limited devices; vertical columns or rectangular arrays of nodes would serve. Aesthetics may also require exorbitant computations or provide only ill-defined notions of tidiness. In such cases, heuristics rather than algorithms will be necessary, both to lower costs and to approximate tidiness.

The problem of drawing a tidy tree reduces to finding a positioning which reconciles aesthetics and physical limits. The separate consideration of aesthetics and physical limits is fruitful because we shall be able to formalize each in a simple way. In the following section, we will be more precise in our discussion.

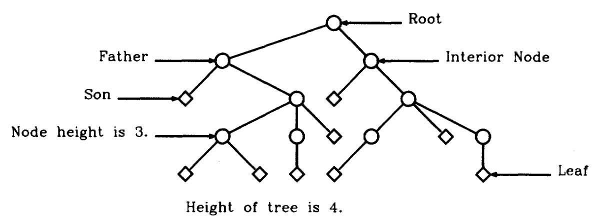


Fig. 1. An example tree.

I. A Naive Tree Drawer

A tree is formally a finite, directed, connected, acyclic graph in which every node has at most one predecessor and exactly one distinguished node, the root, has none. We rely here on the view that anyone who has seen a tree can recognize one. Fig. 1 is a typical tree with the salient features marked. An important feature of the example tree is that all nodes of the same height are drawn at the same horizontal level. Even if a drawing rotated 90° or 180° counterclockwise is preferred (trees with leaves on the left are seldom seen), the nodes at any one height will still be drawn along a line. Hence, our first aesthetic for trees follows.

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

We would like to draw trees on devices like line printers that have drawing surfaces of bounded width. The term width is used to suggest two distinct ideas: first, line printers and some other devices are not able to rotate text, so normal reading conventions imply a directionality on the surface; second, a tree implies a directionality by itself, with height running down the branches and width cutting across levels of equal height. Naturally enough, these two notions of width should be coordinated, so we have a physical limit.

Physical limit: Tree drawings should occupy as little width as possible (the height of a tree drawing is fixed by the tree itself).

The height of a node is the number of branches between that node and the root. If trees are to be drawn as in Fig. 1, the height of a node can be used to determine its y-coordinate.[1] We will also use the height of the tree to allocate storage for some of our algorithms. So our first problem is to find the height of each node. Many applications will provide node height as a by-product. If, however, node height or tree height is not known, any tree walk which visits fathers before sons can be used to find either.

The simplest tree positionings satisfying both the physical limit and Aesthetic 1 jams all nodes as far left as possible along

[1]Computer science is, as has been remarked, perhaps the only discipline in which trees wave their roots in the air and stick their leaves in the ground. Readers unhappy with this ostrichlike behavior will find it straightforward to make appropriate coordinate transformations to reorient drawings.

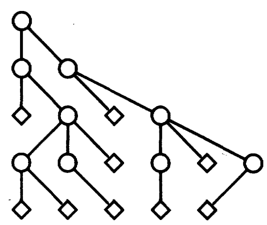


Fig. 2. The example tree repositioned by Algorithm 1.

Input: A branch **root** pointing to the root of a well-formed tree and an integer **max_height** giving the height of the tree.

Output: The **x** and **y** fields of each node are set so that the tree is positioned with as narrow a width as possible. We assume that the **height** of each node has been correctly set.

Method: A counter holding the next free x-coordinate is kept for each level of the tree. We assume that each node has a width and height of one unit and that there should be one unit gaps between the levels of the tree and between the nodes across a level. In this and later algorithms, spacing between levels or nodes can be changed by modifying the spacing constants. This algorithm positions parents before children; any tree walk is acceptable so long as each node is visited *after* its relatives to the left on the same level. All programs assume that the **father** of the **root** is nil.

```
input  root : branch;
       max_height : integer;

var    next_x : array [0..max_height] of integer;
       current : branch;
       i : integer;

begin
   for i := 0 to max_height do next_x[i] := 1; end for;
   root↑.status := 0;
   current := root;
   while current ≠ nil do
      if current↑.status = 0 then
         current↑.x := next_x[current↑.height];
         current↑.y := 2*current↑.height + 1;
         next_x[current↑.height] := next_x[current↑.height] + 2;
         for i := 1 to current↑.#_of_sons do
            current↑.son[i]↑.status := 0; end for;
         current↑.status := 1;
      elseif 1 ≤ current↑.status
            & current↑.status ≤ current↑.#_of_sons then
         current↑.status := current↑.status + 1;
         current := current↑.son[current↑.status-1];
      else  (* current↑.status > current↑.#_of_sons *)
         current := current↑.father;
      fi;
   end while;
end;  (* of Algorithm 1 *)
```

