LINEAR TIME ALGORITHMS FOR LAYOUT OF GENERALIZED TREES

Ari Juutistenaho

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1994-6

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1994-6, MARCH 1994

LINEAR TIME ALGORITHMS FOR LAYOUT OF GENERALIZED TREES

Ari Juutistenaho

University of Tampere
Department of Computer Science
P.O. Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-3528-1
ISSN 0783-6910

Linear Time Algorithms for Layout of Generalized Trees

ARI JUUTISTENAHO
Department of Computer Science, University of Tampere
email: ari@cs.uta.fi

ABSTRACT

Many layout problems involve *n*-ary trees with variable-sized vertices. This paper includes linear time algorithms for the layout of such trees. Two algorithms are presented: the first one creates a layout in a simple way, and the second one makes it narrower.

1. Introduction

In practice we often have to lay out trees that contain variable-sized vertices and have arbitrary arity. In many situations the shape of the tree varies dynamically, which requires that algorithms must be fast and efficient.

First we consider different ways to draw a tree, called conventions, to see what kind of result we are looking for. This paper presents a tree drawing algorithm, which is a simplified version of the algorithm of Eades et al. [ELL]. The algorithm constructs a layout of a generalized tree in linear time, but the layout is not as narrow as possible.

Bloesch [Bl] represents an algorithm that creates tree drawings that occupy as little width as possible, but his algorithm is not linear. Finally we consider a linear time algorithm that is based on Bloesch's algorithm and that takes as input the layout we got with the former algorithm and makes it narrower.

2. Conventions

Eades et al. [ELL] consider three tree drawing conventions: classical convention, tip-over convention and inclusion convention. In each convention the vertices are represented as boxes (rectangles) that do not overlap.

In the *classical convention* the parent - child relationships are represented by lines between the boxes, and the depth of a vertex is represented by its y coordinate, as in Figure 1.

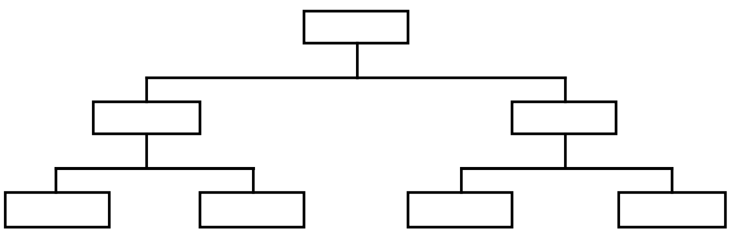

Figure 1: Classical Convention

The *tip-over convention* differs from the classical one by arranging the children of a vertex either vertically or horizontally. Each parent is still above its children. This is demonstrated in Figure 2.

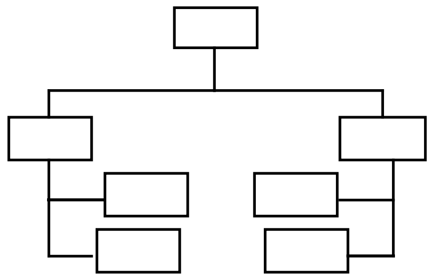

Figure 2: Tip-over Convention

In the *inclusion convention* the box of a child is inside the box of its parent, as in Figure 3.

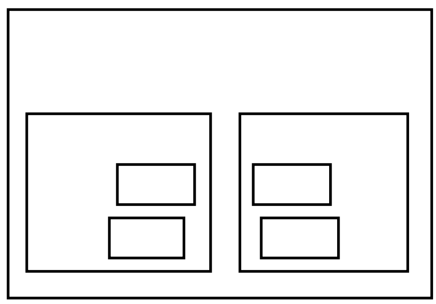

Figure 3: Inclusion Convention

The algorithm that draws a layout in the inclusion convention can be usable for drawing the tip-over convention or the classical convention, too, as we can see in Figures 2 and 3. This is possible if the algorithm fulfils certain conditions: the children must be located either vertically or horizontally; and there must be enough space for the parent, too.

3. New Drawing Algorithm

The algorithm we present here is based on the algorithm of Eades et al. [ELL]. Both algorithms handle vertices with different sizes, and the results can be used to draw trees in the inclusion convention although the main goal is to follow the tip-over convention. Eades's algorithm is for balanced binary trees and locates the children either vertically or horizontally, but our algorithm draws a layout of any kind of tree and locates the children horizontally.

As input the algorithm requires the minimum horizontal and vertical distances between the boxes (*dx* and *dy*), and the name, the width, the height, and the parent (except for the root) for each vertex to be considered.

Our method is divided into two parts: the first algorithm calculates the coordinates of the boxes, and the second one draws the layout.

When we construct the layout of a tree, we need the following data for each vertex (Example values concerning a sample tree in Figure 4 are given in parentheses).

w	width of the box ($\|d - r\|$, $\|d2 - r2\|$)
e	height of the box ($\|l - d\|$, $\|l2 - d2\|$)
a	width of the subtree ($\|c - a\|$, $\|c2 - a2\|$)
z	height of the subtree ($\|z - a\|$, $\|z2 - a2\|$)
RelX	the horizontal distance of the left-top corner of the subtree from the left-top corner of the subtree of the parent ($\|c2 - b\|$ for the vertex B; RelX = 0 for the vertex A)
RelY	the vertical distance of the left-top corner of the subtree from the left-top corner of the subtree of the parent ($\|c2 - c\|$)
Xl	the horizontal distance of the left-top corner of the box from the left-top corner of the subtree ($\|c2 - l2\|$, $\|c - l\|$).

