The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 289 August. 1986

READABLE LAYOUT OF UNBALANCED N-ARY TREES

David M. Solo

ABSTRACT

The automatic layout of unbalanced n-ary tree structures ia a problem of subjectively meshing two independent goals: clarity and space efficiency. This paper presents a minimal set of subjective aesthetics which insures highly readable structures, without overly restricted flexibility in the layout of the tree. This flexibility underlies the algorithm's ability to produce readable trees with greater uniformity of node density throughout the display than achieved by previous algorithms, an especially useful characteristic where nodes are labelled with text.

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be papers to which reference may be made in the literature.

READABLE LAYOUT OF UNBALANCED N-ARY TREES

I. INTRODUCTION

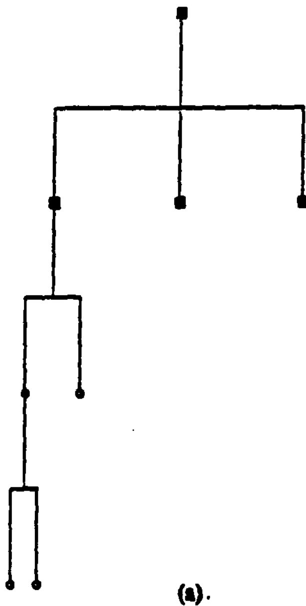
The problem of efficiently laying out tree structures in a readable fashion is really a subjective one, involving the careful meshing of two mutually antagonistic goals. The first goal, creating efficient, compact layouts, drives toward producing tightly intermeshed trees with little wasted display space. The second goal, creating readable, clear structures, is really a question of human engineering. Unchecked, this goal produces symmetric, readable trees, which unfortunately are very wide and space inefficient. Many algorithms do not deal with this subtle balance.

The common application of tree data structures makes utilities which can layout trees highly useful. The layout of fairly balanced, fixed arity trees is not difficult, but highly unbalanced, varying arity trees present a much subtler problem. For simplicity, I am arbitrarily assuming that the tree is being drawn with the root at the top of the display and inferiors branching below it. It seems to be universally acceptable to place nodes at the same depth in a tree aligned horizontally in the display. Therefore, vertical spacing of a tree is straightforward; simply divide the display height by the number of levels in the tree (plus one for spacing) and use this value as the vertical step between a level and the previous one. The question of horizontal positions of nodes is really the crux of the problem. Many tree drawing algorithms exist, but most fail to produce both readable, and space efficient layouts. Ideally, none of the limited display space would be wasted. Also, an even density of nodes throughout the display would provide maximum room to label nodes, assuming they are all the same size.

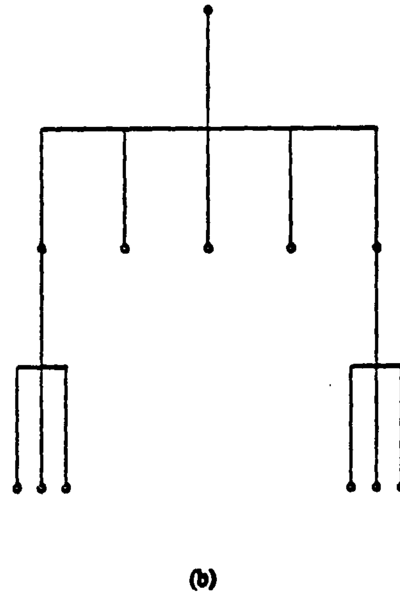
II. MOTIVATIONS

The most straightforward algorithm which typically first occurs to you does not maximize use of available space. This commonly used subdivision plan simply divides the allotted display width by the number of nodes at that level down in the tree, creating that many vertical columns in the display. Each node is centered in its column and all of its inferiors must lie within this vertical column. Therefore, the inferiors of one of these nodes will divide among themselves only this space allotted to their superior, and so on. The flaw in this appealingly obvious algorithm is that precious display space is wasted by failing to alot space according

