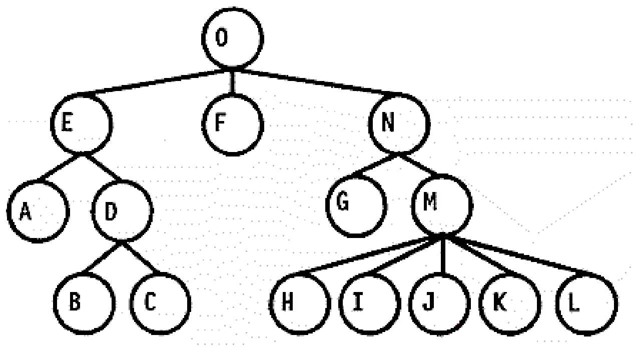
Dr. Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

# Positioning Nodes For General Trees

February 01, 1991
URL:http://www.drdobbs.com/positioning-nodes-for-general-trees/184402320

**Figure 1**



**Listing 1**

```
/***********************************************************
 *  Node-Positioning for General Trees, by John Q. Walker II
 *
 *  Initiated by calling procedure TreePosition().
 **********************************************************/

#include <stdlib.h>

/*----------------------------------------------------------
 * Implementation dependent: Set the values for each of
 * these variables.
 *--------------------------------------------------------*/
#define NODE_WIDTH          20  /* Width of a node?       */
#define NODE_HEIGHT         10  /* Height of a node?      */
#define FRAME_THICKNESS     1   /* Fixed-sized node frame?*/
#define SUBTREE_SEPARATION  5   /* Gap between subtrees?  */
#define SIBLING_SEPARATION  4   /* Gap between siblings?  */
#define LEVEL_SEPARATION    5   /* Gap between levels?    */
#define MAXIMUM_DEPTH       10  /* Biggest tree?          */

/*----------------------------------------------------------
 * Implementation dependent: The structure for one node
 * - The first 4 pointers must be set for each node before
 *   this algorithm is called.
 * - The X and Y coordinates must be set for only the apex
 *   of the tree upon entry; they will be set by this
 *   algorithm for all the other nodes.
 * - The next three elements are used only for the duration
 *   of the algorithm.
 * - Actual contents of the node depend on your application.
 *--------------------------------------------------------*/
typedef int COORD;              /* X,Y coordinate type    */

typedef struct node {
   struct node *parent;        /* ptr: node's parent     */
   struct node *offspring;     /* ptr: leftmost child    */
   struct node *leftsibling;   /* ptr: sibling on left   */
   struct node *rightsibling;  /* ptr: sibling on right  */
   COORD  xCoordinate;         /* node's current x coord */
   COORD  yCoordinate;         /* node's current y coord */

   struct node *prev;          /* ptr: lefthand neighbor */
   float  flPrelim;            /* preliminary x coord    */
   float  flModifier;          /* temporary modifier     */

   char   info[80];            /* pick your contents!    */
} *PNODE;                       /* ptr: a node structure  */

/*----------------------------------------------------------
 * Global variables used by the algorithm
 *--------------------------------------------------------*/
typedef enum { FALSE, TRUE } BOOL;
typedef enum { FRAME, NO_FRAME } FRAME_TYPE;
typedef enum { NORTH, SOUTH, EAST, WEST } ROOT_ORIENTATION;

typedef struct prev_node {      /* one list element        */
   PNODE pPreviousNode;        /* ptr: previous at level  */
   struct prev_node *pNextLevel;   /*  ptr: next element  */
} *PPREVIOUS_NODE;

static FRAME_TYPE FrameType = FRAME;    /*  Show a frame   */
static ROOT_ORIENTATION RootOrientation = NORTH; /*  At top*/
static PPREVIOUS_NODE pLevelZero = (PPREVIOUS_NODE)0;

static COORD xTopAdjustment;    /* How to adjust the apex */
static COORD yTopAdjustment;    /* How to adjust the apex */
static float flMeanWidth;       /* Ave. width of 2 nodes  */

#define FIRST_TIME (0)          /* recursive proc flag    */

/*----------------------------------------------------------
 * Implemented as macros, but could be implemented as
 * procedures depending on your particular node structures
 *--------------------------------------------------------*/

#define FirstChild(node)     ((PNODE)((node)->offspring))
#define LeftSibling(node)    ((PNODE)((node)->leftsibling))
#define RightSibling(node)   ((PNODE)((node)->rightsibling))
#define Parent(node)         ((PNODE)((node)->parent))
#define LeftNeighbor(node)   ((PNODE)((node)->prev))
#define IsLeaf(node)          \
            (((node)&&(!((node)->offspring)))?TRUE:FALSE)
#define HasChild(node)        \
                  (((node)&&((node)->offspring))?TRUE:FALSE)
#define HasLeftSibling(node)   \
             (((node)&&((node)->leftsibling))?TRUE:FALSE)
#define HasRightSibling(node)  \
            (((node)&&((node)->rightsibling))?TRUE:FALSE)

static PNODE GetPrevNodeAtLevel (unsigned nLevelNbr)
{
   /*------------------------------------------------------
    * List Manipulation: Return pointer to previous node at
    * this level
    *----------------------------------------------------*/
   PPREVIOUS_NODE pTempNode;   /* used in the for-loop   */
   unsigned i = 0;             /* level counter          */

   for (pTempNode = pLevelZero; (pTempNode);
       pTempNode = pTempNode->pNextLevel) {
      if (i++ == nLevelNbr)
         /* Reached desired level.  Return its pointer */
         return (pTempNode->pPreviousNode);
   }
   return ((PNODE)0); /* No pointer yet for this level. */
}

static BOOL SetPrevNodeAtLevel (unsigned nLevelNbr,
                           PNODE pThisNode)
{

   /*------------------------------------------------------
    * List Manipulation: Set the list element to the
    * previous node at this level
    *----------------------------------------------------*/

   PPREVIOUS_NODE pTempNode;   /* used in the for-loop   */
   PPREVIOUS_NODE pNewNode;    /* newly-allocated memory */
   unsigned i = 0;             /* level counter          */

   for (pTempNode = pLevelZero; (pTempNode);
       pTempNode = pTempNode->pNextLevel) {
      if (i++ == nLevelNbr) {
         /* Reached desired level. Return its pointer */
         pTempNode->pPreviousNode = pThisNode;
         return (TRUE);
      }
      else if (pTempNode->pNextLevel==(PPREVIOUS_NODE)0) {
         /* Looks like we need a new level: add it.    */
         /* We need to keep going--should be the next. */
         pNewNode = (PPREVIOUS_NODE)
                  malloc(sizeof(struct prev_node));