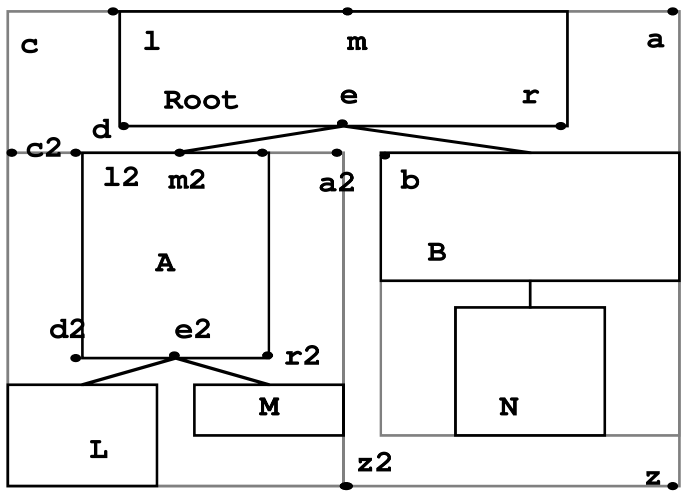

Figure 4: Layout of a tree

When those values are calculated, we need the following fields, too:

g	count of the children
h	shows if the values are calculated, initialized 0
EL	list of the edges to the children.

We get the values of w and e for each vertex as input. In Algorithm 1, the other values are calculated (Appendix 1 contains a C version of the algorithm). When we know the values a and z for each child, we can calculate them for the parent. If the parent is narrower than the subtree (like the vertex A in Figure 4), the parent must be centred; otherwise (like the vertex B in Figure 4) the children must be located so that they are in the centre of the area. The vertices are in the output according to their order in the input.

Algorithm 1. Calculating the coordinates of the boxes of the vertices.

Input: A list VL of vertices with g, h, El, w and e given.
Output: A list VL of vertices with a, z, RelX, RelY and Xl updated.

CountTree(VL: vertex list):
Assign unseen ← count of the vertices and v ← first vertex of VL.
Repeat
    If v is not handled **then**
        CalcSubtree(v, Unseen).
    Assign w ← v and v ← next vertex of the VL.
Until (Unseen = 0).
The vertex w is the root of the tree.

CalcSubtree(v: vertex; unseen: integer):
If v is a leaf **then**
    Assign v→a ← v→w and v→z ← v→e.
Else
    Assign Sx ← -dx; Sy ← dy + v→e;
    Deepest ← 0; R_most ← 0; and L_most ← INT_MAX.
    Handle all the children:
        Assign Sx ← Sx + dx; child→RelX ← Sx and child→RelY ← Sy.
        If child has not been handled **then**
            CalcSubtree(u, unseen).
        Assign Sx ← Sx + child→a.
        Update Deepest, L_most and R_most.

    Assign v→z ← Sy + maxy.
    If (Sx >= V→w) **then**
        Assign v→a ← Sx and v→Xl ← (R_most + L_most - v→w)/2.
    Else
        Assign v→a ← v→w.
        Handle all the children:
            Add (V→w - Sx)/2 to child→RelX.
Assign unseen ← unseen - 1 and mark the vertex handled.

Lemma 1. If T is a tree where u and w are children of v, then Algorithm 1 calculates in linear time, for all the vertices of T, coordinates with the following properties:

1) The boxes of the vertices do not overlap.
2) The left-top corners of the boxes of u and w have the same y-coordinate that is larger than the y-coordinate of the left-down corner of the box of v.
3) If the x-coordinate of the right-top corner of the box of u is smaller than the x-coordinate of the left-top corner of the box of v, then the x-coordinate of the right-top corner of the box of each vertex of the subtree u is smaller

than the x-coordinate of the left-top corner of the box of each vertex of the subtree *v*.

Proof. The property 1 is true if the properties 2 and 3 are true. Because *Sy* is assigned only once, then *u→RelY* = *w→RelY* = *Sy*. Because *u→RelY* represents the relative position of *u* to *v*, it is enough to prove that *Sy* > 0, which is obvious since *dy* > 0 and *v→e* > 0.

If *u* is handled before *w*, then *w→RelX* = *u→RelX* + *dx* + *u→a*. The property 3 follows from the fact that *u→a* represents the width of the subtree of *u*.

CountTree visits each vertex at most once. CalcSubtree(*v*) makes a constant number of calculations for the vertex *v* and for each descendant of *v*. □

After Algorithm 1 has updated the fields of the vertices, we can easily draw the layout of the tree, which is done by Algorithm 2 (Appendix 2 contains a C version of the algorithm).

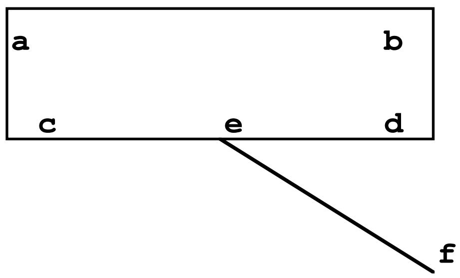

Figure 5: A sample drawing

We use here three drawing procedures: DrawBox, MoveTo, LineTo. We can demonstrate with Figure 5 how to use those procedures. If *l* and *t* are the x- and y-coordinates of the point *a*, and if *w* and *h* are the width and height of the box *abcd*, then DrawBox(*l, t, w, h*) draws that box. The line *ef* can be drawn by the procedure calls MoveTo(e_x, e_y), LineTo(f_x, f_y).