Algorithm 1. A naive positioning for trees.

each level. Fig. 2 shows the tree of Fig. 1 so repositioned. Each node's y-coordinate is simply a multiple of its height; Aesthetic 1 is obviously satisfied. (All of our more sophisticated positionings use this same method to satisfy Aesthetic 1.) The algorithm for x-coordinate assignment is nearly as easy. An array of available positions, with one entry for each tree level, is initialized with the value one in each entry. A tree walk is begun, the only requirement being that each node must be visited before any node to the right on its own level. When a node is visited, the node is given the current value of the available position array indexed by the node's level; then the array entry is incremented by the width of the node and desired spacing between nodes. The physical limit is satisfied since each level is filled solidly from left to right.

Algorithm 1 is a program to supply a positioning. It is also a model for the more sophisticated positioning programs to come. We take this opportunity to discuss the programming considerations common to all the programs, using Algorithm 1 for an example.

All the programs are written in a variant of Pascal [2]. Extensions include dynamic arrays, some "syntactic sugar," and parameterized data types. Dynamic arrays make the programs independent of arbitrary limits on storage space. The syntactic extensions include explicit closers for each grouping statement (if, case, for, while), an elseif construct, and both break and # characters for identifiers. A parameterized data type may be seen in Fig. 3 where the declarations for tree nodes are given. The parameter of a data type is instantiated with a value each time a new object of the type is created, i.e., at block entry or during invocation of the Pascal system procedure new. Once instantiated, the parameter may be referenced like a field, may be used in the declarations of other fields (e.g., array **son** and flag **status**), but may not have its value modified. Declaration of an array with an empty index set is not an error; reference to an element of such an array is. These programs always check for the existence of the index set before access if such an error is possible. (The check may be done by a for loop with empty range—Fortran programmers beware.) Readers using these programs will find that the application will eliminate the dynamic arrays and parameterized types by supplying specific application values for array bounds and parameters; however, applications with dynamic data structures will need these dynamic storage application features in some form.

```
type node(parameter #_of_sons : integer);
   record
      data : ;         (* Whatever the user wants. *)
      father : branch;
      son : array [1..#_of_sons] of branch;
      height : integer;
      x, y : integer;
      status : 0..#_of_sons+1;
   end;  (* node *)

type branch : pointer to node;
```

Fig. 3. Data types used in positioning programs.

Finally, we shall not assume that our language is recursive. Instead, all the programs will iterate over the tree structure. There are a number of ways to use a tree to save the history of a routine walking the tree.

- Build a parallel stack of nodes whose processing was interrupted by processing the current node.
- Place a back trace pointer in each node to unwind the tree walk.
- Maintain a status marker in each node and a pointer to the node currently in process.

Readers probably know other means to the same end. We prefer the method of status markers. Although iterative versions of the programs may be slightly more obscure than recursive versions, they are no less efficient. Further, they may be translated directly into Fortran, assembly language, or other languages in which recursion is difficult or impossible. The basic structure of such iterations is described by Knuth [4], Bird [1], and Soule [5].

II. Binary Tree Drawings

As Fig. 2 illustrates, Algorithm 1 places fathers left of, right of, or centered over their sons. If a tree has no labeling on its branches, such a positioning is fine; graph theory books are full of trees drawn helter-skelter. But trees used in programs commonly are labeled, and perhaps most common are the

Input: A branch root pointing to a well-formed binary tree and an integer **max_height** giving the height of the tree. We assume that the **height** field of every node is set correctly,.