         if (pNewNode) {
            pNewNode->pPreviousNode = (PNODE)0;
            pNewNode->pNextLevel = (PPREVIOUS_NODE)0;
            pTempNode->pNextLevel = pNewNode;
         }
         else return (FALSE);    /* The malloc failed. */
      }
   }

   /* Should only get here if pLevelZero is 0.           */
   pLevelZero = (PPREVIOUS_NODE)
              malloc(sizeof(struct prev_node));
   if (pLevelZero) {
      pLevelZero->pPreviousNode = pThisNode;
      pLevelZero->pNextLevel = (PPREVIOUS_NODE)0;
      return (TRUE);
   }
   else return (FALSE);        /* The malloc() failed.   */
```

```
}

static void InitPrevNodeAtLevel (void)
{
   /*----------------------------------------------------
    * List Manipulation: Initialize the list of the
    * previous node at each level to all zeros.
    *--------------------------------------------------*/

   PPREVIOUS_NODE pTempNode = pLevelZero; /*  the start    */

   for ( ; (pTempNode); pTempNode = pTempNode->pNextLevel)
      pTempNode->pPreviousNode = (PNODE)0;
}

static BOOL CheckExtentsRange(long lxTemp, long lyTemp)
{
   /*---------------------------------------------------
    * Insert your own code here, to check when the
    * tree drawing is going to be too big. My region is no
    * more that 64000 units square.
    *-------------------------------------------------*/

   if ((labs(lxTemp) > 32000) || (labs(lyTemp) > 32000))
      return (FALSE);
   else
      return (TRUE);
}

static void TreeMeanNodeSize (PNODE pLeftNode,
                         PNODE pRightNode)
{
   /*----------------------------------------------------
    * Write your own code for this procedure if your
    * rendered nodes will have variable sizes.
    *----------------------------------------------------
    * Here I add the width of the contents of the
    * right half of the pLeftNode to the left half of the
    * right node. Since the size of the contents for all
    * nodes is currently the same, this module computes the
    * following trivial computation.
    *--------------------------------------------------*/

   flMeanWidth = (float)0.0;   /* Initialize this global */

   switch (RootOrientation) {
      case NORTH:
      case SOUTH:
         if (pLeftNode) {
            flMeanWidth = flMeanWidth +
                       (float)(NODE_WIDTH/2);
            if (FrameType != NO_FRAME)
               flMeanWidth = flMeanWidth +
                             (float)FRAME_THICKNESS;
         }
         if (pRightNode) {
            flMeanWidth = flMeanWidth +
                       (float)(NODE_WIDTH/2);
            if (FrameType != NO_FRAME)
                flMeanWidth = flMeanWidth +
                              (float)FRAME_THICKNESS;
         }
         break;
      case EAST :
      case WEST :
         if (pLeftNode) {
            flMeanWidth = flMeanWidth +
                       (float)(NODE_HEIGHT/2);
            if (FrameType != NO_FRAME)
                flMeanWidth = flMeanWidth +
                              (float)FRAME_THICKNESS;
         }
         if (pRightNode) {
            flMeanWidth = flMeanWidth +
                       (float)(NODE_HEIGHT/2);
            if (FrameType != NO_FRAME)
                flMeanWidth = flMeanWidth +
                              (float)FRAME_THICKNESS;
         }
         break;
   }
}

static PNODE TreeGetLeftmost(PNODE pThisNode,
                        unsigned nCurrentLevel,
                        unsigned nSearchDepth)
{
   /*----------------------------------------------------
    * Determine the leftmost descendant of a node at a
    * given depth. This is implemented using a post-order
    * walk of the subtree under pThisNode, down to the
    * level of nSearchDepth. If we've searched to the
    * proper distance, return the currently leftmost node.
    * Otherwise, recursively look at the progressively
    * lower levels.
    *--------------------------------------------------*/
   PNODE pLeftmost;    /* leftmost descendant at level   */
   PNODE pRightmost;   /* rightmost offspring in search  */

   if (nCurrentLevel == nSearchDepth)
      pLeftmost = pThisNode; /*  searched far enough.    */
   else if (IsLeaf(pThisNode))
      pLeftmost = 0;  /* This node has no descendants    */
   else {  /* Do a post-order walk of the subtree.        */
      for (pLeftmost = TreeGetLeftmost(pRightmost =
                          FirstChild(pThisNode),
                          nCurrentLevel + 1,
                          nSearchDepth)

         ;
         (pLeftmost==0) && (HasRightSibling(pRightmost))
         ;
         pLeftmost = TreeGetLeftmost(pRightmost =
                          RightSibling(pRightmost),
                          nCurrentLevel + 1,
                          nSearchDepth)
         ) { /* Nothing inside this for-loop. */ }
   }
   return (pLeftmost);
}

static void TreeApportion (PNODE pThisNode,
                      unsigned nCurrentLevel)
{
   /*------------------------------------------------------
    * Clean up the positioning of small sibling subtrees.
    * Subtrees of a node are formed independently and
    * placed as close together as possible. By requiring
    * that the subtrees be rigid at the time they are put
    * together, we avoid the undesirable effects that can
    * accrue from positioning nodes rather than subtrees.
    *----------------------------------------------------*/

   PNODE pLeftmost;            /* leftmost at given level*/
   PNODE pNeighbor;            /* node left of pLeftmost */
   PNODE pAncestorLeftmost;    /* ancestor of pLeftmost  */
   PNODE pAncestorNeighbor;    /* ancestor of pNeighbor  */
   PNODE pTempPtr;             /* loop control pointer   */
   unsigned i;                 /* loop control           */
   unsigned nCompareDepth;     /* depth of comparison    */
                          /* within this proc       */
   unsigned nDepthToStop;      /* depth to halt          */