Algorithm 2. Drawing the layout of the tree.

Input: A list v of vertices updated by Algorithm 1.
Output: A layout of the tree v.

DrawTree(v: the root of the tree):
MoveTo(v→Xl + (v→w)/2, 0).
DrawSubtree(v, 0, 0).
DrawSubtree(v: vertex; px, py: integer):
Assign x2 ← px + v→RelX; px ← px + v→RelX + v→Xl and
py ← py + v→RelY.
LineTo(px + (v→w)/2, py); DrawBox(px, py, v→w, v→e).
Handle all the children:
    MoveTo(px + (v→w)/2, py + v→e).
    DrawSubtree(child , x2, py).

Lemma 2. Algorithm 2 draws a layout in the tip-over convention in linear time.

Proof. DrawSubtree calculates the absolute coordinates of a vertex using its relative coordinates and the absolute coordinates of its parent. Because the procedure draws the box according to the absolute coordinates, the box is located in the right place. MoveTo begins the edge in the centre of the bottom of the parent, and LineTo ends it in the centre of the top of the child. □

In Figure 6 we see a tree layout drawn by Algorithm 1 and Algorithm 2. Bloesch presents two algorithms that create tree drawings that occupy as little width and height as possible, but at the cost of O(nh) time, where the tree is h rasters high. In the following section we present an algorithm that makes the layout of Algorithm 1 narrower using only linear time.

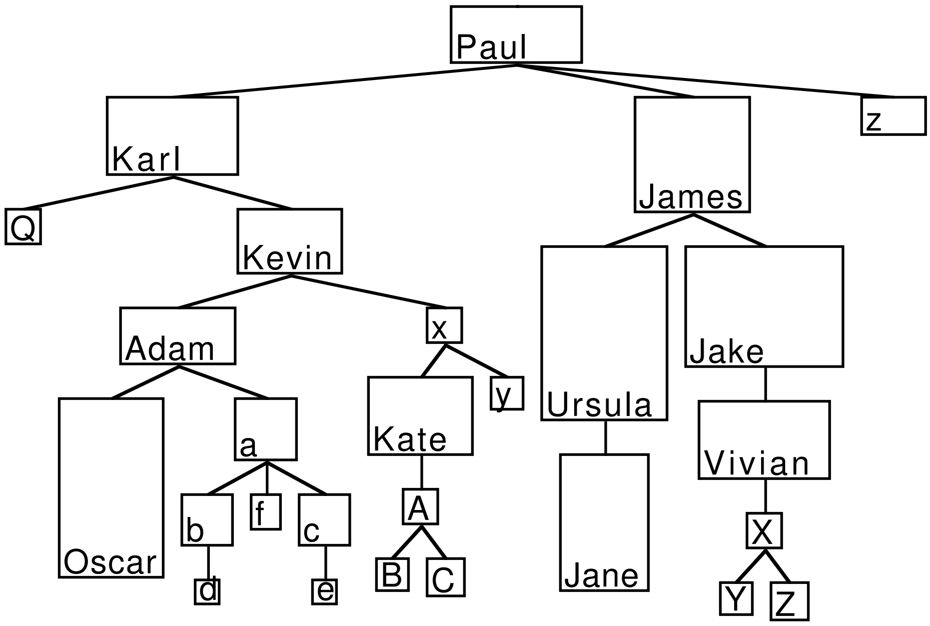

Figure 6 :Tree layout

4. Making the Layout Narrower

When we begin with Bloesch's first algorithm and make some changes to it, we can construct Algorithm 3 (Appendix 3 contains a C version of the algorithm) that uses the result of Algorithm 1 and comes to the same result as Bloesch's first algorithm [Bl, p. 821], but in linear time. The major disadvantage of Bloesch's first algorithm is that it compares the vertices raster by raster. In what follows we show that it is possible to use our knowledge of the sizes of the vertices. To be able to use that knowledge, we must construct, for each subtree, a path of the leftmost descendants and a path of the rightmost descendants.

Suppose that w is the first unhandled child of v and w is not the first child of v. The first path (R_v), called the *right path* of v, contains the rightmost handled vertices of the subtree of v, and the second one (L_w), called the *left path* of v, contains the leftmost vertices of the subtree of w. (Examples of paths are given in Figure 7.) After having compared the paths, we know (mindist(R_v, L_w)), the minimum distance between them. We can move the subtree of w to the left so that the new minimum distance is dx, the earlier defined minimum distance between the boxes. That can be done in the following way:

$$w \rightarrow RelX = w \rightarrow RelX - mindist(R_V, L_W) + dx$$

In the case of the first child, no movements of subtrees are done. Besides, in this case there is no right path of the parent we could compare the left path of the child with. We only update the paths of the parent. This happens e.g. in Figure 7 when we handle the vertices *a, t, f, c,* and *e*.

Algorithm 3 is recursive. If the vertex has children, we study their paths in order to create the paths of the parent. For that purpose, we must update the paths L_v and R_v each time we handle a child of v. Notice that the paths can have vertices of many subtrees, and L_v is not always the left path of the first child, nor is R_v always the right path of the last child.