Output: The x and y fields of each node are set so that each node is in its in-order position.

Method: A variable **next_number** keeps track of the next number in the in-order sequence. At each node, the left subtree is numbered, the node itself is numbered, and then the right subtree is numbered. As in the first two algorithms, **status** fields and variable **current** record the progress of the numbering. In particular, **status** is set to **first_visit** before the first visit to the node, to**left_visit** while the left son is numbered, and to **right_visit** while the right son is numbered. The same technique is used in the later programs.

```
input  root : branch;
       max_height : integer;

var    current : branch;
       next_number : integer;

begin
   next_number := 1;
   root↑.status := first_visit;
   current := root;
   while current ≠ nil do
      case current↑.status of
         first_visit : begin
            current↑.status := left_visit;
            if current↑.left_son ≠ nil then
               current := current↑.left_son;
               current↑.status := first_visit;
            fi;
         end;
         left_visit : begin
            current↑.x := next_number;
            next_number := next_number + 1;
            current↑.y := 2*current↑.height + 1;
            current↑.status := right_visit;
            if current↑.right_son ≠ nil then
               current := current↑.right_son;
               current↑.status := first_visit;
            fi;
         end;
         right_visit : current := current↑.father;
      esac;
   end while;
end;  (* of Algorithm 2 *)
```

Algorithm 2. Position binary tree nodes by in-order (Knuth).

binary trees. In a binary tree, each branch is labeled *left* or *right*, and no node may have more that one *left* and one *right* son. In drawings, a label may often be inferred from the position of a son with respect to its father. This suggests the following.

Aesthetic 2: In a binary tree, each left son should be positioned left of its father and each right son right of its father.

Algorithm 2, due to Knuth [3], satisfies Aesthetic 2 by assigning to each node an x-coordinate proportional to the node's index in an in-order numbering of the tree. Since the in-order index of any node is always greater than that of its left son and less than that of its right son, each node must be correctly positioned with respect to its sons. By induction, every node is thus correctly positioned. But we shall see, Algorithm 2 does not satisfy the physical limit.

Algorithms 2 and 3 both manipulate binary trees, so the data structure for a **node** must be modified, as seen in Fig. 4. Fields **left_son** and **right_son** will have value nil if a node has no left son or right son, respectively.

III. Drawings Satisfying the Physical Limit

Algorithm 2 constructs drawings which satisfy Aesthetic 1, but which may be far too wide. Once a node occupies a column on the paper, no other node may occupy the same column; the drawing width is always equal to the number of nodes in the tree. In some cases, this width may be very nearly the best achievable; in others, considerable space may be wasted. But Fig. 5 illustrates what can happen to a sparse

Input: A branch root pointing to a well-formed binary tree and an integer **max_height** giving the maximum height of the tree. We assume that each node has its height assigned.

Output: A tree positioned to satisfy Aesthetics 1 and 2 and usually satisfying the Physical Limit.

