This material may be protected by copyright law (Title 17 U.S. Code)

Borrower: UAA

Lending String: *IQU,UAF,OUP,CSL,WUG

Patron: Ray, Elena

Journal Title: IEEE transactions on software

engineering.

Volume: 5 Issue: 5

Month/Year: 09/1979Pages: 514-520

Article Author: Wetherall, C. and Shannon, A.

Article Title: Tidy drawings of trees

Imprint: [New York, Institute of Electrical and E

ILL Number: 3965009

Call #: QA76.6 I17

Location: sper

ARIEL Charge

Maxcost: RECIP or \$20.00IFM

Shipping Address:

UNIVERSITY OF ALASKA - ANCHORAGE

INTERLIBRARY LOAN 3211 PROVIDENCE DRIVE ANCHORAGE, ALASKA 99508

Fax: 907-786-1841 Ariel: 137.229.112.5

5

- [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.

 —, "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.
- [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.
- "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.
- "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.

 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 compute science from Purdue University, Lafayette, IN, in 1972, 1974, and 1978, respectively.

She is currently an Assistant Professor Computer Science at Michigan Technologic University, Houghton, MI. Her research terests include software science, software reliability, and programming methodologic Dr. Ottenstein is a member of the Associ

tion 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.

NOMPUTER programmers know well that trees are tremely common data structures. Trees model many world problems and a host of efficient and useful tree-base gorithms exist. Equally important, a good drawing of a tag often a powerful intuitive guide to a modeled problem; in some real problems consist of little more than finding drawing a particular tree. But programmers who use tree dom provide pretty graphic output. Users commonly to listings of trees rather than demanding pictures. We att the lack of pictures to a dearth of published technique tree drawing. In this paper, we present some algorithm heuristics for drawing tidy trees.

What, exactly, are the difficulties of drawing a tree? of course, each node of the tree must be assigned a posit the drawing surface. We assume that the drawing surface ways a flat sheet (e.g., of paper) and we will make use expedients such as twisting a rubber surface. Positions

gular coo tent. On be drawn garies of l drawing a the tree no But the

WETHERE

assigned u

surface. moment, is actually be pieced ruled out are bound plotter) or bounded drawings, 1 drawing be that nodes tion. Node bels may v so that lab Some device other than lems severe trary points Of the ph with the eff lems caused **leave** proble dow resoluti **have** librarie culties; any we could no . Aesthetics rawings of hat a drawi **he** propertie dy drawing Trees are Trees im Ber to the iIn a binar the left of wright of th aturally e cical limit **itio**nings v idevices; d serve. 🕏 or provi **he**uristic

wer costs

problem problem

Me which

te consid cause w 83 (4124),

BER 1979

Rome Air

software,"

empirical 1975, pp.

Rome Air 76. software," 74. ing." ACM

heboygan, ceived the computer ayette, IN,

ofessor of innological search insoftware todologies. e Associaty.

are exany realbased alf a tree is ; indeed, ding and trees seltolerate attribute ques for hms and

? First, ition on ace is afse of no swill 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 nuled 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 doser 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 hysical limits of drawing devices. Were it not so, very simple ositionings would suffice for drawings even on severely limited devices; vertical columns or rectangular arrays of nodes rould serve. Aesthetics may also require exorbitant computations or provide only ill-defined notions of tidiness. In such ases, heuristics rather than algorithms will be necessary, both olower costs and to approximate tidiness.

The problem of drawing a tidy tree reduces to finding a posioning which reconciles aesthetics and physical limits. The parate consideration of aesthetics and physical limits is fruital because we shall be able to formalize each in a simple

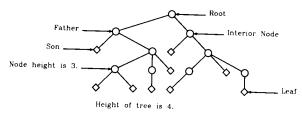


Fig. 1. An example tree.

way. In the following section, we will be more precise in our discussion.

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 <u>root</u>, 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. 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

¹Computer 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.

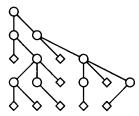


Fig. 2. The example tree repositioned by Algorithm 1.

```
f: A branch root pointing to the root of a well-formed tree and an
integer max_height giving the height of the tree.
                                             each node are set so that the tree is
                           fields of
                  and y
positioned with as narrow a width as possible.
height of each node has been correctly set.
                                                                       We assume that the
           counter holding the
```

od: 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 feature of the root is not programs assume that the father of the root is nil.

```
root : branch;
                                                max_height : integer;
                                               next\_x : array [0..max\_height] of integer;
 var
                                               current : branch;
                                                  i : integer;
begin
                     for i := 0 to max height do next x[i] := 1; end for;
                   root'.status := 0;
current := root;
                root terront := root;

while current * nil do

if current*.status = 0 then

current*.y := 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

curr
                                       for i := 1 to current'.#eigntj := next x[current
for i := 1 to current'.#of sons do
current'.son[i]*.status := 0; end for;
current'.status := 1;
elseif 1 s current'.status
                                                           & current'.status & current'.#.of.sons then
current'.status := current'.status + 1;
current := current'.son[current'.status-1];
se (* current'.status > current'.#.of.sons *)
                                         else
                                                            current := current . father;
                     end while;
1; (* 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]. Ex-