to need. For example, suppose three nodes exist at a certain level, but only one has a subtree. Of the column given to these three nodes' superior, only one third will be used to display what may be a complex subtree, whereas two thirds will simply contain the other two nodes, each with no inferiors as in figure (a) of the first pair of layouts. The two nodes without inferiors have no need for the space below them. This space could be used by the subtree of the other node as in figure (a) of the second pair of layouts. Comparing the layouts in figures 1 and 2 also points out that by using an algorithm that alots display space with regard to each node's inferiors, greater uniformity of node density is achieved, providing much more room to label nodes with text.

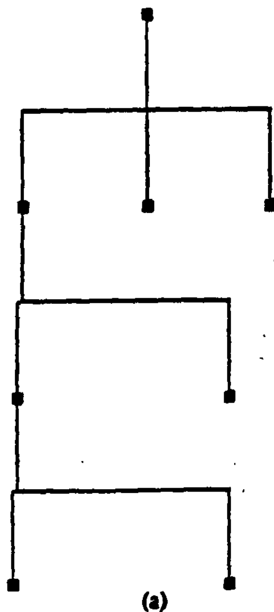


(a).

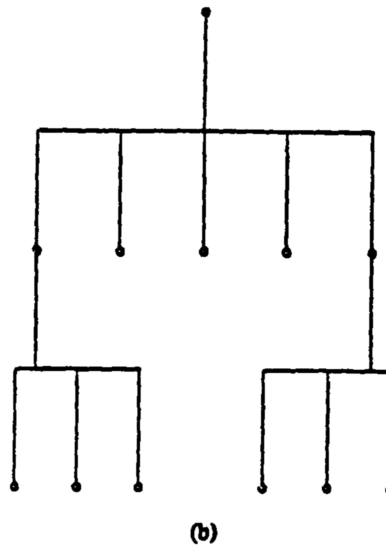


(b)

1. A few examples of the inefficient use of space associated with this simple algorithm.



(a)



(b)

2. The same structures efficiently displayed using the algorithm here presented.

The simple algorithm used in figure 1 strives for symmetry and readability without concern for economical or uniform density layouts; therefore, space inefficient trees with little room for text labels are often produced. Other, more complex, algorithms have also been proposed for calculating the horizontal positioning of nodes. Many such algorithms are really intended for binary trees, but are claimed to be useful for n-ary trees simply by evenly inserting the extra nodes between the leftmost and rightmost inferiors of each node. These binary tree algorithms actually find only the positions of the two outside nodes. For a very unbalanced tree, this will often lead to undesired crowding below nodes with many inferiors. The algorithm presented here weights the number of inferiors that each node owns in laying out unbalanced trees. This inferior weighting approach gives more attention to the goal of efficient, economical layouts. In unbalanced trees, the maximum use of an intrinsically limited display space is a must, if the complex subtrees of some nodes are not going to be squeezed into an unduly small fraction of the display area.

III. AESTHETICS RULES

As the successful combination of these two goals, human engineering, and space efficiency, is a subjective matter, deciding where compromises can be made in each of the two ideals and arriving at set of actually aesthetic rules, underlies the implementation of the layout algorithm presented here. My approach has been to settle upon a minimal set of constraints for the human engineering, and then allow the space efficiency goal to maximize economy in the layout. As a result, logically readable trees can be composed which are very space efficient.

Composing the set of minimal human engineering aesthetics involves deciding which aesthetics are needed to insure clarity in the display, and which actually are too constraining, destroying the flexibility in possible layout structures needed by the efficiency goal. In a paper by Reingold and Tilford [1], which builds upon a paper by Wetherell and Shannon [2], four aesthetics for drawing trees are detailed.