Define that the depth of the path is the absolute location of the bottom of the last vertex of the path. There are four kinds of updating situations: the vertex is the eldest child, the right path is deeper, the left path is deeper, or the paths are of equal depth. Notice that depth(R_v) = depth(L_v) because R_v contains the rightmost vertices of subtree v, and L_v contains the leftmost ones of subtree v. The paths have often the same last vertex.

If depth(R_v) = depth(L_w), then L_v does not change and R_w including v becomes the new R_v. This is demonstrated in Figure 7 (R_v = { v, u, b, c }, L_v = { v, t, a }, L_w = { w, e, d }, R_w = { w, d }, and the new R_v = { v, w, d }). Only after this kind of comparison, the paths L_v and R_v do not have the same last vertex.

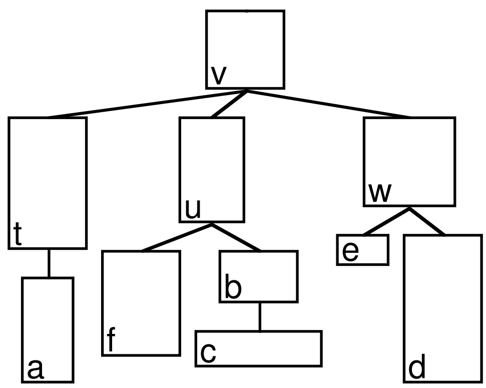

Figure 7: depth(R_v) = depth(L_w)

If depth(R_v) > depth(L_w), then L_v does not change and the new R_v contains R_w and those vertices of R_v whose depths are greater than the depth of R_w. This is demonstrated in Figure 8, where originally R_v = { v, u, b, c, a }, and after updating R_v = { v, w, d, c, a }.

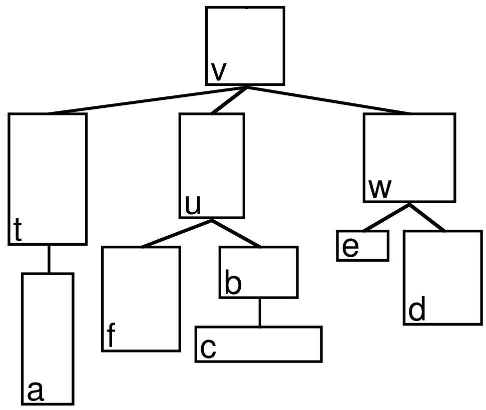

Figure 8: depth(R_v) > depth(L_w)

If depth(R_v) < depth(L_w), then R_w including v becomes the new R_v and the new L_v contains the old L_v and those vertices of L_w whose depths are greater than the depth of L_v. This is demonstrated in Figure 9, where originally L_v = { v, t, a }, and after updating L_v = { v, t, a, d }.

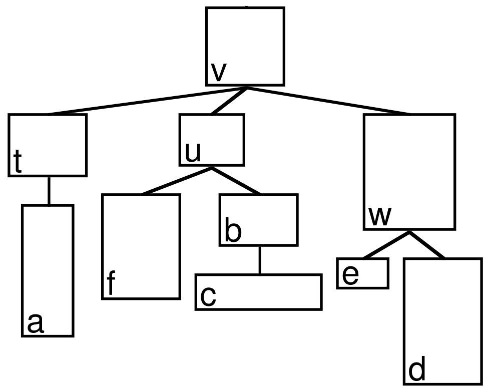

Figure 9: depth(R_v) < depth(L_w)

It is possible to make the layout narrower in Figure 7 and Figure 8, but not in Figure 9. The approach sometimes causes some problems: the boxes can temporarily overlap, or be located outside the drawing area. Those problems are demonstrated in Figure 10.

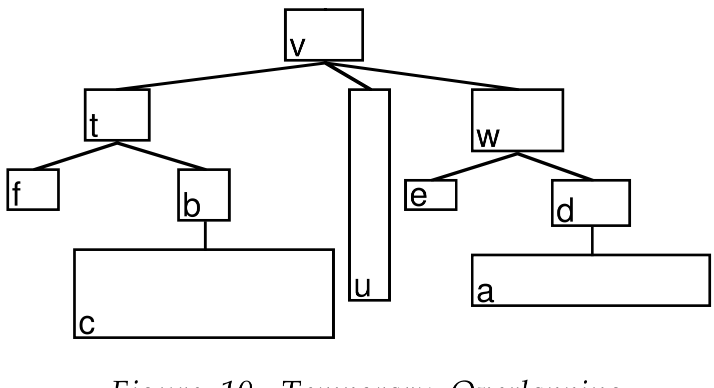

Figure 10: Temporary Overlapping

When the paths L_d and R_w are compared, the subtree d is moved to the left so that the distance between vertices e and d is minimal; but the vertices a and u are overlapping until the paths L_w and R_v are compared. Because mindist(R_v, L_w) is negative, the subtree w is then moved to the right so that the distances are large enough.

A more serious conflict occurs with the vertex c. When the paths L_b and R_t are compared, a part of the box of vertex c is located outside the drawing area. Therefore in the end of the algorithm, the items of the path L_v are compared to the left border of the drawing area, and the whole tree must be moved to the right, if necessary.

To keep the time complexity linear, we can go through the whole path only before we delete it. To avoid extra visits, we need certain pointers. For example, when we add the vertex d to L_v in Figure 9, there must be a pointer to the last item of L_v, which is updated to point to d. For the same reason, when we move a subtree, we update only the location of the path item of the root. The movements of the items of the paths will be updated when the items are compared.