   unsigned nLeftSiblings;     /* nbr of siblings to the */
           /* left of pThisNode, including pThisNode,  */
           /* til the ancestor of pNeighbor            */
   float flLeftModsum;         /* sum of ancestral mods  */
   float flRightModsum;        /* sum of ancestral mods  */
   float flDistance;           /* difference between     */
        /* where pNeighbor thinks pLeftmost should be   */
        /* and where pLeftmost actually is              */
   float flPortion;            /* proportion of          */
        /* flDistance to be added to each sibling       */

   pLeftmost = FirstChild(pThisNode);
   pNeighbor = LeftNeighbor(pLeftmost);

   nCompareDepth = 1;
   nDepthToStop = MAXIMUM_DEPTH - nCurrentLevel;

   while ((pLeftmost) && (pNeighbor) &&
         (nCompareDepth <= nDepthToStop)) {

      /* Compute the location of pLeftmost and where it */
      /* should be with respect to pNeighbor.           */
      flRightModsum = flLeftModsum = (float)0.0;
      pAncestorLeftmost = pLeftmost;
      pAncestorNeighbor = pNeighbor;
      for (i = 0; (i < nCompareDepth); i++) {
         pAncestorLeftmost = Parent(pAncestorLeftmost);
         pAncestorNeighbor = Parent(pAncestorNeighbor);
         flRightModsum = flRightModsum +
                      pAncestorLeftmost->flModifier;
         flLeftModsum = flLeftModsum +
                      pAncestorNeighbor->flModifier;

      }

      /* Determine the flDistance to be moved, and apply*/
      /* it to "pThisNode's" subtree.  Apply appropriate*/
      /* portions to smaller interior subtrees          */

      /* Set the global mean width of these two nodes   */
      TreeMeanNodeSize(pLeftmost, pNeighbor);

      flDistance = (pNeighbor->flPrelim +
                  flLeftModsum +
                  (float)SUBTREE_SEPARATION +
                  (float)flMeanWidth) -
                 (pLeftmost->flPrelim + flRightModsum);

      if (flDistance > (float)0.0) {
         /* Count the interior sibling subtrees        */
         nLeftSiblings = 0;
         for (pTempPtr = pThisNode;
              (pTempPtr) &&
              (pTempPtr != pAncestorNeighbor);
             pTempPtr = Leftsibling(pTempPtr)) {
            nLeftSiblings++;
         }

         if (pTempPtr) {
            /* Apply portions to appropriate          */
            /* leftsibling subtrees.                  */
            flPortion = flDistance/(float)nLeftSiblings;
            for (pTempPtr = pThisNode;
                 (pTempPtr != pAncestorNeighbor);
                 pTempPtr = LeftSibling(pTempPtr)) {
               pTempPtr->flPrelim =
                    pTempPtr->flPrelim + flDistance;
               pTempPtr->flModifier =
                    pTempPtr->flModifier + flDistance;
               flDistance = flDistance - flPortion;
             }
         }
         else {
            /* Don't need to move anything--it needs  */
            /* to be done by an ancestor because      */
            /* pAncestorNeighbor and                  */
            /* pAncestorLeftmost are not siblings of  */
            /* each other.                            */
            return;
         }
      }  /* end of the while                           */

      /* Determine the leftmost descendant of pThisNode */
      /* at the next lower level to compare its         */
      /* positioning against that of its pNeighbor.     */

      nCompareDepth++;
      if (IsLeaf(pLeftmost))
         pLeftmost = TreeGetLeftmost(pThisNode, 0,
                                     nCompareDepth);
      else
         pLeftmost = FirstChild(pLeftmost);
      pNeighbor = LeftNeighbor(pLeftmost);
   }
 }

 static BOOL TreeFirstWalk(PNODE pThisNode,
                      unsigned nCurrentLevel)
 {
   /*-----------------------------------------------------
    * In a first post-order walk, every node of the tree is
    * assigned a preliminary x-coordinate (held in field
    * node->flPrelim). In addition, internal nodes are
    * given modifiers, which will be used to move their
    * children to the right (held in field
    * node->flModifier).
    * Returns: TRUE if no errors, otherwise returns FALSE.
    *----------------------------------------------------*/

   PNODE pLeftmost;            /* left- & rightmost      */
   PNODE pRightmost;           /* children of a node.    */
   float flMidpoint;           /* midpoint between left- */
                         /* & rightmost children   */

   /* Set up the pointer to previous node at this level  */
   pThisNode->prev = GetPrevNodeAtLevel(nCurrentLevel);

   /* Now we're it--the previous node at this level      */
   if (!(SetPrevNodeAtLevel(nCurrentLevel, pThisNode)))
      return (FALSE);        /* Can't allocate element  */

   /* Clean up old values in a node's flModifier         */
   pThisNode->flModifier = (float)0.0;

   if ((IsLeaf(pThisNode)) ||
      (nCurrentLevel == MAXIMUM_DEPTH)) {
      if (HasLeftSibling(pThisNode)) {

      /*--------------------------------------------
       * Determine the preliminary x-coordinate
       *   based on:
       * - preliminary x-coordinate of left sibling,
       * - the separation between sibling nodes, and
       * - mean width of left sibling & current node.
       *--------------------------------------------*/
      /* Set the mean width of these two nodes      */
      TreeMeanNodeSize(LeftSibling(pThisNode),
                   pThisNode);

      pThisNode->flPrelim =
                (pThisNode->Leftsibling->flPrelim) +
                (float)SIBLING_SEPARATION +
                flMeanWidth;
   }
   else    /*  no sibling on the left to worry about  */
      pThisNode->flPrelim = (float)0.0;
 }
 else {
    /* Position the leftmost of the children          */
    if (TreeFirstWalk(pLeftmost = pRightmost =
                  FirstChild(pThisNode),
                  nCurrentLevel + 1)) {
       /* Position each of its siblings to its right */