```
type node(parameter #_of_sons : integer);
   <u>reco</u>rd
                       (* 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.

tensions include dynamic arrays, some "syntactic sugar," and parameterized data types. Dynamic arrays make the programm independent of arbitrary limits on storage space. The syntaction tic 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 value each time a new object of the type is created, i.e., block entry or during invocation of the Pascal system proce dure new. Once instantiated, the parameter may be refer enced like a field, may be used in the declarations of other fields (e.g., array son and flag status), but may not have it value modified. Declaration of an array with an empty inde set is not an error; reference to an element of such an array is These programs always check for the existence of the index se before access if such an error is possible. (The check may be done by a for loop with empty range—Fortran programmer beware.) Readers using these programs will find that the appli cation will eliminate the dynamic arrays and parameteriza types by supplying specific application values for array bound and parameters; however, applications with dynamic date structures will need these dynamic storage application feature in some form.

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 a routine walking the tree.

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

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

II. BINARY TREE DRAWINGS

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

field Output: its i Method: in-or node in the the pi left : VAL <u>begin</u> root + . st while cu

max

WETHER

Algoria inary tre **light**, and ton. In di tion of a following.

end while (* of

rig

Aesthetic ioned left

Algorithm **ig**ning to e ndex in an ndex of a nd less tha ositioned thus cor pes not sat Algorithm tructure fo telds left

III. Di Algorithm at which n inn on the nn; the d des in th parly the b

wasted.

o left son

er tr

Cursiv

ístory **a**

ras interi

the tree

er to this

re ve

ley may

age, or

ossible.

Knuth

of, right.

g on its 🖟

oks are

rograms

are the

ructure,

```
f: A branch root pointing to a well-formed binary tree and an <u>integer</u> max_height giving the height of the tree. We assume that the height
     field of every node is set correctly,.
            The \mathbf{x} and \mathbf{y} fields of each node are set so that each node is in
  its in-order position.
   ad: 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
   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, toleft_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.
  t root : branch;
      max_height : integer;
      current : branch;
      next number : integer:
ext_number := 1;
root:.status := first visit;
urrent := root;
hile current = n:1 do
case current status of
         first_visit : begin
current*.status := left visit;
              current : status := lett visit,
if current : left son # nil then
current := current : left son:
                   current'.status := first visit;
              fi:
         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;
        end;
        right_visit : current := current+.father;
md <u>while;</u>
(* of Algorithm 2 *)
  Algorithm 2. Position binary tree nodes by in-order (Knuth).
```

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

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

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

Algorithms 2 and 3 both manipulate binary trees, so the data fucture for a **node** must be modified, as seen in Fig. 4. **telds left_son** and **right_son** will have value <u>nil</u> if a node has o left son or right son, respectively.

