An Incremental Editor for Dynamic Hierarchical Drawing of Trees

D. Workman,
University of Central Florida
School of EE and Computer Science
workman@cs.ucf.edu

M. Bernard
The University of the West Indies
Dept. Math. & Computer science
mbernard@fsa.uwi.tt

S. Pothoven
IBM Corporation

Abstract

We present an incremental tree editor based on algorithms for manipulating shape functions. The tree layout is hierarchical, left-to-right. The tree is laid out on a 2D-grid where parent and first child always occupy the same row on the grid and the parent is in the column to the immediate left of the child. This convention is recursive. Nodes of variable size and shape are supported. The paper presents algorithms for basic tree editing operations, including cut and paste. The layout algorithm for positioning child-subtrees rooted at a given parent is incrementally recomputed with each edit operation; it attempts to conserve the total display area allocated to child-subtrees while preserving the user's mental map. The runtime and space efficiency is good as a result of exploiting a specially designed Shape abstraction for encoding and manipulating the geometric boundaries of subtrees as monotonic step functions to determine their best placement. All tree operations, including loading, saving trees to files, and incremental cut and paste, are worst case O(N) in time, but typically cut and paste are $O(\log(N)^2)$, where N is the number of nodes.

1. Introduction

Effective techniques for displaying static graphs are well established but in many applications the data are dynamic and require special visualization techniques. Applications arise when the data are dynamically generated or where there is need to interact with and edit the drawing. Dynamic techniques are also used for displaying large graphs where the entire graph cannot fit on the screen. Subsets of the graph are processed incrementally and nodes are added dynamically to the considered set. A major issue with the drawing of dynamic graphs is the preservation of the user's mental map [8]. Insertions and deletions of nodes or subgraphs should not dramatically change the layout of the graph as such unpredictable changes disrupt the user's sense of context. Hence, an incremental approach is desirable.

In this paper, we present an incremental tree editor, DW-tree, for dynamic hierarchical display of trees. Graphs that are trees are found in many applications including Software Engineering and Program Design, which is the genesis of this Dynamic Workbench DW-tree software[14, 1, 12]. Trees have received a lot of attention in the Graph Drawing literature because of their ubiquity and simplicity. The Dynamic Workbench DW-tree is an interactive editor; it allows users to interact with the drawing, changing nodes and subtrees. In redrawing the tree after user changes, the system reuses global layout information and only localized subtree data need to be updated. A full complement of incremental editing operations is supported, including changing node shape and size as well as cutting and pasting subtrees. The editor has the capability of loading (saving) a tree from (to) an external file. The tree editor uses Shape vectors for defining the boundary of tree or subtree objects. Incremental tree editing operations are based on algorithms for manipulating Shape vectors. The tree layout is hierarchical, left-to-right (horizontal).

Two desirable properties of dynamic hierarchical graph drawings are consistency and stability [10]. A *consistent* drawing is one that adheres to a particular layout style. For example, in hierarchical, horizontal display of trees a parent node should always be drawn to the left of its children after some sequence of updates. A *stable* drawing is one in which small changes made to the graph do not cause large changes to the drawing. The DW-tree drawing algorithms construct tree drawings with these two properties. In addition, the algorithms attempt to conserve the total display area allocated to child-subtrees (area efficient) without appreciably distracting the user's mental continuity upon re-display. For a tree of size N nodes, the runtime efficiency is O(N) for load and save operations to an external file. For cut and paste operations at depth d, under reasonable assumptions, the runtime is $O(d^2)$, with $d = \log(N)$.

The remainder of the paper is organized as follows. First (Section 2), we present the principles for tree layout using the Shape abstraction. In section 3, we present key principles and algorithms that form the basis for the incremental editor operations and in Section 4 we discuss related work; we close with some concluding remarks in section 5.

2. Tree Layout Principles

In this section we present the layout design principles and definitions that provide the conceptual foundation and framework for remainder of the paper. The display area of the editor defines a 2D coordinate system as depicted in Figure 1.