Method: In a first post-order walk, every node of the tree is assigned a preliminary x-coordinate (held in field x). In addition, internal nodes are given **modifier**'s which will later be used to move their sons right. During a second pre-order walk, each node is given a final x-coordinate by summing its preliminary x-coordinate and the **modifier**'s of all the node's ancestors. The y-coordinate depends, as before, on the height of the node.

```
input  root : branch;
       max_height : integer;

var    modifier : array [0..max_height] of integer;
       next_pos : array [0..max_height] of integer;
       i : integer;
       place : integer;
       h : integer;
       is_leaf : Boolean;
       modifier_sum : integer;

begin
   for i := 0 to max_height do
      modifier[i] := 0; next_pos[i] := 1;
   end for;
   current := root;
   current↑.status := first_visit;
   while current ≠ nil do
      case current↑.status of
         first_visit : begin
            current↑.status := left_visit;
            if current↑.left_son ≠ nil then
               current := current↑.left_son;
               current↑.status := first_visit;
            fi;
         end;
         left_visit : begin
            current↑.status := right_visit;
            if current↑.right_son ≠ nil then
               current := current↑.right_son;
               current↑.status := first_visit;
            fi;
         end;
         right_visit : begin
            h := current↑.height;
            is_leaf := (current↑.left_son = nil)
                    & (current↑.right_son = nil);
            if is_leaf
               then place := next_pos[h];
            elseif current↑.left_son = nil
               then place := current↑.right_son↑.x - 1;
            elseif current↑.right_son = nil
               then place := current↑.left_son↑.x + 1;
            else
               place := (current↑.left_son↑.x+current↑.right_son↑.x) ÷ 2;
            fi;
            modifier[h] := max(modifier[h], next_pos[h]-place);
            if is_leaf
               then current↑.x := place;
               else current↑.x := place + modifier[h];
            fi;
            next_pos[h] := current↑.x + 2;
            current↑.modifier := modifier[h];
            current := current↑.father;
         end;
      esac;
   end while;


   current := root;
   current↑.status := first_visit;
   modifier_sum := 0;
   while current ≠ nil do
      case current↑.status of
         first_visit : begin
            current↑.x := current↑.x + modifier_sum;
            modifier.sum := modifier.sum + current↑.modifier;
            current↑.y := 2*current↑.height + 1;
            current↑.status := left_visit;
            if current↑.left_son ≠ nil then
               current := current↑.left_son;
               current↑.status := first_visit;
            fi;
         end;
         left_visit : begin
            current↑.status := right_visit;
            if current↑.right_son ≠ nil then
               current := current↑.right_son;
               current↑.status := first_visit;
            fi;
         end;
         right_visit: begin
            modifier_sum := modifier_sum - current↑.modifier;
            current := current↑.father;
         end;
      esac;
   end while;
end;  (* of Algorithm 3 *)
```

Algorithm 3. A tidy tree drawer.

```
type node record
   data : ;  (* Whatever the user wants. *)
   father : branch;
   left_son, right_son : branch;
   height : integer;
   x, y : integer;
   status : (first_visit, left_visit, right_visit);
   modifier : integer;  (* Used by Algorithm 3. *)
end node;
```

Fig. 4. The data type **node** modified for binary trees.

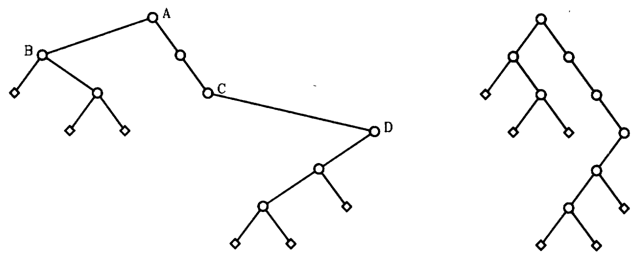


Fig. 5. Two drawings of the same tree.

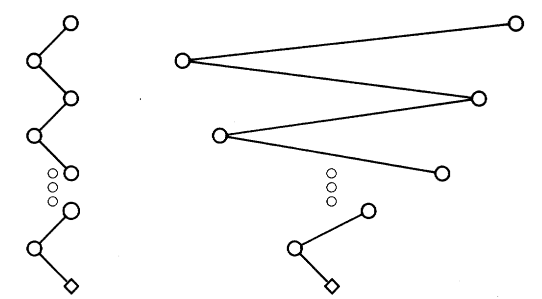
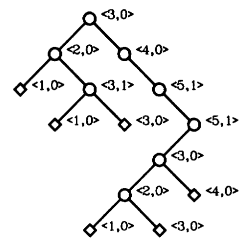


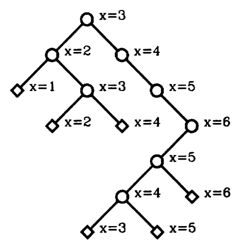
Fig. 6. A worst case example for Knuth's tree drawing algorithm.



x and modifier fields are shown.

h	next_pos	modifier
0	~~1~~ ~~3~~ 5	0
1	~~1~~ ~~4~~ 6	0
2	~~1~~ ~~3~~ ~~5~~ 7	~~0~~ 1
3	~~1~~ ~~5~~ 7	~~0~~ 1
4	~~1~~ ~~3~~ ~~5~~ 3	0
5	~~1~~ 4	0
6	~~1~~ ~~3~~ 5	0

next_pos and modifier values.