       while (HasRightSibling(pRightmost)) {
          if (TreeFirstWalk(pRightmost =
                        RightSibling(pRightmost),
                        nCurrentLevel + 1)) {
          }
          else return (FALSE); /* malloc() failed   */
       }

       /* Calculate the preliminary value between   */
       /* the children at the far left and right    */
       flMidpoint = (pLeftmost->flPrelim +
                  pRightmost->flPrelim)/(2.0);

       /* Set global mean width of these two nodes  */
       TreeMeanNodeSize(LeftSibling(pThisNode),
                    pThisNode);

          if (HasLeftSibling(pThisNode)) {
             pThisNode->flPrelim =
                    (pThisNode->leftsibling->flPreLim) +
                            (float)SIBLING_SEPARATION +
                            flMeanWidth;
             pThisNode->flModifier =
                 pThisNode->flPrelim - flMidpoint;
             TreeApportion(pThisNode, nCurrentLevel);
          }
          else pThisNode->flPrelim = flMidpoint;
       }
       else return (FALSE); /* Couldn't get an element  */
    }
    return (TRUE);
 }

 static BOOL TreeSecondWalk(PNODE pThisNode,
                      unsigned nCurrentLevel)
 {
    /*-----------------------------------------------------
     * During a second pre-order walk, each node is given a
     * final x-coordinate by summing its preliminary
     * x-coordinate and the modifiers of all the node's
     * ancestors.  The y-coordinate depends on the height of
     * the tree.  (The roles of x and y are reversed for
     * RootOrientations of EAST or WEST.)
     * Returns: TRUE if no errors, otherwise returns FALSE.
     *---------------------------------------- ----------*/

    BOOL bResult = TRUE;        /* assume innocent        */
    long lxTemp, lyTemp;        /* hold calculations here */
    float flNewModsum;          /* local modifier value   */
    static float flModsum = (float)0.0;

    if (nCurrentLevel <= MAXIMUM_DEPTH) {
       flNewModsum = flModsum;  /* Save the current value  */
       switch (RootOrientation) {
          case NORTH:
             lxTemp = (long)xTopAdjustment +
               (long)(pThisNode->flPrelim + flModsum);
             lyTemp = (long)yTopAdjustment +
               (long)(nCurrentLevel * LEVEL_SEPARATION);
             break;
          case SOUTH:
             lxTemp = (long)xTopAdjustment +
               (long)(pThisNode->flPrelim + flModsum);
             lyTemp = (long)yTopAdjustment -
               (long)(nCurrentLevel * LEVEL_SEPARATION);
             break;
          case EAST:
             lxTemp = (long)xTopAdjustment +
               (long)(nCurrentLevel * LEVEL_SEPARATION);
             lyTemp = (long)yTopAdjustment -
               (long)(pThisNode->flPrelim + flModsum);
             break;
          case WEST:
             lxTemp = (long)xTopAdjustment -
               (long)(nCurrentLevel * LEVEL_SEPARATION);
             lyTemp = (long)yTopAdjustment -
               (long)(pThisNode->flPrelim + flModsum);
             break;
       }

       if (CheckExtentsRange(lxTemp, lyTemp)) {
          /* The values are within the allowable range */

          pThisNode->xCoordinate = (COORD)lxTemp;
          pThisNode->yCoordinate = (COORD)lyTemp;

          if (HasChild(pThisNode)) {
             /* Apply the flModifier value for this    */
             /* node to all its offspring.             */
             flModsum = flNewModsum =
                    flNewModsum + pThisNode->flModifier;
             bResult = TreeSecondWalk(
                FirstChild(pThisNode),nCurrentLevel + 1);
             flNewModsum = flNewModsum -
                         pThisNode->flModifier;
          }

          if ((HasRightSibling(pThisNode)) && (bResult)) {
             flModsum = flNewModsum;
             bResult = TreeSecondWalk(
                 RightSibling(pThisNode), nCurrentLevel);
          }
       }
       else bResult = FALSE;   /* outside of extents   */
    }
    return (bResult);
 }

 static BOOL TreePosition(PNODE pApexNode)
 {
    /*----------------------------------------------------
     * Determine the coordinates for each node in a tree.
     * Input: Pointer to the apex node of the tree
     * Assumption: The x & y coordinates of the apex node
     * are already correct, since the tree underneath it
     * will be positioned with respect to those coordinates.
     * Returns: TRUE if no errors, otherwise returns FALSE.
     *----------------------------------------------------*/

    if (pApexNode) {
       /* Initialize list of previous node at each level */
       InitPrevNodeAtLevel();

       /* Generate the properly-placed tree nodes.      */
       /* TreeFirstWalk: a post-order walk              */
       /* TreeSecondWalk: a pre-order walk              */
       if (TreeFirstWalk (pApexNode, FIRST_TIME)) {
          /* Determine how to adjust the nodes with     */
          /* respect to the location of the apex of the */
          /* tree being positioned.                     */
          switch (RootOrientation) {
             case NORTH:
             case SOUTH:
                /* Create the adjustment from x-coord  */
                xTopAdjustment = pApexNode->xCoordinate -
                         (COORD)(pApexNode->flPrelim);
                yTopAdjustment = pApexNode->yCoordinate;
                break;
             case EAST :
             case WEST :
                /* Create the adjustment from y-coord  */
                xTopAdjustment = pApexNode->xCoordinate;
                yTopAdjustment = pApexNode->yCoordinate +
                         (COORD)(pApexNode->flPrelim);
                break;
          }
          return (TreeSecondWalk(pApexNode, FIRST_TIME));
       }
       else return (FALSE); /*  Couldn't get an element   */
    }
    else return (TRUE); /*  Easy: null pointer was passed  */
 }
```