- The Vertical Spacing Rule simply stipulates that nodes of the same level in a tree appear at the same vertical depth in the display.
- The Multiple Inferior Rule is produced by applying the aesthetic concerning binary trees presented in their paper to n-ary trees. It states that the leftmost inferior must be placed to the left below its superior and the rightmost inferior to its superior's right.

* The Centered Superior Rule states that a superior must be centered over its inferiors.

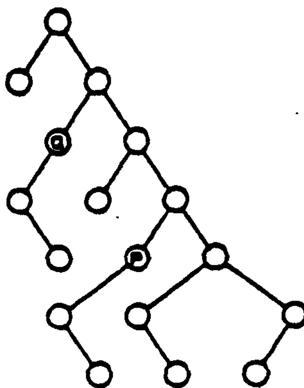
* The Relocatable Subtree Rule stipulates that isomorphic subtrees be drawn identically regardless of where they lie within a tree.

Clarity demands the use of the Vertical Spacing and Multiple Inferior aesthetics. Their use greatly enhances the readability of a structure without unduly limiting the flexibility of a tree's shape. In contrast, the Centered Superior and Relocatable Subtree aesthetics seem logical, but give too much weight to the human engineering side of the problem, with its drive for symmetry. Overall, they compromise too much flexibility and are unproductive in creating both readable and space efficient trees.

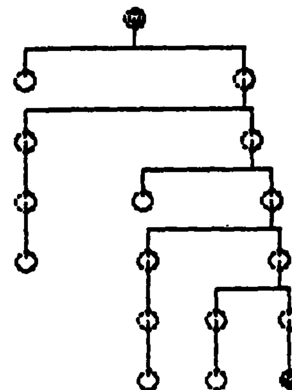
Eliminating this last pair of aesthetics, and adding the Single Inferior Rule (stated below) establishes the only needed constraints on readability. With these rules, the algorithm can take even highly unbalanced trees, and produce compact, and highly readable tree layouts.

* The Single Inferior Rule states that a node with only one inferior must have that inferior placed directly below itself, not off to either side.

The same natural aesthetic leading to the Multiple Inferior Rule implies the need for the Single Inferior Rule. Layouts are clearer following this rule. It is reasonable to allow a branching subtree of a node to be displaced unevenly below a distant superior a few levels up in a tree, but it is confusing to allow a repeating subtree of single inferiors to snake down the display. A straight column of nodes provides for much more logical and readable structures (compare diagrams). Though this algorithm is tailored more to the non-uniformities encountered in varying arity trees, if the distinction of a single inferior being either a left or a right inferior is needed, as in some binary tree applications, a text marker could be used when implementing the Single Inferior constraint.