Final x-coordinates are given.

Fig. 7. Example output from Algorithm 3.

tree. On the left is a drawing produced by Algorithm 2. The long branches **A-B** and **C-D** are required to leave room for the subtrees beneath. On the right the same tree is drawn in a width of six (versus 14) by folding subtrees beneath their ancestors where possible. Fig. 6 is a worst case example for Algorithm 2.

Algorithm 1 and 2 each satisfy one constraint completely while ignoring another; in each case, grotesque trees may result. Algorithm 3 merges the ideas of the two previous algorithms. As in Algorithm 1, array **next_pos** maintains the next available node position for each level in the tree. If a leaf at level **h** is under consideration, placement of the leaf at **next_pos[h]** is legal and satisfies the minimum width requirement. But an internal node placed willy-nilly at **next_pos[h]** may well violate Aesthetic 2. Rather, a provisional **place** for an internal node is the average of its sons' positions (with appropriate special cases when a son is missing). The actual position assigned must be the maximum of the provisional **place** and **next_pos[h]**, since sons may try to drag their father too far to the left and cause the father to collide with his relatives to the left.

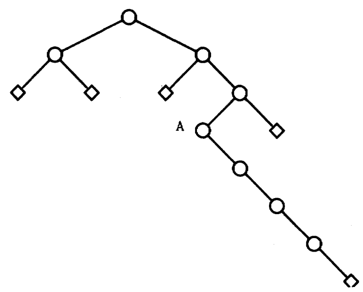
If the actual position of an interior node is right of its provisional **place**, the subtree rooted at the node must be moved bodily right to center the sons around the father. Rather than immediately readjust all the nodes in the subtree [a process that could make the algorithm run in time $\mathcal{O}(n^2)$], each node remembers the distance to the provisional **place** in a **modifier** field. In a second pass *down* the tree, **modifier's** are cumulated and applied to every node. During the first pass, which assigns positions as described, a **modifier** is kept for each level; the **modifier's** of the nodes across a row must not decrease or subtrees may overlap.

Fig. 7 shows our example tree as drawn by Algorithm 3. Values from the two passes of the algorithm are also displayed so that tracing the execution on this input tree will be easy. The algorithm cost is linear in the number of tree nodes since only two walks are necessary. The example tree is properly positioned under Aesthetics 1 and 2 and the physical limit.

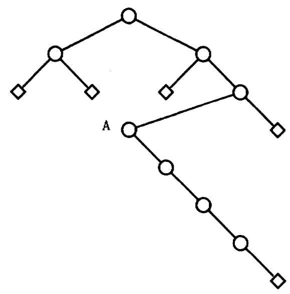
However, Algorithm 3 does not always meet the physical limit. Fig. 8 provides an example of a tree positioned badly by Algorithm 3. When the node marked A is pushed left, a narrower, if uglier, positioning of the same tree results. The violation arises because Algorithm 3 attempts to enforce the following strong version of Aesthetic 2.

Aesthetic 3: A parent should be centered over its children.

The tree of Fig. 8 can be expanded *ad libitum* to make the critical father arbitrarily biased towards his right son. These drawings suggest the following.



The tree drawn by Algorithm 3.



A Narrower Version.