**Table 1**

| **Node** | **Final X-coordinate (preliminary x-coordinate + modifiers of ancestors)** |
|---|---|
| O | 13.5 |
| E | 3 + 0 = 3 |
| A | 0 + 0 + 0 = 0 |
| D | 6 + 0 + 0 = 6 |
| B | 0 + 3 + 0 + 0 = 3 |
| C | 6 + 3 + 0 + 0 = 9 |
| F | 13.5 + 0 = 13.5 |
| N | 24 + 0 = 24 |
| G | 0 + 21 + 0 = 21 |
| M | 6 + 21 + 0 = 27 |
| H | 0 + -6 + 21 + 0 = 15 |
| I | 6 + -6 + 21 + 0 = 21 |
| J | 12 + -6 + 21 + 0 = 27 |
| K | 18 + -6 + 21 + 0 = 33 |
| L | 24 + -6 + 21 + 0 = 39 |

# Positioning Nodes For General Trees

**John Q. Walker II**

*John Q. Walker II manages one of the software development teams implementing the SNA communications protocols on OS/2. He joined IBM in 1978, where he was a developer and tester for the operating system software of the IBM System/38. In Research Triangle Park, NC, John has been an architect for the IBM Token-Ring Network, serving as co-editor of the IEEE 802.5 local area network standard in 1983 and 1984. John received a B.S., B.A., and M.S. from Southern Illinois University and is completing a Ph.D. in computer science at the University of North Carolina at Chapel Hill. You may contact John at IBM Corp., Dept. E42, Bldg. 673, P.O. Box 12195, Research Triangle Park, NC 27709.*

*The bulk of this article first appeared in Software - Practice and Experience, July 1990, Copyright 1990 by John Wiley and Sons, Ltd. Reprinted by permission.*

Drawing a tree consists of two stages: determining the position of each node, and actually rendering the individuals nodes and interconnecting branches. The algorithm described here concerns the first stage. That is, given a list of nodes, an indication of the hierarchical relationship among them, and their shape and size, where should each node be positioned for optimal aesthetic effect?

This algorithm determines the positions of the nodes for an arbitrary general tree. The positioning, specified in *x*, *y* coordinates, minimizes the width of the tree. A general tree may have an unlimited number of offspring per node. Binary and ternary trees, on the other hand, can have only two and three offspring per node, respectively. This algorithm operates in time *0(N)*, where *N* is the number of nodes in the tree.

In textbooks, most tree drawings have been positioned by the sure hand of a human graphic designer. Many computer-generated positionings have been either trivial or contained irregularities. Earlier work by Wetherell and Shannon [8] and Tilford [6], upon which the algorithm builds, failed to correctly position the interior nodes of some trees. The algorithm in this article correctly positions a tree's nodes in two passes. It also handles several practical considerations: alternate orientations of the tree, variable node sizes, and out-of-bounds conditions. Radack [2], also building on Tilford's work, solved this same problem with an algorithm that makes four passes.

A history of algorithms leading up to this one is presented in reference [7].

This algorithm lets a computer reliably generate tree drawings equivalent to those done by a skilled human. Below are some of the applications that often use tree-drawings.