1. A tree displayed without the Single Inferior aesthetic.

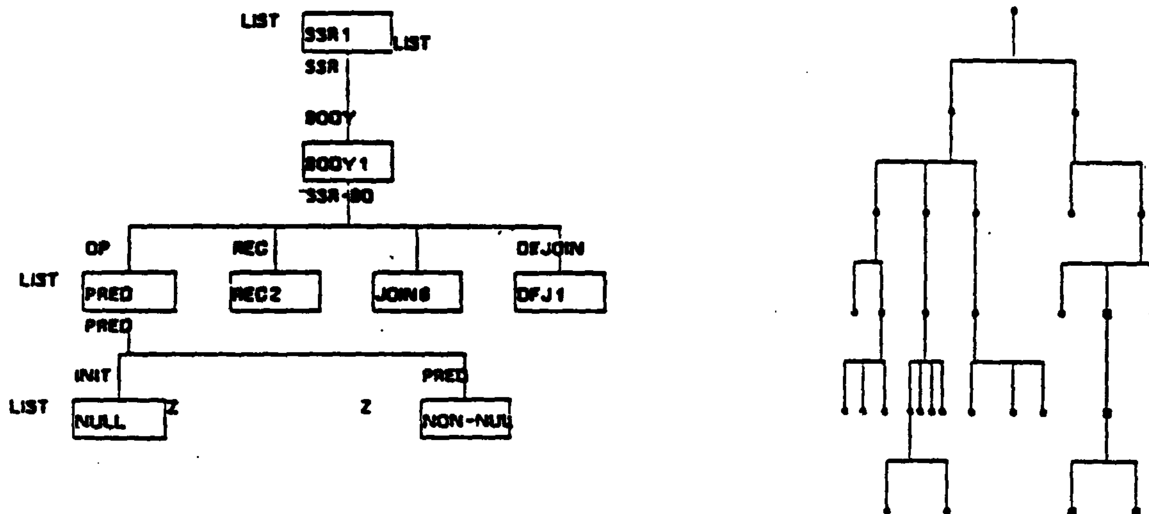


2. The same tree displayed using the Single Inferior aesthetic.

Eliminating the Centered Superior and Relocatable Subtree Rules increases flexibility of a given layout, allowing for more economical structures. A space efficient tree uses the available horizontal space effectively by spreading out nodes at each level as evenly as possible within the prescribed constraints. This method produces trees with greater uniformity of node density throughout the display. Areas of highly packed nodes and seemingly isolated nodes are minimized. These are superior to layouts made using the added pair of constraints when viewed in light of both readability and layout economy. Specifically, allowing a superior to be anywhere above and within the right and left edges of its immediate inferiors, in conjunction with allowing subtrees to be displayed differently depending upon global considerations, underlies the algorithms ability to successfully layout any arbitrary tree structure.

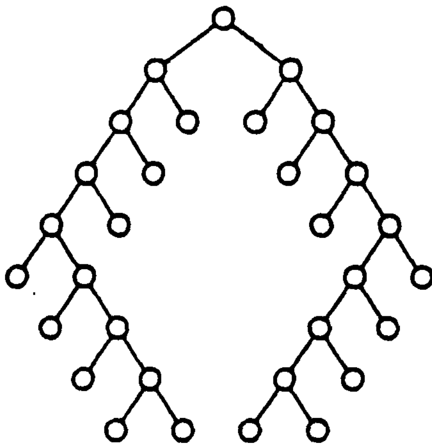
IV. THE ALGORITHM

The inferior weighting algorithm uses a few rules to produce clear, economical drawings of an unbalanced tree structure. First, if a node has no inferiors, it has no claim to any of the space below it, and receives no weighting when the space for infeiors is divided up among the nodes at a level. After the nodes at one level are drawn, the space below them is divided according to the number of inferiors each node has. After each node at a level has the horizontal allottment for its inferiors set, its inferiors are evenly spaced in this area, governed only by the Single Inferior and Multiple Inferior Rules to insure clarity.

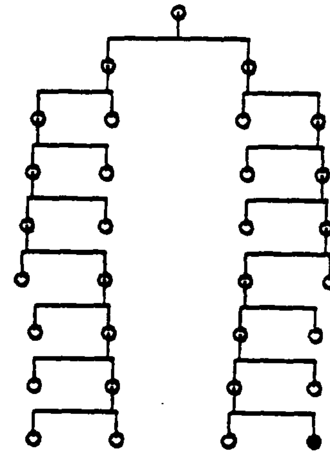


Two trees displayed using the algorithm here presented.

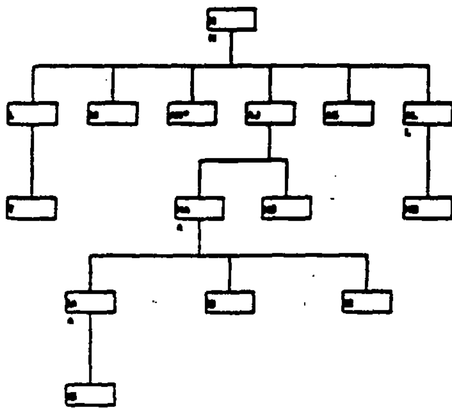
The advantages in efficient space use and logical readability of using these versatile aesthetics is seen when comparing structures graphed with and without more rigid constraints. Forcing isomorphic subtrees to be drawn identically, regardless of where they lie, seems natural, but is actually inefficient as in figure 1. It destroys the ability to optimally utilize display space through creating subtree layouts that are sensitive to global considerations, their positions in the tree as a whole (figure 2).