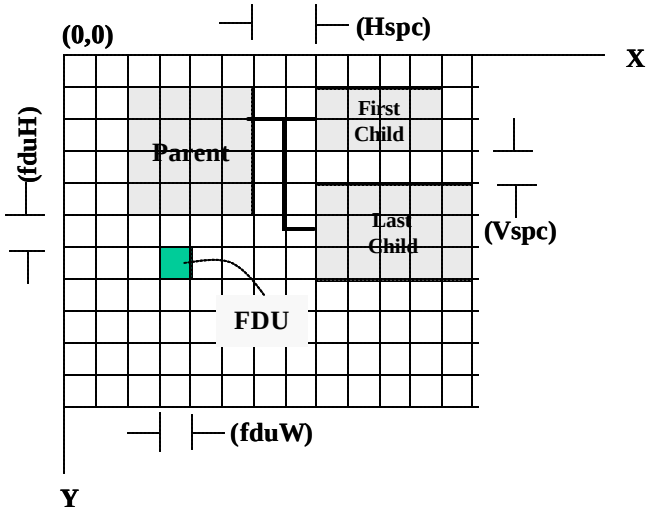

Figure 1. Display Coordinate System and Tree Layout.

Coordinate values along both axes are multiples of the width (*fduW*) and height (*fduH*) of the *Fundamental Display Unit* (*FDU*). The FDU size is measured in pixels and represents the smallest unit of display space allocated in the horizontal and vertical directions, respectively. Increasing the size of a FDU lowers space utilization for display objects, while decreasing its size increases the space utilization. Clearly, the smallest size of an FDU is one pixel and is ultimately limited by the dis play device resolution. The size of the FDU affects the amount of memory required to represent Shape vectors and also indirectly influences the runtime of various editor operations.

Nodes can be of any size, but are assumed to occupy a display region bounded by a rectangle having width and height that are multiples of fduW and fduH, respectively. The coordinates of a node always define the upper left corner of its bounding rectangle. *Trees* are oriented left-to-right as shown in Figure 1, where the Parent node and its First Child always have the same vertical displacement (Y coordinate). All children with the same parent node have the same relative horizontal displacement, *Hspc*, (and X coordinate). That is, if (x,y) denotes the FDU coordinates of a parent node, and the parent node has width (*Pwidth*) and height (*Pheight*), then the coordinates of child node, k, will always be ($x + Pwidth + Hspc$, $y + \Delta_k$), where for $1 \le k \le Nchildren$, $\Delta_1 = 0$, and for $k > 1$, $\Delta_k = \Delta_{(k-1)} + Cheight_{(k-1)} + Vspc$. *Vspc* is the minimum vertical separation between children. The actual vertical separation between children of the same parent is defined by our layout algorithms to conserve display area and will be presented in a later section entitled, Tree Operations. H denotes the vertical displacement of all children ($H = \Delta_{Nchildren} + Cheight_{Nchildren}$).

Edges are represented as polygonal lines in which the segments are either horizontal or vertical (orthogonal standard). An edge from parent to first child is a horizontal line. Edges from parent to other children have exactly two bends and consist of horizontal-vertical-horizontal segments. The edges from a parent to all its children overlap to form a 'trunk'. To allow space for edge drawings, each node is allocated an additional Hspc to its right in the horizontal direction. The rectangular region of width *Hspc* and height H, located at coordinates ($x + Pwidth$, y), is used by the parent node to draw the edges to all its children. Figure 2 illustrates a hierarchical tree drawn by the DW-tree software.