Algorithm 3 needs the absolute coordinates of left-top corners of the vertices. Because each vertex is in a right path and in a left path, we need a different data structure for a item of a path. Let "path" be a pointer to a record with the following fields:

Next	pointer to the next path
Vx	pointer to the record of the vertex
downY	Y-coordinate of the bottom
rightX	X-coordinate of the right border
leftX	X-coordinate of the left border
Moved	shows how much the subtree of the vertex has been moved

Algorithm 3. Making the layout narrower using paths.

Input: A list v of vertices updated by Algorithm 1.
Output: A list v of vertices with locations updated.

NarrowTree(v: the root of the tree):
Assign RightPath ← NewPath(v) and LeftPath ← NewPath(v).
Point LastRight to RightPath and LastLeft to LeftPath.
Assign MaxDepth ← v→z + dy.
NarrowSubtree(RightPath, LastRight, LeftPath, LastLeft, v, MaxDepth, 1).
Move v so much to the right that vertices are inside the area.
DelPath(RightPath, NIL); DelPath(LeftPath, NIL).

NarrowSubtree(RightPath, LastRight, LeftPath, LastLeft: Path; v: Vertex; MaxDepth: Integer; Card: Integer):
Assign OwnRight ← NewPath(v) and OwnLeft ← NewPath(v).
Point OwnLastR to OwnRight and OwnLastL to OwnLeft.
Assign OwnDepth ← OwnRight→downY.

```
If v is not a leaf then
        Assign Count ← 0.
        Handle all the children of v:
                Assign Count ← Count + 1.
                NarrowSubtree(OwnRight, OwnLastR, OwnLeft, OwnLastL,
                                child, OwnDepth, Count).
        Move v to the middle of its children and get Moved.
        Add Moved to v→a.
        If (v→a < v→w) then
                Assign v→a ← v→w.
        Else
                Add Moved to  v→Xl, OwnRight→Moved, OwnLeft→Moved.
                Subtract Moved from OwnRight→Next→Moved and
                                        OwnLeft→Next→Moved.

Assign OwnLeftFUd ← OwnLeft and FirstUndel ← RightPath→Next.
If (Card > 1) then
        Movement ← ComparePaths(FirstUndel, OwnLeftFUd).
        Add Movement to OwnRight→Moved, OwnLeft→Moved, v→RelX.

Assign FirstDel ← RightPath→Next and RightPath→Next ← OwnRight.

If (Card = 1) then
        Update RightPath, LastRight, LeftPath, LastLeft and MaxDepth.
Elsif (OwnDepth = MaxDepth) then
        Assign LastRight ← OwnLastR.
Elsif (OwnDepth < MaxDepth) then
        Subtract Movement from FirstUndel→Moved.
        Assign OwnLastR→Next ← FirstUndel.
Elsif (OwnDepth > MaxDepth) then
        Assign MaxDepth ← OwnDepth; LastLeft→Next ← OwnLeftFUd;
        LastRight ← OwnLastR; LastLeft ← OwnLastL.

DelPath(FirstDel, FirstUndel); DelPath(OwnLeft, OwnLeftFUd).
```

ComparePaths(Handled, Unhandled: path): integer:

```
Assign LeftSum, RightS ← 0.
Assign MinDist ← INT_MAX.
While (Handled <> NIL) and (Unhandled <> NIL) do
        Assign LeftB ← Handled→rightX + Handled→Moved + LeftSum.
        Assign RightB ← Unhandled→leftX + Unhandled→Moved + RightS.
        If (Handled→downY < Unhandled→downY) then
                Add LeftSum to Handled→Moved.
                Assign LeftSum ← Handled→Moved.
                Assign Handled ← Handled→Next.
```

```
        Elsif (Handled→downY > Unhandled→downY) then
                Add RightS to Unhandled→Moved.
                Assign RightS ← Unhandled→Moved.
                Assign Unhandled ← Unhandled→Next.
        Else  do both former if and former elsif.

        MinDist ← Min(MinDist, RightB - LeftB).

If (Handled <> NIL) then
        Add LeftSum to Handled→Moved.
If (Unhandled <> NIL) then
        Add RightS to Unhandled→Moved.
Return( -MinDist).

NewPath(v: vertex): p: path:
Reserve space for p.
Assign p→Vx ← v; p→downY ← v→CY + v→e + dy;
p→leftX ← v→CX; p→rightX ← v→CX + v→w + dx;
p→Moved ← 0; p→Next ← NIL.
Return(p).

DelPath( FirstDel, FirstUndel: path):
While ((FirstDel <> FirstUndel) and (FirstDel <> NIL)) do
        Assign Del ← FirstDel.
        Assign FirstDel ← FirstDel→Next.
        Free the space of Del.
```

Lemma 3. Algorithm 3 works correctly. Its time complexity is linear.

Proof. In the procedure NarrowSubtree, the vertex is moved if its rightmost child has been moved. Because the relative locations of the children to their parent change, the children must be updated, too. The procedure ComparePaths updates the location of a vertex before the comparison, goes through the paths until either of them ends, and finds the minimum distance between them. Finally, the last visited item is updated, too.