III. DRAWINGS SATISFYING THE PHYSICAL LIMIT

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

```
f: 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.
                A tree positioned to satisfy Aesthetics 1 and 2 and usually
        satisfying the Physical Limit.
       od: 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.
        before, on the height of the node.
input root : branch;
           max_height : integer;
          modilier: <u>array</u> [0..max_height] of <u>integer;</u> next_pos: a<u>rray</u> [0..max_height] of integer;
                 integer;
           place : integer;
          h : <u>integer;</u>
is leaf : <u>Boole</u>an;
           modifier sum : integer;
   for 1 := 0 to max height do
          modifier[i] := 0; mext[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;
             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 : <u>begin</u>
h := current*.height;
is_leaf := (current*.left_son = <u>nil</u>)
                  & (current1.right_son = <u>nil</u>);
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;
                       place := (current*.left_son*.x+current*.right_son*.x) ÷ 2;
                 modifier[h] := max(modifier[h], next_pos[h]-place);
                       then current .x := place;
                       else current .x := place + modifier[h];
                 next_pos[h] := current + x + 2
                 current'.modifier := modifier[h];
current := current'.father;
            end:
  end while;
  current := root;
  current . status := first visit:
 current .x := current .x + modifier_sum;
                current : = current : 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;
                <u>f</u>i;
           left_visit : begin
    current*.status := right_visit;
                if current'.right son * nil then
current := current'.right son
                     current'.status := first_visit;
           end:
          right_visit: <a href="mailto:begin">begin</a>
modifier_sum := modifier_sum - current'.modifier;
          end;
esac;
end while;
d: (* 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.

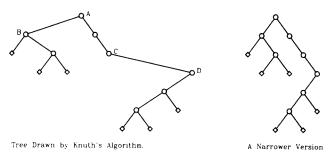


Fig. 5. Two drawings of the same tree.

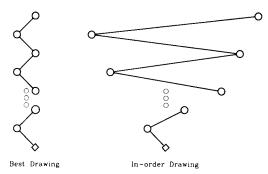
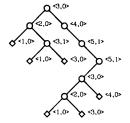


Fig. 6. A worst case example for Knuth's tree drawing algorithm,

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.

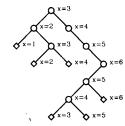
If the actual position of an interior node is right of its provisional place, the subtree rooted at the node must be moved



x and modifier fields are shown.

h	${\tt next_pos}$	modifier
O	1 3 5	0
1	\ \ \ 6	0
2	N 3 5 7	Ò , 1
3	1 5 7	Ò , 1
4	1 3 5 3	0
5	\ 4	0
6	1 , 3 , 5	0

next_pos and modifier values.



Final x-coordinates are given.

Fig. 7. Example output from Algorithm 3.

bodily right to center the sons around the father. Rather the immediately readjust all the nodes in the subtree [a proof that could make the algorithm run in time $O(n^2)$], each not remembers the distance to the provisional place in a modificial. 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 of 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.

nethe difier. ed. H ather, the tre this o at pos pal pos ble pos ne, its f ition th hange to right bra pply the o replac iithm 3. Algorit uitable 1 trary tre nust be ranches.

o find t

tion desir

ion coul

jels, as w

osition.

ied. Oth

itioned b

ght chil

WETHE

The tree drawn by Algorithm 3.

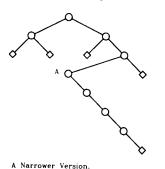


Fig. 8. A tree badly positioned by Algorithm 3.

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

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

Algorithm 3 has been presented for binary trees. But by hitable policy choices, it may be modified to work for arbitary trees. During the first while loop, the case statement ust be modified to allow for the increased number of ranches. Also, the simple average of the childrens' positions of 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 latels, as well as positional information, to determine a father's osition. 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 ight child. Although implementation of such policies may

```
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
   if current t. left_son + nil then
              current := current + . left_son
              current + . status := first_visit;
           fi;
       end;
       left_visit : <u>begin</u>
          current + .x = min(next_pos[current + .height],
                             current + .x + modifier_sum-current + .modifier);
          \underline{if} current+.left_son \neq \underline{nil}
              then current \cdot x = \max(\text{current} \cdot x)
                                       current+.left_son+.x+1); fi;
          if current+.father # nil
              then if current . father t. status = right_visit
                 then current \cdot x := max(current \cdot 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;
      <u>fi</u>;
end;
ri
      right_visit : begin
          modifier_sum := modifier_sum - current*.modifier;
          current := current + . father ;
      end;
end while;
```

Fig. 9. A modification to Algorithm 3.

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

- 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.

er than process h node cumu-, which h level; tease or

BER 1979"

thm 3. splayed e easy. es since roperly hit. hysical idly by a nar-

iren. ke the These

e viola-

he fol-

[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

ROGRAM testing is widely used in software validation [1]. It consists of selecting a set of test paths that cover certain features of the program and finding appropriate ter data that exercise these paths. One may choose, for example a test set in which every program statement is executed least once, or a more extensive test set that would exercise a exits from all branch statements. If we represent a program a 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 nati rally interested in finding path covers with the minimum nur ber of paths. Krause et al. [3] suggested a method where the path covering the maximum number of the remaining unteste elements is chosen as the next test path. Miller et al. [4] pro posed a method where a program is decomposed into decision to-decision paths which are then combined to form optim test path covers. Both of these methods are used in automate validation tools together with test data generation technique and neither is guaranteed to produce a minimum path cover

covers for t cal as well lems, arisin reduce to the We will g for the vert

for the pro

for the vert minimum p flow proble matching pr techniques relates the pendence m

To certify in the prog ecution sequ tive testing set of the manageable ing program interesting of code segmen for finding a of vertices V interaction a among code pairs" and ' vertices (i.e. at least one execution su Techniques f erate sets of that the prol of required p s given for the set of requi ng an approx leveloped by nd then solvi A limitation at some gra equences, i.e krause et al. model this ng an s-t pat an NP-comp ad out-degre he notion of ubpaths of a etermining th n s-t path th raphs is an o **je**its vertex *v_i* ill show that ertices of the n is NP-com

xists no efficience of the inpu