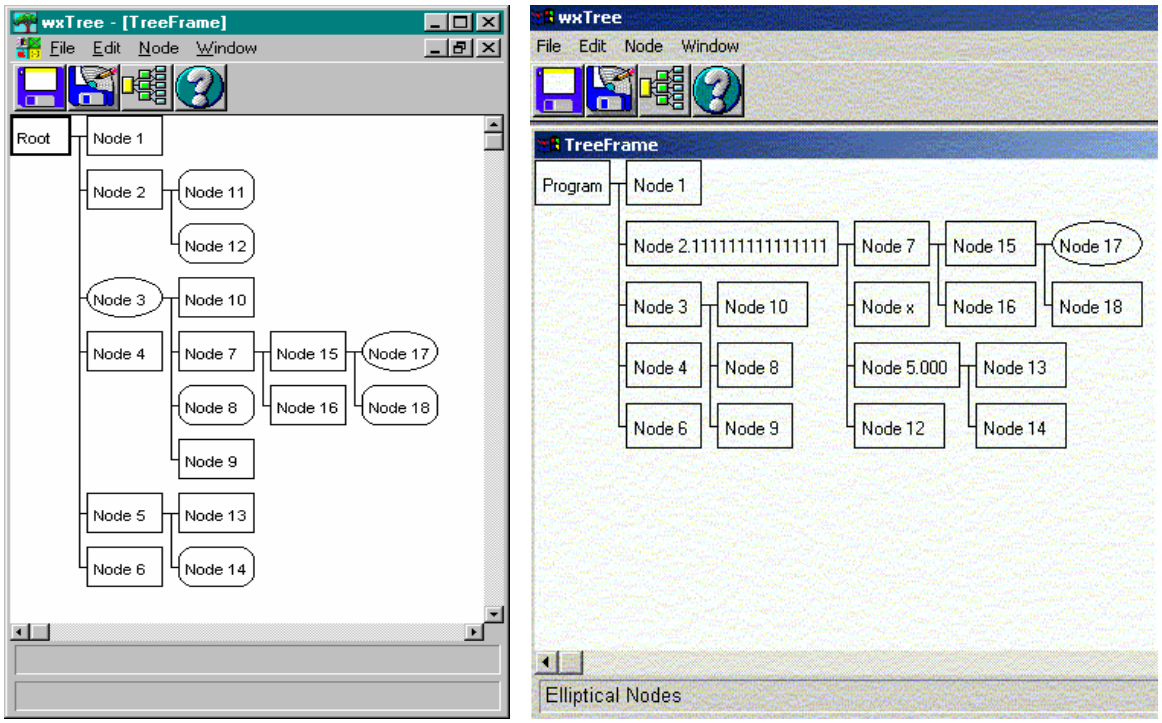

Figure 2. Examples of DW-Tree layouts

2.1 Shape Functions

Shape functions are objects that define the outline of a tree or subtree. Figure 3 illustrates the concept for some tree, T. The blue line outlines the upper shape (U_T) of T, the red line outlines the lower shape (L_T) of T. The subtree rooted at a given node has relative coordinates (0, 0) and corresponds to the coordinates of its root node. Every such subtree, T, has a bounding rectangle of width W_T and height H_T; in Figure 3, $W_T = 20$ and $H_T = 12$. The width of a subtree always takes into account the horizontal spacing (Hspc) associated with the node(s) having maximum extent in the X direction.

The upper shape function, U_T, is a step function of its independent variable, x. We therefore represent U_T as a vector of pairs: $U_T = <(\delta x_1, y_1), \ldots (\delta x_n, y_n)>$, where $\delta x_k = 1$ denotes the *length* of the k^{th} *step*, $1 = k = n$, and $y_1 = 0$. y_k gives the value of the function in the k^{th} step (interval). Along the X-axis the kth step defines a half-open interval $[x_{k-1}, x_k)$ where $x_0 = 0$ and $x_k = \sum_{j=1}^{k} dx_j$. The *width of T* then becomes $W_T = x_n$ – the right extreme of the last step interval of U_T. Finally, we define *steps*$(U_T) = \{x_0, x_1, \ldots, x_n\}$, the set of interval end-points defined by U_T, and *dom*$(U_T) = [x_0, W_T]$, the closed interval defined by the extreme X-coordinates of the minimal bounding rectangle for T.

The lower shape function, L_T, is defined in an analogous way with the roles of x and y reversed. That is, we represent L_T as a step function with y as the independent variable. Specifically, for some $m = 1$, $L_T = < (x_1, \delta y_1), \ldots (x_m, \delta y_m)>$. Along the Y-axis the kth step defines a half-open interval $[y_{k-1}, y_k)$ where $y_0 = 0$ and $y_k = \sum_{j=1}^{k} dy_j$. Thus the *height of T* becomes $H_T = y_m$ – the right extreme of the last step interval of L_T. And, we define steps$(L_T) = \{ y_0, y_1, \ldots, y_m \}$, the set of interval end-points defined by L_T, and dom$(L_T) = [y_0, H_T]$, the closed interval defined by the extreme Y-coordinates of the minimal bounding rectangle for T.