Fig. 8. A tree badly positioned by Algorithm 3.

```
for i := 0 to max_height do next_pos[i] := 1; end for;
current := root;
current↑.status := first_visit;
modifier_sum := 0;
while current ≠ nil do
   case current↑.status of
      first_visit : begin
         modifier_sum := modifier_sum + current↑.modifier;
         current↑.status := left_visit;
         if current↑.left_son ≠ nil then
            current := current↑.left_son;
            current↑.status := first_visit;
         fi;
      end;
      left_visit : begin
         current↑.x = min(next_pos[current↑.height],
                      current↑.x+modifier_sum-current↑.modifier);
         if current↑.left_son ≠ nil
            then current↑.x = max(current↑.x,
                                  current↑.left_son↑.x+1); fi;
         if current↑.father ≠ nil
            then if current↑.father↑.status = right_visit
               then current↑.x := max(current↑.x,
                                      current↑.father↑.x+1); fi; fi;
         next_pos[current↑.height] := current↑.x + 2;
         current↑.y := 2*current↑.height + 1;
         current↑.status := right_visit;
         if current↑.right_son ≠ nil then
            current := current↑.right_son;
            current↑.status := first_visit;
         fi;
      end;
      right_visit : begin
         modifier_sum := modifier_sum - current↑.modifier;
         current := current↑.father;
      end;
   esac;
end while;
```

Fig. 9. A modification to Algorithm 3.

Theorem (Uglification): Minimum width drawings exist which violate Aesthetic 3 by arbitrary amounts.

Nonetheless, we can modify Algorithm 3 to produce minimum width trees. In the second pass, a post-order walk passed the **modifier_sum** down the tree. The direction of the walk was chosen only for convenience; it seemed clearer to apply the **modifier_sum** to a node the first time the node was encountered. However, a pre-order walk would have done as well. Further, a pre-order walk positions nodes down the left edge of the tree before right subtrees are seen. We take advantage of this ordering by maintaining for each tree level the actual next position available at the level. When a node is given its final position, that position is the minimum of the next available position at the node's level, its left son's position plus one, its father's position plus one (for a right son), and the position that would have been applied by Algorithm 3. No change to the node's **modifier** need be made since the children (right branch only) affected by the change in positioning will apply the modification to themselves. Fig. 9 provides coding to replace the second while loop and its initialization in Algorithm 3.

Algorithm 3 has been presented for binary trees. But by suitable policy choices, it may be modified to work for arbitrary trees. During the first while loop, the case statement must be modified to allow for the increased number of branches. Also, the simple average of the childrens' positions to find the father's position must be replaced with any function desired to "center" fathers over sons. The centering function could use information about the children's values and labels, as well as positional information, to determine a father's position. In the second pass, the case statement must be modified. Otherwise, we need only ensure that each left child is positioned before its father, who is in turn positioned before his right child. Although implementation of such policies may take a considerable decision-making code, especially for trees in which positions relative to *absent* children are important, the basic idea is simple.

Since we first developed these algorithms, Knuth has drawn our attention to the dissertation of Sweet. In his work, Sweet had need to draw many trees and developed a primitive program to do so. It embodies the same basic ideas as our algorithms, but Sweet did not bother to develop them beyond his specific problem. A brief discussion of his tree drawing technique can be found in an Appendix to his dissertation [6].

IV. Conclusions

We have presented two algorithms for the tidy positioning of trees. Within some aesthetic constraints, one algorithm uses the minimum possible paper width and the other uses somewhat more paper to draw a prettier tree. Both algorithms run in linear time, taking several walks over the tree structure. No recursion is necessary, which makes the algorithms attractive for use in nonrecursive programming languages. Both are easy to code in common languages, although the presentation here is somewhat long-winded so that the algorithms can be seen in their full generality.

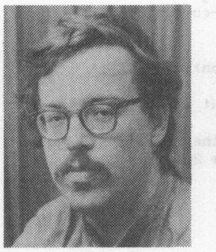
Trees are very common data structures, as has been mentioned. However, other planar and nonplanar graphs also appear as computer output. We are currently studying methods for the tidy display of other graph structures, a subject not covered in the literature.

References

[1] R. S. Bird, "Notes on recursion elimination," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 434–439, June 1977.

[2] K. Jensen and N. Wirth, *PASCAL User Manual and Report: Lecture Notes in Computer Science*, vol. 18. Berlin, Germany: Springer-Verlag, 1974.