The procedure ComparePaths returns how much the vertex can be moved. If OwnDepth < MaxDepth, then the result must be subtracted from the first old item of the right path so that the following items could be correctly updated. In the end of the procedure NarrowTree, the relative coordinates of the vertices are correct, but the absolute coordinates are not updated. That can be done later in linear time.

Each vertex is visited once when we create the paths. During that visit, we create two items of path for each vertex. If n is the number of vertices, we show that the total number of visits of the items on the paths is O(n).

Each item of the paths is handled once when it is created and once when it is deleted. Each item is updated at most once when its children are handled, at most once when its parent is recentred, and at most once when the paths have been compared.

The procedure ComparePaths(R_v, L_w) handle the paths until either of them ends. If those paths have k items, the number of comparisons $\leq k$. After the comparison, all the handled items of the paths are deleted, except one that can be compared later, the item "FirstUndel" in R_v if depth(R_v) > depth(L_w), or the item "OwnLeftFUd" in L_v if depth(R_v) < depth(L_w). So the items of paths are usually visited once in comparison. For each child there can be one item that has already been visited and will be able to be visited later. So there can be $3n$ comparisons for n vertices. □

If Algorithm 2 gets as input the results of Algorithm 3, it draws a layout that is often narrower than the layout drawn with the results of Algorithm 3. In Figure 11 we have a narrower layout for the tree shown in Figure 6. In the following section we list some aesthetics and look how well our layouts meet those requirements.

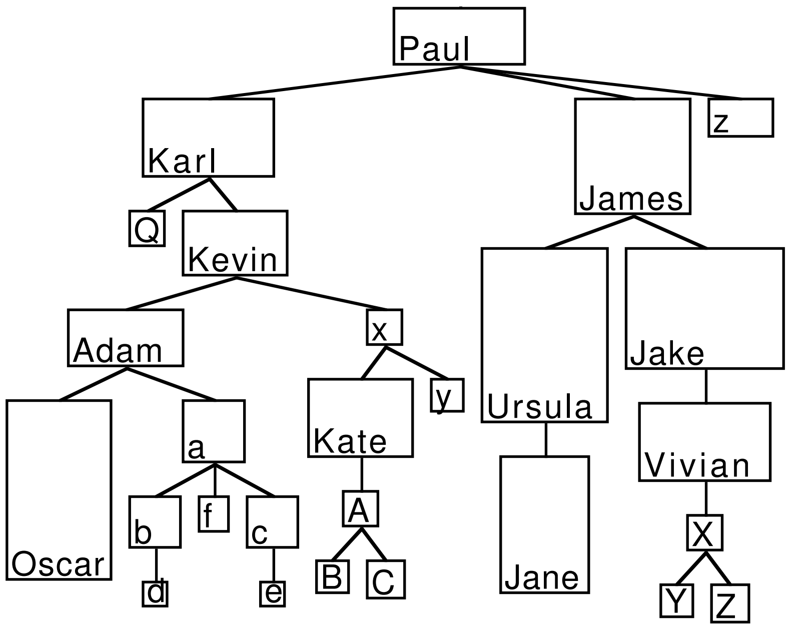

Figure 11: Narrower tree layout

5. Bloesch's criteria

Bloesch [Bl] lists the following 7 aesthetics for a tree layout:

1. Sibling vertices should have their top edges aligned horizontally.

2. Sibling vertices should be drawn in the same left-to-right order as their logical order.

3. Parent vertices should be centred over the centre of their leftmost and rightmost children.

4. A tree and its logical mirror image should be drawn as reflections of each other, and a subtree should be drawn in the same way no matter where it appears in a tree.

5. No edge joining the centre of the bottom of a vertex with the centre of the top of a child should cross any other such edge or vertex.

6. All vertices that share a raster should be separated horizontally by at least a distance $p > 0$.

7. Each vertex should be separated vertically from its parent by exactly a distance q. If vertices are composed of lines of text on a bit-mapped display, then q should be a multiple of the line height.

Bloesch's algorithm creates a layout that meets all requirements 1 - 7. Algorithm 1 and Algorithm 2 create layouts that meet all of those criteria except 3. The vertex is centred over the centre of the left border of its leftmost child and the right border of its rightmost child, which location is quite near to the centre of the centre of its leftmost and the centre of its rightmost children.

This narrower layout meet the same Bloesch's criteria as the wider layout, except 4, since the narrower layout of a symmetrical tree is not always symmetrical.

References

[Bl] A. Bloesch, Aesthetic layout of generalized trees, *Software - Practice and Experience*, **23**, 817 - 827 (1993).

[ELL] P. Eades, T. Lin and X. Lin, Two tree drawing conventions, *International Journal of Computational Geometry & Applications*, **Vol. 3, No. 2**, 133 - 153 (1993).