- Drawings of B-trees and 2-3 trees
- Structure editors that draw trees
- Flow charts without loops
- Visual LISP editors
- Parse trees
- Decision trees
- Hierarchical database models
- Hierarchically-organized file systems (for example, directories, subdirectories, and files)
- Organizational charts
- Tables of contents in printed matter
- Biological classifications

**General Trees**

This paper deals with rooted, directed trees, that is, trees with one root and hierarchical connections from the root to its offspring. No node may have more than one parent.

Each node in a general tree can have an unlimited number of offspring. A general tree is also known as an *m*-ary tree, since each node can have m offspring (where *m* is *0* or more). As a class, binary trees differ from general trees in that an offspring of a binary tree node must be either the left offspring or the right offspring. Binary tree drawings preserve this left-right distinction. Thus, a single offspring is placed under its parent node either to the left or right of its parent's position. This left-right distinction does not apply in a general tree. If a node has a single offspring, the offspring is placed directly below its parent.

This algorithm positions a binary tree by ignoring the left-right distinction. If a node has exactly one offspring, it is positioned directly below its parent. Supowit and Reingold [4] noted that it is NP-hard to optimally position minimum-width binary trees (while adhering to the left-right distinction) to within a factor of less than about four percent.

**Aesthetic Rules**

Wetherell and Shannon [8] first described a set of aesthetic rules against which a good positioning algorithm must be judged. Tilford [6] and Supowit and Reingold [4] expanded that list in an effort to produce better algorithms.

A tidy tree drawing occupies as little space as possible while satisfying certain aesthetics:

1. Nodes at the same level of the tree should lie along a straight line, and the straight lines defining the levels should be parallel [8].

2. A parent should be centered over its offspring [8].

3. A tree and its mirror image should produce drawings that are reflections of one another. Moreover, a subtree should be drawn the same way regardless of where it occurs in the tree. In some applications, one wishes to examine large trees to find repeated patterns. Searching for patterns is facilitated by having isomorphic subtrees drawn isomorphically [3].

This implies that small subtrees should not appear arbitrarily positioned among larger subtrees. Small, interior subtrees should be spaced out evenly among larger subtrees (where the larger subtrees are adjacent at one or more levels). Small subtrees at the far left or far right should be adjacent to larger subtrees.

The algorithm described in this article satisfies these aesthetic rules.

**How The Algorithm Works**

This algorithm initially assumes that the root of the tree is at the top of the drawing [1]. Node-positioning algorithms are concerned only with determining the *x*-coordinates of the nodes. The *y*-coordinate can be determined from its level in the tree, due to Aesthetic 1 and the convention of providing a uniform vertical separation between consecutive levels. A later section presents a variation for placing the root in other positions.

This algorithm uses two positioning concepts, the first of which is building subtrees as rigid units. Moving a node moves all of its descendants (if it has any) — the entire subtree is thus treated as a rigid unit. A general tree is positioned by building it up recursively from its leaves toward its root.

Second is the concept of using two fields for the positioning of each node. They are a preliminary *x*-coordinate, and a modifier field.

Two tree traversals are used to produce a node's final *x*-coordinate. The first traversal assigns the preliminary *x*-coordinate and modifier fields for each node. The second traversal computes the final *x*-coordinate of each node by summing the node's preliminary *x*-coordinate with the modifier fields of all of its ancestors. This technique makes moving a large subtree simple and allows the algorithm to operate in time *0(N)*.

For example, to move a subtree four units to the right, increment both the preliminary *x*-coordinate and the modifier field of the subtree's root by four. As another example, the modifier field associated with the tree's apex node is used in determining the final position of all of its descendants. (I use the term *apex node* to distinguish the root of the entire tree from the roots of individual internal subtrees.)

The first tree traversal is a postorder traversal, positioning the smallest subtrees (the leaves) first and recursively proceeding from left to right to build up the position of larger and larger subtrees. Sibling nodes are always separated from one another by a predefined minimal distance (the sibling separation). Adjacent subtrees are separated by at least a predefined subtree separation. Subtrees of a node are formed independently and placed as close together as these separation values allow.

As the tree walk moves from the leaves to the apex, it combines smaller subtrees and their root to form a larger subtree. For a given node, its subtrees are positioned one-by-one, moving left to right. Imagine that its newest subtree has been drawn and cut out of paper along its contour. Now superimpose the new subtree atop its neighbor to the left, and move them apart until no two points touch. Initially the subtrees' roots are separated by the sibling separation value. The roots are then pushed apart until the adjacent subtrees at the lower level are separated by the subtree separation value. This process continues at successively lower levels until it arrives at the bottom of the shorter subtree. Note that the new subtree being superimposed may not always bump against a descendant of its nearest sibling to the left. Siblings much farther to the left, but with many offspring, may cause the new subtree to be pushed to the right. At some levels no movement may be necessary, but at no level are subtrees moved closer together. When this process is complete for all of the offspring of a node, the node is centered over its leftmost and rightmost offspring.

Pushing a new, large subtree farther and farther to the right may open a gap between the large subtree and smaller subtrees that had been positioned correctly, but are now bunched on the left with an empty area to their right. This undesirable left-to-right gluing was the failing of the algorithms by Sweet, Wetherell and Shannon, and Tilford.