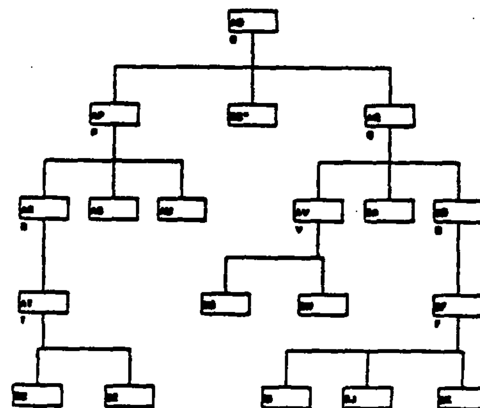


1. A tree displayed using the algorithm TR[1].



2. The same tree displayed without the two limiting aesthetics.





3. Two trees displayed using the algorithm here presented.

To use this algorithm, one must have utilities which, when given a node, can retrieve its superior, and a list of its immediate inferiors. The algorithm works down a tree, moving through the nodes at the first level from left to right, then the second level, and so on until finishing. For each node, the program computes how much space for inferiors the node should be allotted. Then the left and right boundaries of this space are set. Finally, the horizontal (x) position of each of the node's inferiors is set, and the program moves on to the next node. Note that when the program reaches each node it sets the position of the node's inferiors, not the position of the node itself. The node's position has already been set when the program was at this node's superior. Initially the root node of the tree has its x position set to the middle of the horizontal display in order to start the program.

The algorithm takes two actions at each node. First it finds the left boundary and right boundary for drawing the nodes inferiors. Then it actually divides this space up and assigns each inferior a horizontal position.

Here is an outline of the code.