APPENDIX 1: C CODE FOR ALGORITHM 1

```
/* These appedices differ in some situations
    from the algorithms they are based on */

/* CalcSubtree calculates the demand of space for the subtree
   and updates the count of the unhandled vertices.
   It goes the vertices of the subtree in the depth-first order.
*/
void CalcSubtree( V, Unseen )
struct vertex *V;
int  *Unseen;
{
     edgenode  *Edge_1,  /*   pointer to the beginning */
               *Edge_i;       /*   of the    list */

     vertexnode      *u;
     int       Sx, Sy, S, maxy = 0;
     int       leftest = INT_MAX, rightest = 0;
```

```
	if (V->g == 0) {  /* a leaf  */
		V->a = V->w;
		V->z = V->e;
	} else {
		Edge_1 = Edge_i = V->EdgeList;
		Sx = -dx; Sy = dy + V->e;
		do {
			if (!(Edge_i->EndVertexChild)) { /* edge to */
				Sx += dx;                 /* child */
				u = Edge_i->ToVertex;
				u->RelX = Sx;
				u->RelY = Sy;
				if (u->h > -1)
					CalcSubtree(u, Unseen);
				Sx += u->a;
				maxy = ( u->z > maxy) ? u->z : maxy
				if ( leftest > (u->RelX + u->Xl))
					leftest = u->RelX + u->Xl;
				if ( rightest < (u->RelX + u->Xl + u->w))
					rightest = u->RelX + u->Xl + u->w;
			} /* else edge to V's parent */
		Edge_i = Edge_i->Next;
		} while (Edge_i != Edge_1);
		V->z = Sy + maxy;
		if (Sx >= V->w)     {
			V->a = Sx;
			V->Xl = (rightest + leftest - V->w) / 2;
		} else {
		/*   The width of the parent is larger than
		the one of the children. */
		/*   So the children are located
		in the middle of the area */
			Edge_1 = V->EdgeList;
			Edge_i = Edge_1;
			do {
				if (!(Edge_i->EndVertexChild)) { /* edge to
				                                  V's child */
					S = (V->w - Sx)/2;
					Edge_i->ToVertex->RelX += S;
				}
				Edge_i = Edge_i->Next;
			} while (Edge_i != Edge_1);
			V->a = V->w;
		}
	}
	(*Unseen)--;
	V->h--;             /* mark the vertex handled */
	MaxY = (V->z > MaxY) ? V->z : MaxY;
	MaxX = (V->a > MaxX) ? V->a : MaxX;
}


/* CountTree calculates the layout according to Algorithm 1  */
boolean CountTree( VertexList )
vertexnode **VertexList;

{	vertexnode *V, *FirstV, *Root;
	int *UnseenVs;
	V = *VertexList;
	*UnseenVs = VertexCount;
```

```
    /* Because the order of the vertices is not known,
      we must go through the whole tree.
      If the vertex has been handled, V->h = -1.
      Else the values of the whole subtree are calculated: */
    do {
         if (V->h > -1)
              CalcSubtree(V, UnseenVs);
              V = V->Next;
    } while (*UnseenVs > 0);
    /* The values of the root remain zeros */

    *VertexList = V->Prev;   /* Let the VertexList point */
    V = V->Prev;        /* to the root */
}
```

APPENDIX 2: C CODE FOR ALGORITHM 2

```
static void DrawSubtree(Root, px, py)
vertexnode *Root;
int px, py;
{
     vertexnode *v;
     edgenode   *e, *e1;
     int x2;

     v = Root;
     px += v->RelX;
     py += v->RelY;
     x2 = px;
     px += v->Xl;
     LineTo( px + (v->w)/2, py );  /* edge from the parent */
     DrawBox( px, py, v->w, v->e );

     if (v->Name)
          DrawTrName( px, py, v->e, v->Name);
     if (Root->g > 0) {
          e = e1 = v->EdgeList;
          do {
               if (!(e->EndVertexChild)) {
                    MoveTo( px + (v->w)/2, py + v->e );
                    /* edge to the child */
                    DrawSubtree(e->ToVertex, x2, py);
               }
               e = e->Next;
          } while (e != e1);
     }
}

static void HandleTreeVertices(VertexList)
vertexnode *VertexList;
{
     vertexnode *v;
     v = VertexList;
     MoveTo( v->CX + v->Xl +(v->w)/2, 0 );
     DrawSubtree(v, 0, 0);
}
```