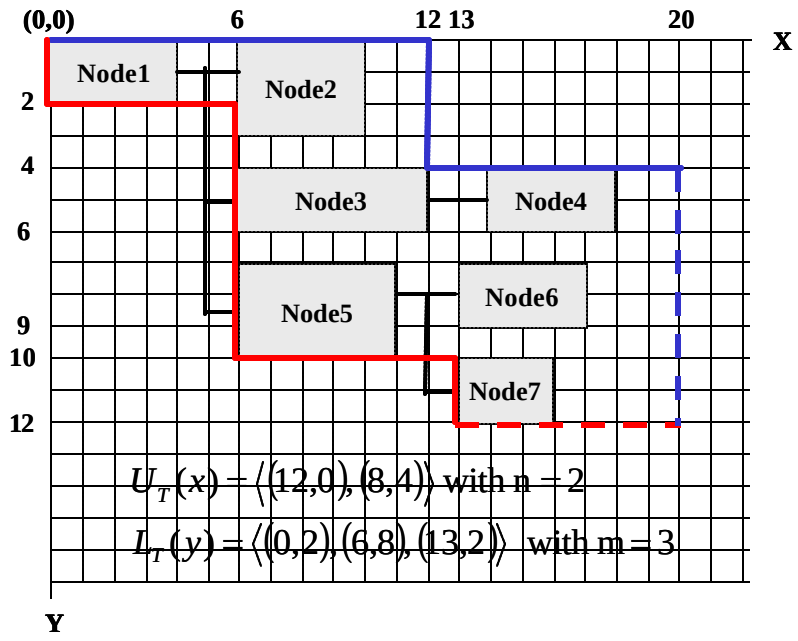

Figure 3. Shape Functions for Tree, T.

Because of our layout conventions, the key step in our layout algorithm requires computing the vertical separation of adjacent siblings in a given subtree. For example, to compute the separation of the subtrees T3 and T5 rooted at Node3 and Node5, respectively, in Figure 3, we must use the lower shape function of T3 and the upper shape function of T5. Since these shape functions are based on different independent variables, we must convert the lower shape function to an equivalent step function with x as the independent variable. We let Λ_T denote the lower shape function of T where x is the independent variable. $\Lambda_T = < (\delta x_1, y_1), \ldots (\delta x_m, y_m)>$, where $\delta x_k = (x_{k+1} - x_k)$ and y_k are computed from L_T as defined above. However, because x_{m+1} is not defined in L_T we compute $\delta x_m = (W_T - x_m)$. As we will show later, it will always be the case that $W_T > x_m$, where x_m is always taken from the last step interval of L_T.

2.2 The Shape Algebra

Incremental tree editing operations are based on algorithms for manipulating shape functions. Some of the most important of these algorithms will be presented in the next section. As the basis for these algorithms, we introduce a simple algebra for manipulating shape functions. There are essentially seven operations of significance. Each will be defined for upper shape functions – analogous definitions apply to lower shape functions (L_T and Λ_T).

UShape(dx ,y) = an upper step function = <(dx, y)>. Similarly, ***LShape(x, dy)*** = <(x, dy)>.
UShape and LShape are distinct *function types*.
Example. UShape(3, 4) = <(3,4)> ? LShape(3, 4) = <(3,4)>.

Max(R,S)= Z*, where R,S and Z are UShape functions. Assume that dom(R) Í dom(S). Then dom(Z) = dom(S) and steps(Z) = steps(R) È steps(S) for each x Î steps(Z), Z(x) = max(R(x),S(x)), if x Î dom(R) Ç dom(S); Z(x) = S(x), if x Î dom(S) – dom(R).*
Example. If R = <(2,0),(1,7),(1,12),(2,15)>, S = <(3,2),(1,4),(7,7)>, then steps(R) = {0,2,3,4,6}, dom(R) = [0,6], steps(S) = {0,3,4,11}, dom(S) = [0,11], steps(Z) = {0,2,3,4,6,11}, dom(Z) = dom(S), and Z = Max(R,S) = Max(S,R) = <(2,2),(1,7),(1,12),(2,15),(5,7)>.

Min(R,S) *is defined in an analogous fashion to Max.*
Example. If R = <(2,0),(1,7),(1,12),(2,15)>, S = <(3,2),(1,4),(7,7)>, then Z = Min(R,S) = Min(S,R) = <(2,0),(1,2),(1,4), (7,7)>. **Note:** the number of steps in Z can be reduced from *steps(R) È steps(S)* because two successive steps have the same y-value. In general, we assume that a Shape function is represented using the fewest number of steps.