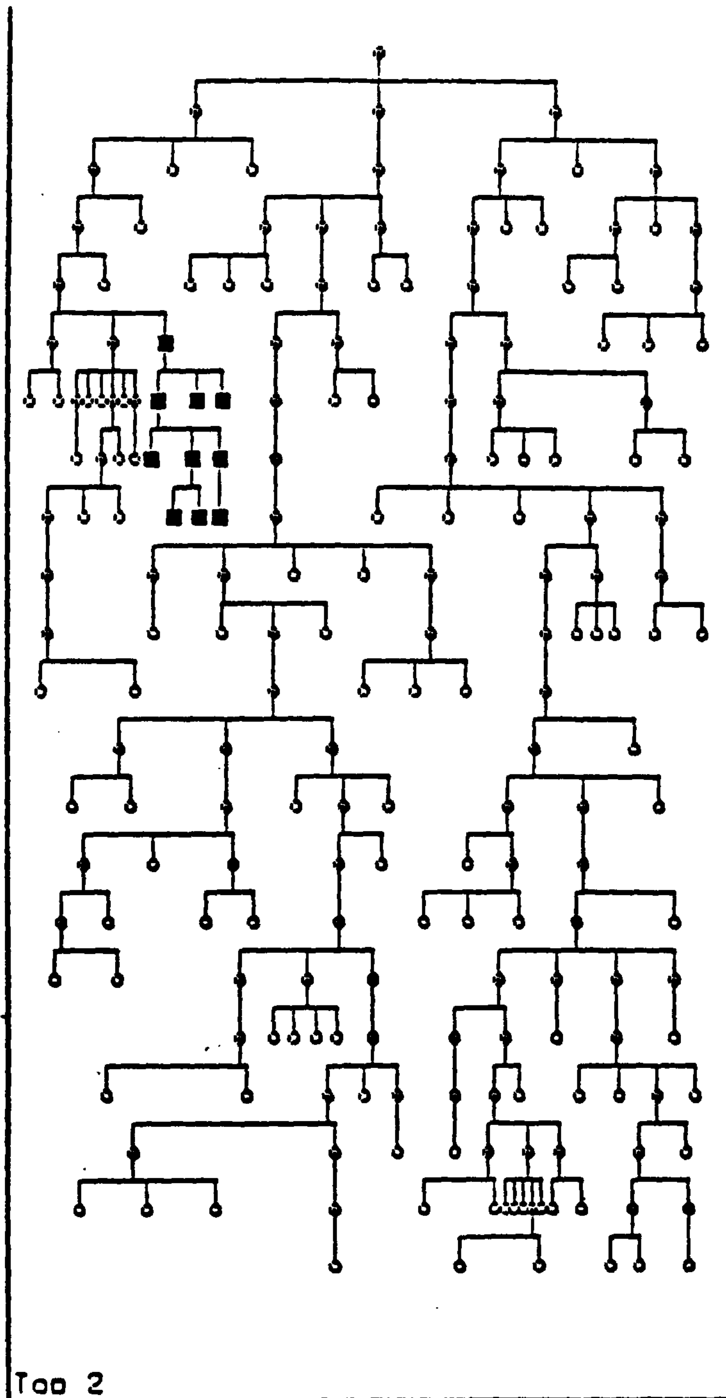
```
I.  Initially at each new level of nodes:
    LEFT-BOUNDARY = 0
    RIGHT-BOUNDARY = 0

  A.  For each node in a level {starting with the leftmost}

    1.  LEFT-BOUNDARY = RIGHT-BOUNDARY.
        ;;New LEFT-BOUNDARY is simply last node's
           RIGHT-BOUNDARY, new RIGHT-BOUNDARY is now found.

    2.  Conditional

          IF no inferiors
            THEN RIGHT-BOUNDARY = LEFT-BOUNDARY
            ;;In effect giving this node no space for inferiors.

          IF last node with inferiors on the level
            THEN RIGHT-BOUNDARY = maximum display width

          OTHERWISE
            ;;A new value for RIGHT-BOUNDARY is found
              using inferior weighting:

                A-NODE = the next node to the right,
                         on this level, with inferiors.
```

```
            SPACE = A-NODE's x - node's x

            MYSONS = # of inferiors of node

            YOURSONS = # of inferiors of A-NODE

        ==> RIGHT-BOUNDARY =
              SPACE * {MYSONS/ MYSONS+YOURSONS}  + {node's x}

3. ;; Left and right boundary are now set.
      For each inferior of the node, assign it an x.

  IF one inferior
    THEN x = node's x

  OTHERWISE
    x = LEFT-BOUNDARY +

          {RIGHT-BOUNDARY - LEFT-BOUNDARY} *

           {this inferior's number / total # of inferiors}

4.  Check layout *

  IF rightmost inferior is to left of node
    THEN its x = the node's x + 1.

  IF leftmost inferior is to right of node
    THEN its x = the node's x - 1.
```

----

* This check is needed to insure clarity. The algorithm will always produce a left-boundary to the left of the node, and a right boundary to the node's right, but in dividing the space by the number of inferiors plus a small buffer space, an inferior may be placed slightly to the wrong side of a node. The correction will never cause an overlap or appear incorrect, for the inferior will always still remain within the node's space boundaries.

In a large tree such as this one, the relatively uniform node density and efficient use of space become evident. The algorithm is not based upon binary trees. It is designed entirely around the non-uniformities encountered in arbitrarily structured trees such as this one.



V. CONCLUSION

This paper presents the problem of laying out unbalanced trees as a difficult and often subjective one. By establishing a flexible set of aesthetics to insure readability, excluding appealing, but overly restrictive rules, an effective algorithm can be designed and is here presented. The power of this algorithm lies in its consistent ability to layout any arbitrary tree in an economical, and highly readable fashion.