The algorithm presented here produces evenly distributed, proportional spacing among subtrees. The distance that a large subtree is moved is apportioned to smaller, interior subtrees, satisfying Aesthetic 3. Moving these subtrees is accomplished as before — by adding the proportional values to the preliminary *x*-coordinate and modifier fields of the roots of the small interior subtrees. For example, if three small subtrees are bunched at the left because a new large subtree has been positioned to the right, the first small subtree is shifted right by one-fourth of the gap, the second small subtree is shifted right by one-half of the gap, and the third small subtree is shifted right by three-quarters of the gap.

The second tree traversal, a preorder traversal, determines the final x-coordinate for each node. It starts at the apex node and sums each node's x-coordinate value with the combined sum of the modifier fields of its ancestors. The second traversal also adds a value that guarantees centering the display with respect to the position of the apex node.

**Changing Root Orientation**

The algorithm allows tree positionings where the apex is not at the top of the drawing. One such orientation puts the root on the left and the siblings to its right. Four such orientations of the root are the values taken by the global constant *RootOrientation*, set before the algorithm is called.

*NORTH* — root is at the top, as in the preceding algorithm

*SOUTH* — root is at the bottom, its siblings are above it

*EAST* — root is at the right, its siblings are to its left

*WEST* — root is at the left, its siblings are to its right

**An Extended Example**

The algorithm's operation during the two walks can be best illustrated with an example (see Figure 1) . At least three levels are needed to illustrate its operation, since a small subtree must be centered between larger sibling subtrees. The following example has fifteen nodes lettered in the order that they are visited in the first, postorder traversal. For this example, the mean size of each node is two units and the sibling separation and subtree separation values are four units.

In the postorder walk, nodes' preliminary *x*-coordinate value and modifier values are calculated.

*A* is a leaf with no left sibling.

```
PRELIM(A) = 0
MODIFIER(A) = 0
```

*B* is also a leaf with no left sibling.

```
PRELIM(B) = 0
MODIFIER(B) = 0
```

*C* is the right sibling of node *B* and is separated from node *B* by the sibling separation value plus the mean size of the two nodes.

```
PRELIM(C) = 0 + 4 + 2 = 6
MODIFIER(C) = 0
```

*D* is the parent of nodes *B* and *C*, and the right sibling of node *A*. *D* is separated from node *A* by the sibling separation value plus the mean size of the two nodes. *D*'s modifier value is set so that applying it to nodes *B* and *C* will center them under *D*. The modifier is determined by subtracting the mean of the *PRELIM* (its mostly widely-separated offspring) values from *PRELIM(D)*.

```
PRELIM(D) = 0 + 4 + 2 = 6
MODIFIER(D) = 6 - (0 + 6)/2 = 3
```

*E* is the parent of nodes *A* and *D*, and is centered over nodes *A* and *D*.

```
PRELIM(E) = (0 + 6)/2 = 3
MODIFIER(E) = 0
```

*F* is a right sibling of node *E*, and is separated from *E* by the sibling separation value plus the mean size of the two nodes. That would place it directly over node *C*. You can see now that node *N*'s subtree will later be placed much further to the right, leaving the spacing between nodes *E* and *F* smaller, and hence different, than the spacing between nodes *F* and *N*. When node *N* is finally positioned, the position of node *F* will be adjusted. But for now,

```
PRELIM(F) = 3 + 4 + 2 = 9
MODIFIER(F) = 0
```

*G* and *H* are leaves with no left sibling.

```
PRELIM(G) = 0
MODIFIER(G) = 0

PRELIM(H) = 0
MODIFIER(H) = 0
```

*I* is the right sibling of node *H*. It is separated from *H* by the sibling separation value plus the mean size of the two nodes.

```
PRELIM(I) = 0 + 4 + 2 = 6
MODIFIER(I) = 0
```

*J* is the right sibling of node *I*. As before, it is separated by the standard spacing from node *I*.

```
PRELIM(J) = 6 + 4 + 2 = 12
MODIFIER(J) = 0
```

*K* is the right sibling of node *J*, and *L* is the right sibling of node *K*.

```
PRELIM(K) = 12 + 4 + 2 = 18
MODIFIER(K) = 0

PRELIM(L) = 18 + 4 + 2 = 24
MODIFIER(L) = 0
```

*M* is the parent of nodes *H* through *L*, and is the right sibling of node *G*. It is separated from node *G* by the sibling separation value plus the mean size of the two nodes. Its modifier is set so that when it is applied to nodes *H* through *L*, they will appear centered underneath it.

```
PRELIM(M) = 0 + 4 + 2 = 6
MODIFIER(M) = 6 -  (0 + 24)/2 = -6
```

*N* is the parent of nodes *G* and *M*, and is the right sibling of node *F*. *N* first receives its standard positioning to the right of node *F*, with a modifier that reflects the centering of its offspring beneath it.

```
PRELIM(N) = 9 + 4 + 2 = 15
MODIFIER(N) = 15 - (0 + 6)/2 = 12
```

Now verify that node *E*'s subtree and node *N*'s subtree are properly separated.

Move down one level. The leftmost descendant of node *N*, node *G*, currently has a positioning of 0 + 12 = 12 (*PRELIM(G)* plus the *MODIFIER(N)*, its parent). The rightmost descendant of node *E*, node *D* is positioned at 6 + 0 = 6 (*PRELIM(D)* plus the *MODIFIER(E)*, its parent). Their difference is 12 - 6 = 6, which is equal to the minimum separation (subtree separation plus mean node size), so *N*'s subtree does not need to be moved, since there is no overlap at this level.