APPENDIX 4: C CODE FOR ALGORITHM 3

```
/* Paths goes through the tree according to Algorithm 3 */
void Paths( tree )
vertexnode *tree;
{
     pathnode  *lp, *rp, *ll, *lr;
     int       MD;

     ll =lp  = NewPath(v);
     lr =rp = NewPath(v);
     MD = tree->z + dy;
     NarrowSubtree(rp, &lr, lp, &ll, tree, &MD, 1);
     ll = lr = (pathnode *)NULL;
     DeletePath( rp, lr); DeletePath( lp, ll);
}


/* NarrowSubtree makes the subtree narrower */
void NarrowSubtree( RightPath, LastRight, LeftPath, LastLeft,
                    v, MaxDepth, Card)
pathnode  *RightPath, **LastRight; /* Right path of the par */
pathnode  *LeftPath, **LastLeft;   /* Left path of the parent */
vertexnode  *v;                    /* the root of the subtree */
int       *MaxDepth;               /* old maximal depth */
int       Card;                    /* = n for the n:th child */
{
     pathnode  *OwnRight, *OwnLeft, *OwnLastL, *OwnLastR;
     pathnode  *FirstDel, *FirstUndel, *OwnLeftFUd;
     vertexnode  *u;          /* child */
     edgenode  *e;
     int       OwnDepth, Movement = 0, count = 0;
     int       NewPlace, OldPlace, PlaceMoves;
     pathnode  *FirstChild, *LastChild;

     /* Own paths are made */
     OwnLastL = OwnLeft  = NewPath(v);
     OwnLastR = OwnRight = NewPath(v);
     OwnDepth = OwnRight->downY;

     /* The subtrees of the children are handled: */
     if (v->g > 0) {
          e = v->EdgeList;
          do {
               if (e->EndVertexChild)
                    e = e->Next;
               count++; u = e->ToVertex;
               NarrowSubtree( OwnRight, &OwnLastR, OwnLeft,
                              &OwnLastL, u, &OwnDepth, count);
               e = e->Next;
          } while (count < v->g);
          OldPlace   = 2 *(OwnLeft->leftX + OwnLeft->Moved);
          FirstChild = OwnLeft->Next;
          LastChild  = OwnRight->Next;
          NewPlace   = (FirstChild->leftX + FirstChild->Moved +
                        LastChild->rightX + LastChild->Moved -
                        dx - v->w);
          PlaceMoves = (NewPlace - OldPlace)/2;
          v->a += PlaceMoves;
```

```
          if (v->a < v->w)
               v->a = v->w;
          else {
               v->Xl += PlaceMoves;
               OwnRight->Moved += PlaceMoves;
               OwnLeft->Moved += PlaceMoves;
               OwnRight->Next->Moved -= PlaceMoves;
               OwnLeft->Next->Moved -= PlaceMoves;
          }
     }

     /* If not 1st child, the paths are compared */
     if (Card > 1)
          Movement = Compare(RightPath->Next, OwnLeft);
     /*and the subtree is moved */
     OwnRight->Moved += Movement;
     OwnLeft->Moved += Movement;
     v->RelX += Movement;

     /* The paths are updated and unneeded items are deleted */
     FirstDel = RightPath->Next;
     RightPath->Next = OwnRight;
     OwnLeftFUd = (pathnode *)NULL;
     FirstUndel = (pathnode *)NULL;

     if (Card == 1) {
          *LastRight = OwnLastR;
          LeftPath->Next = OwnLeftFUd = OwnLeft;
          *LastLeft = OwnLastL;
          *MaxDepth = OwnDepth;
     } else if (OwnDepth == *MaxDepth) {
          *LastRight = OwnLastR;
     } else if (OwnDepth < *MaxDepth) {
          FirstUndel = FirstDel;
          while (FirstUndel->downY <= OwnDepth)
               FirstUndel = FirstUndel->Next;
          FirstUndel->Moved -= Movement;
          OwnLastR->Next = FirstUndel;
     } else if (OwnDepth > *MaxDepth) {
          OwnLeftFUd = OwnLeft;
          *LastRight = OwnLastR;
          while (OwnLeftFUd->downY <= *MaxDepth)
               OwnLeftFUd = OwnLeftFUd->Next;
          (*LastLeft)->Next = OwnLeftFUd;
          *LastLeft = OwnLastL;
          *MaxDepth = OwnDepth;
     }

     DeletePath( FirstDel, FirstUndel);
     DeletePath( OwnLeft, OwnLeftFUd);
}

/* Compare compares the paths h and unh and returns the value
how much the path unh can be moved. */
int Compare(h, unh)
pathnode  *h, *unh;
{    pathnode  *Handled, *Unhandled, *DeepItem;
     int       leftBorder, rightBorder;
     int       MinDist = INT_MAX, LeftSum = 0, RightSum = 0;
```

```
    Handled = h;   Unhandled = unh;

    while ((Handled != (pathnode *)NULL) &&
           (Unhandled != (pathnode *)NULL)) {
         leftBorder =   Handled->rightX + Handled->Moved +
                        LeftSum;
         rightBorder =  Unhandled->leftX + Unhandled->Moved +
                        RightSum;
         if (Handled->downY < Unhandled->downY) {
              Handled->Moved += LeftSum;
              LeftSum = Handled->Moved;
              Handled = Handled->Next;
         } else if (Handled->downY > Unhandled->downY) {
              Unhandled->Moved += RightSum;
              RightSum = Unhandled->Moved;
              Unhandled = Unhandled->Next;
         } else   {
              Handled->Moved += LeftSum;
              Unhandled->Moved += RightSum;
              LeftSum = Handled->Moved;
              RightSum = Unhandled->Moved;
              Handled = Handled->Next;
              Unhandled = Unhandled->Next;
         }
         if (MinDist > rightBorder - leftBorder)
              MinDist = rightBorder - leftBorder;
    }
    /* Update the last item: */
    if (Unhandled != (pathnode *)NULL)
         Unhandled->Moved += RightSum;
    if (Handled != (pathnode *)NULL)
         Handled->Moved += LeftSum;
    return( 0 - MinDist);
}

/* NewPath creates a new path item that points to V */
pathnode *NewPath( V )
vertexnode     *V;
{    pathnode    *pp;
     pp = (pathnode *) malloc( sizeof( pathnode ) );
     if (pp != (pathnode *)NULL) {
          pp->Next       = (pathnode *)NULL;
          pp->Vx         = V;      pp->Moved      = 0;
          pp->downY      = V->CY + V->e + dy;
          pp->leftX      = V->CX + V->Xl;
          pp->rightX     = V->CX + V->Xl + V->w + dx;
     }
     return (pp);
}

/* DeletePath deletes the path items FirstDel..(FirstUndel-1) */
void DeletePath( FirstDel, FirstUndel)
pathnode  *FirstDel, *FirstUndel;
{    pathnode  *Del;
     while (FirstDel != FirstUndel) {
          Del = FirstDel;
          FirstDel = FirstDel->Next;
          free( Del);
     }
}
```