Appendix 1

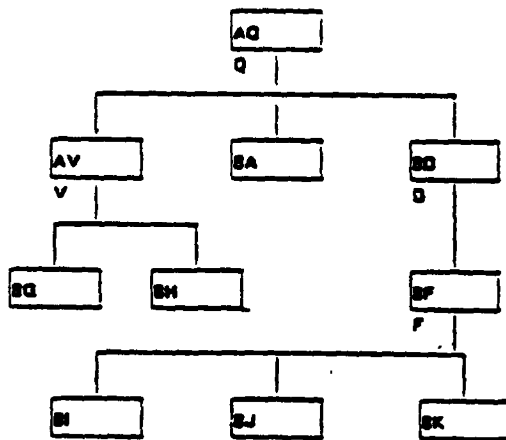
This code is written in zetalisp on a Symbolics 3600 Lisp Machine.

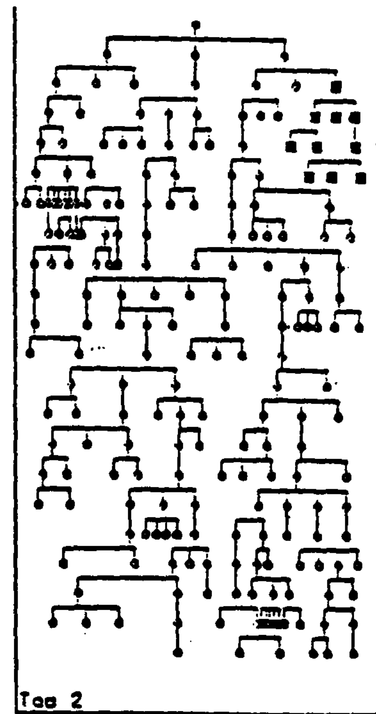
```
(defmethod (top :draw-the-map)(start)
 (multiple-value-bind (width height)
     (send self ':inside-size)         ;binds width and height to screen available screen size
   (let* ((mm (create-level-list start))    ;creates a level list of the tree-
          (depth (length mm)) ;list of lists of the nodes at the successive levels.
          (vstep (// height (+ depth 1)))  ;binds Vertical-STEP to the vertical spacing between levels
          (vpos vstep))       ; initially VPOSition is simply vstep
     (send start          ; set the root's x position to half the total width
          ':set-x
          (// width 2))
     (send start ':set-y vpos)    ; set its y to vpos
     (mapcar             ; for each set of nodes at a new level,
       #'(lambda (level)
          (setq vpos (+ vstep vpos))   ; set vpos to old vpos + vstep
          (cond (( <= vpos (+ 1 (* vstep depth)))  ;finished when vpos > (the number of levels in the tree)
                 (setq lbound 0) ;initially for each level set LeftBOUND and RightBOUND for maximum space.
                 (setq rbound width)
                 (mapcar
                   #'(lambda (node)     ; for each node at a level (moving from left to right)
                       (cond ((null (send node ':inferior-list)); if no inferiors, then set rbound to
                                (setq rbound lbound )) ; lbound,  giving it no space for inferiors.
                              ((null (send node ':superior)))  ; if no superior (the root) or
                              ((null (cdr (memq node        ;   the last node at a level or
                                                level))))
                              ((null (next-node-with-inferiors-on-level (cdr (memq node level)))))
                              ;;last node at level with inferiors,
                              ;; then leave the boundaries where they are, all the space is available
                              (t                   ; otherwise find new value for rbound
                               (let ((a-node (next-node-with-inferiors-on-level
                                                ;; a-node is the next node on level with inferiors
                                                (cdr (memq node
                                                           level )))))
                                 (setq rbound        ;;; Find new value of rbound
                                       (+ (send node ':x)
                                          (*
                                            (//
                                              (float
                                                (length (send node ':inferior-list)))
                                              (+ (length (send node ':inferior-list))
                                                 (length
                                                   (send  a-node ':inferior-list))))
                                            (- (send a-node ':x)
                                               (send node ':x)))))))))
                       (let ((ll (send node ':inferior-list)))   ;; Set the x positions
                         (cond ((equal 1 (length ll))          ;;  of all the node's inferiors
                                ;; If only one inferior node, then it is placed directly below superior
                                (send (car ll) ':set-y vpos)
                                (send (car ll) ':set-x
                                      (send node ':x)))
                               ((null ll))
                               (t                  ;;  For multipiple inferiors,
                                (mapcar     ;;    divide up the space delimited by the values
                                  #'(lambda (son)    ;; of rbound and lbound
                                      (send son ':set-y vpos)
                                      (send son ':set-x
                                            (fix
                                              (+
                                                lbound
                                                *space*    ;;Put a small blank buffer between the
                                                (*          ;; outside inferiors of adjacent nodes.
                                                  (- rbound lbound (* 2 *space*))  ;;Usable width
                                                  (//                     ;;after subtracting edge buffer.
                                                    (float
                                                      (-
                                                        (length ll)   ;;Finds the fraction of the width
                                                        (length (memq son ll)))) ;; in which to
                                                    (- (length ll) 1)))))))     ;;place this inferior.
                                  ll)
```

```
                                     (check node)))  ;;; Checks layout
                              (setq lbound rbound)  ;;For the next node set lbound to the old value of rbound
                              (setq rbound width)))  ;; Initially set rbound to maximum.
                       level)))) ;; this loop operates an the list of nodes at a level
      mm))))     ;; the outer loop operates on the list of lists of nodes at a level (the level-list).

(defun check (node) ;;; Checks that a node's rightmost inferior is to its right etc.
  (let ((sons (send node ':inferior-list)))
    (cond ((<= (send node ':x)
               (send (car sons) ':x))
           (send (car ll) ':set-x
                 (- (send node ':x) 3))))
    (cond ((>= (send node ':x)
               (send (car (last sons)) ':x))
           (send (car (last sons)) ':set-x
                 (- (send node ':x) 3))))))

(defun next-node-with-inferiors-on-level (l)    ;;; Given a node, finds the next node
  (cond ((null l)                               ;;; on the some level in the tree with inferior(s), or else nil.
         nil)
        ((null (send (car l) ':inferior-list))
         (next-node-with-inferiors-on-level (cdr l)))
        (t
         (car l))))
```