Move down another level. The leftmost descendant of node *N* is node *H*. It is positioned at 0 + -6 + 12 = 6 (*PRELIM(H*) plus *MODIFIER(M)* and *MODIFIER(N)*). The rightmost descendant of node *E*, node *C*, is positioned at 6 + 3 + 0 = 9 (*PRELIM(C)* plus *MODIFIER(D)* and *MODIFIER(E)*). Their difference is 6 - 9 = -3; it should be 6, the minimum subtree separation plus the mean node size. Thus node *N* and its subtree need to be moved to the right a distance of 6 - -3 = 9.

```
PRELIM(N) = 15 + 9 = 24
MODIFIER(N) = 12 + 9 = 21
```

Moving node *N* and its subtree opens a gap of size nine between sibling nodes *E* and *N*. This difference must be evenly distributed to all contained sibling nodes, and node *F* is the only one. It is moved to the right a distance of 9/2 = 4.5.

```
PRELIM(F) = 9 + 4.5 = 13.5
MODIFIER(F) = 0 + 4.5 = 4.5
```

*0* is the parent of nodes *E*, *F*, and *N*. It is positioned halfway between the position of nodes *E* and *N*.

```
PRELIM(0) = (3 + 24)/2 = 13.5
MODIFIER(0) = 0
```

All the nodes are visited a second time in a preorder traversal. Their final *x*-coordinates are determined by summing their preliminary *x*-coordinates with the modifier fields of all of their ancestors. (See Table 1. )

## The Algorithm

Since the algorithm operates by making two recursive walks of the tree, several variables are global for the sake of runtime efficiency. These variables are described below, alphabetically. All other variables are local to their respective procedures and functions.

*pLevelZero*: The algorithm maintains a list containing the previous node at each level, that is, the adjacent neighbor to the left. *pLevelZero* is a pointer to the first entry in this list.

*xTopAdjustment*: A fixed distance used in the final walk of the tree to determine the absolute *x*-coordinate of a node with respect to the apex node of the tree.

*yTopAdjustment*: A fixed distance used in the final walk of the tree to determine the absolute *y*-coordinate of a node with respect to the apex node of the tree.

The following global constants must be set before the algorithm is called.

*LevelSeparation*: The fixed distance between adjacent levels of the tree. Used in determining the *y*-coordinate of a node being positioned.

*MaximumDepth*: The maximum number of levels in the tree to be positioned. If all levels are to be positioned, set this value to positive infinity (or an appropriate numerical value).

*SiblingSeparation*: The minimum distance between adjacent siblings of the tree.

*SubtreeSeparation*: The minimum distance between adjacent subtrees of a tree. For proper aesthetics, this value is normally somewhat larger than *SiblingSeparation*.

Upon entry to function *TreePosition*, four hierarchical relationships are required for each node. Also, the *X* and *Y* coordinates of the apex node are required. Upon its successful completion, the algorithm has set the *X* and *Y* coordinate values for each node in the tree.

The algorithm is invoked by calling function *TreePosition*, passing it a pointer to the apex node of the tree. If the tree is too wide or too tall to be positioned within the coordinate system being used, *TreePosition* returns the boolean *FALSE*; otherwise it returns *TRUE*.

I use the internal tree notation described by Knuth [Reference 1, section 2.3.3] for a triply-linked tree. Each node consists of three pointers, **parent*, **offspring*, and **rightsibling*, and its information in field **info*. Thus, if node *T* is the root of a binary tree, the root of its left subtree is

```
T->offspring
```

and the root of its right subtree is

```
T->offspring->rightsibling
```

**Acknowledgements**

I appreciate the written correspondence I received from the experts on this topic (listed alphabetically): Brad A. Myers at the Computer Systems Research Institute in Toronto, Andy Poggio at SRI International, Edward Reingold at the University of Illinois, Bob Tarjan at AT&T Bell Laboratories, and C.S. Wetherell at AT&T Information Systems. Thanks also to Jane Munn, Jim Staton, Bob Gibson, John Broughton III, Steve Joyce, and Dr. Bill Wright at IBM. The bulk of this article first appeared in Software — Practice and Experience, July 1990, Copyright 1990 by John Wiley and Sons, Ltd. Reprinted by permisison.

**References**

[1] Knuth, D.E., The *Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1973.

[2] Radack, G.M., "Tidy Drawing of M-ary Trees," Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, Report CES-88-24, November 1988.

[3] Reingold, E.M. and J.S. Tilford., "Tidier Drawings of Trees," *IEEE Transactions on Software Engineering* SE-7, 2, March 1981, 223-228.

[4] Supowit, K.J. and E.M. Reingold, "The complexity of drawing trees nicely," *Acta Informatica* 18, 4, January 1983, 377-392.

[5] Sweet, R.E., "Empirical estimates of program entropy." Department of Computer Science, Stanford University, Stanford, CA, Report STAN-CS-78-698, November 1978. Also issued as Report CSL-78-3, Xerox PARC, Palo Alto, CA, September 1978.

[6] Tilford, J.S., "Tree drawing algorithms." M.S. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, Report UIUCDCS-R-81-1055, April 1981. Available as document UILU-ENG-81-1711 from the College of Engineering Document Center at the Univ. of Illinois.

[7] Walker II, J. Q., "A Node-Positioning Algorithm for General Trees," *Software — Practice and Experience* 10, 1980 553-561.

[8] Wetherell, C.S. and A. Shannon, "Tidy Drawings of Trees," *IEEE Transactions on Software Engineering*, SE-5, 5 September 1979, 514-520.

**Listing 1**