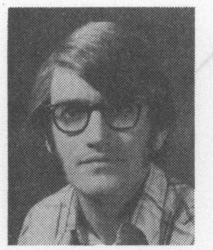
[3] D. E. Knuth, "Optimum binary search trees," *Acta Informatica*, vol. 1, pp. 14-25, 1971.
[4] ——, *The Art of Computer Programming/Fundamental Algorithms.* Reading, MA: Addison-Wesley, 1968.
[5] S. Soule, "A note on the nonrecursive traversal of binary trees," *Comput. J.*, vol. 20, no. 4, pp. 350-352, 1977.
[6] R. Sweet, "Empirical estimates of program entropy," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1977.



Charles Wetherell received the A.B. degree in applied mathematics from Harvard University, Cambridge, MA, in 1967 and the Ph.D. degree in computer science from Cornell University, Ithaca, NY, in 1975.

He currently holds a joint appointment as an Assistant Professor at the University of California, Davis, and as a Research Scientist at Lawrence Livermore Laboratory, Livermore, CA. His research interests include programming language design and implementation. Lately, he has been concerned with design and standardization of advanced Fortran dialects for use within the Department of Energy. His new textbook, *Etudes for Programmers*, was published last winter by Prentice-Hall.

Dr. Wetherell is a member of the Association for Computing Machinery and the Computer Society.



Alfred Shannon received the B.S. degree in applied mathematics from California State University at Hayward in 1974 and the M.S. degree in computing science from the University of California at Davis in 1976.

He is presently continuing work at Davis towards a Ph.D. degree. He is a Computer Scientist at Lawrence Livermore Laboratory, Livermore, CA, and works in the compiler group. His current responsibilities include code generator production for a new compiler on the Cray-1 and development of tools for automatic syntax analysis. Doctoral research is in the area of parser generation and automation of complier production.

Mr. Shannon is a member of the Association for Computing Machinery.

On Path Cover Problems in Digraphs and Applications to Program Testing

S. C. NTAFOS AND S. LOUIS HAKIMI, FELLOW, IEEE

Abstract—In this paper various path cover problems, arising in program testing, are discussed. Dilworth's theorem for acyclic digraphs is generalized. Two methods for finding a minimum set of paths (minimum path cover) that covers the vertices (or the edges) of a digraph are given. To model interactions among code segments, the notions of required pairs and required paths are introduced. It is shown that finding a minimum path cover for a set of required pairs is NP-hard. An efficient algorithm is given for finding a minimum path cover for a set of required paths. Other constrained path problems are considered and their complexities are discussed.

Index Terms—Algorithmic complexity, Dilworth number, minimum path cover, must pairs, must paths, NP-hard, program testing, required pairs, required paths.

Manuscript received April 26, 1978; revised January 29, 1979. This work was supported by the U.S. Air Force Office of Scientific Research, Systems Command, under Grant AFOSR-76-3017.

S. C. Ntafos was with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. He is now with the Department of Mathematical Sciences, University of Texas at Dallas, Richardson, TX 75080.

S. L. Hakimi is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

I. Introduction

PROGRAM testing is widely used in software validation [1]. It consists of selecting a set of test paths that covers certain features of the program and finding appropriate test data that exercise these paths. One may choose, for example, a test set in which every program statement is executed at least once, or a more extensive test set that would exercise all exits from all branch statements. If we represent a program as a digraph (program graph), these strategies correspond to the problems of finding sets of source to sink paths, to be called *s-t* paths, that cover the vertices or the edges of the digraph. In view of the high cost of program testing [2], we are naturally interested in finding path covers with the minimum number of paths. Krause *et al.* [3] suggested a method where the path covering the maximum number of the remaining untested elements is chosen as the next test path. Miller *et al.* [4] proposed a method where a program is decomposed into decision-to-decision paths which are then combined to form optimal test path covers. Both of these methods are used in automated validation tools together with test data generation techniques and neither is guaranteed to produce a minimum path cover

0098-5589/79/0900-0520$00.75 © 1979 IEEE