Appendix 2

This display algorithm is part of an interactive display interface. This user interface provides a useful system for displaying and editing complex tree structures. The monitor has two graphics areas. One displays an unlabeled, mouse-sensitive layout of the entire tree that is being worked with. Using a mouse, nodes can be selected for expansion in the large display window where detailed labeling is added to explain a node's structure and connection with other nodes. As much of the subtree of a selected node is expanded in the large window as is possible without crowding the display. The nodes of the expanded subtree are highlighted in the overall tree display as a pointer to one's position in the global structure. The entire screen is mouse sensitive for ease of editing.





1) A sample display of the system.

Acknowledgements

Funding and technical support were provided by Dr. Charles Rich and Dr. Richard C. Waters under their Programmer's Apprentice Project. Additional support was provided by MIT's Undergraduate Research Opportunities Program. This paper describes research done at the Artificial Intelligence Laboratory

References

[1] E. Reingold and J. Tilford, "Tidier Drawings of Trees," IEEE Transactions on Software Engineering, Vol. SE-7, pp. 223-228, 1981.

[2] C. Wetherell and A. Shannon, "Tidy Drawings of Trees," IEEE Transactions on Software Engineering, Vol. SE-5, pp. 514-520, 1979.

[3] J.G. Vaucher, "Pretty-Printing of Trees," Software Practice and Experience, Vol. 10, pp. 553-561, 1980.