Sum(R,S) *= Z, where R,S and Z are UShape functions. Dom(Z) = dom(R) Ç dom(S) and steps(Z) = { x | x Î (steps(R) È steps(S))) Ç dom(Z) }*
Example. If R = <(2,0),(1,7),(1,12),(2,15)>, S = <(3,2),(1,4),(7,7)>, then Z = Sum(R,S) = Sum(S,R) = <(2,2),(1,9),(1,16),(2,22)>.

Diff(R,S) *is defined analogously to Sum, but is not commutative.*
Example. If R = <(2,0),(1,7),(1,12),(2,15)>, S = <(3,2),(1,4),(7,7)>, then Z = Diff(R,S) = <(2,-2), (1,5), (3,8)> and Diff(S,R) = <(2,2),(1,-5),(3,-8)>. **Note:** The sign of the y-coordinates of each step in Diff(R,S) is reversed to obtain Diff(S,R).

ScalarAdd(c, R) = ScalarAdd(R,c) *= Z, where R and Z are UShape functions and c is a scalar constant. Z(x) = R(x)+c, for all x Î dom(Z) = dom(R). In effect, the scalar c is added to the y-component (dependent variable) of each step in a UShape, and added to the x-component of each step in an LShape. Note: steps(Z) = steps(R).*
Example. If R = <(2,0),(1,7),(1,12),(2,15)>, then ScalarAdd(R, 3) = ScalarAdd(3,R) = <(2,3),(1,10),(1,15),(2,18)>.

Cat(R,S) *= Z, where R, S and Z are UShape functions. steps(Z) = { x | x Î steps(R) or x = x'+W_R, for x' Î steps(S) } – {W_R | y-value of the last step of R = y-value of the first step of S}, and dom(Z) = [0, W_R+W_S]; Z(x) = R(x), if x Î dom(R); Z(x) = S(x'), if x = x'+ W_R, where x' Î dom(S).*
Example. If R = <(2,0),(1,7),(1,12),(2,15)>, S = <(3,2),(1,4),(7,7)>, then Z = <(2,0),(1,7),(1,12),(2,15),(3,2),(1,4),(7,7) >, steps(Z) = { 0, 2, 3, 4, 6, 9, 10, 17}, dom(Z) = [0,17].

MaxElt(R) *= C, where R is an UShape function. C = Max{ R(x) | x Î dom(R)} = Max{ R(x) | x Î steps(R)}.*
Example. If Z = <(2,0),(1,7),(1,12),(2,15),(3,2),(1,4),(7,7) >, of the previous example, then MaxElt(Z) = Z(6) = 15.

Runtime Note: For Max(R,S) and Min(R,S) the runtime is O(|R|+|S|), where |R| denotes the number of steps in R (analogously of S). For ScalarAdd(R), MaxElt(R) the runtime is O(|R|). For Cat(R, S) the runtime can be O(1) if linked structures are used to represent shape functions.

3.0 Tree Operations

Some of the most basic operations on trees, and the algorithms that implement them, are discussed in this section. These operations are:

(1) Create and edit a selected node.
(2) Cut and paste a subtree.
(3) Read (write)a tree from (to) an external file.
(4) Select a node in a tree.

(Op-1) *Creating/editing a node* involves defining/changing the size of the bounding rectangle enclosing the node's display image. This means (re-)computing the shape function for the node and then propagating these change throughout the rest of the tree in which the node is embedded. As this is a special case of cutting/pasting a subtree, we describe how shape functions are defined for a single node and defer the rest of the discussion to the subtree cutting/pasting operations. If R is a free-standing node with width (R.width) and height (R.height), then: U_R = UShape(R.width + Hspc, 0) and L_R = LShape(0, R.height). **Runtime Note:** this is an O(1) operation.

(Op-2) *Cutting/Pasting a node*. Let S and T be fully defined free-standing trees. We consider the operation of pasting S into T at some position. The position in T is a node previously identified through a *select operation* defined in detail below. Let P denote the selected node in T and let R = root(S) denote the root of the tree S. The paste operation requires a parameter that defines the new relationship P is to have with R in the composite tree, that is: R can be an upper/lower sibling of P (assuming P ≠ root(T)); R can be the (new)(only) first/last child of P. We describe the case where R is to become a new lower sibling of P. The other variations differ only in minor details. Figures 4(a) and 4(b) illustrate the trees S and T before and after the paste operation, respectively. The relevant shape functions are presented in the table below.

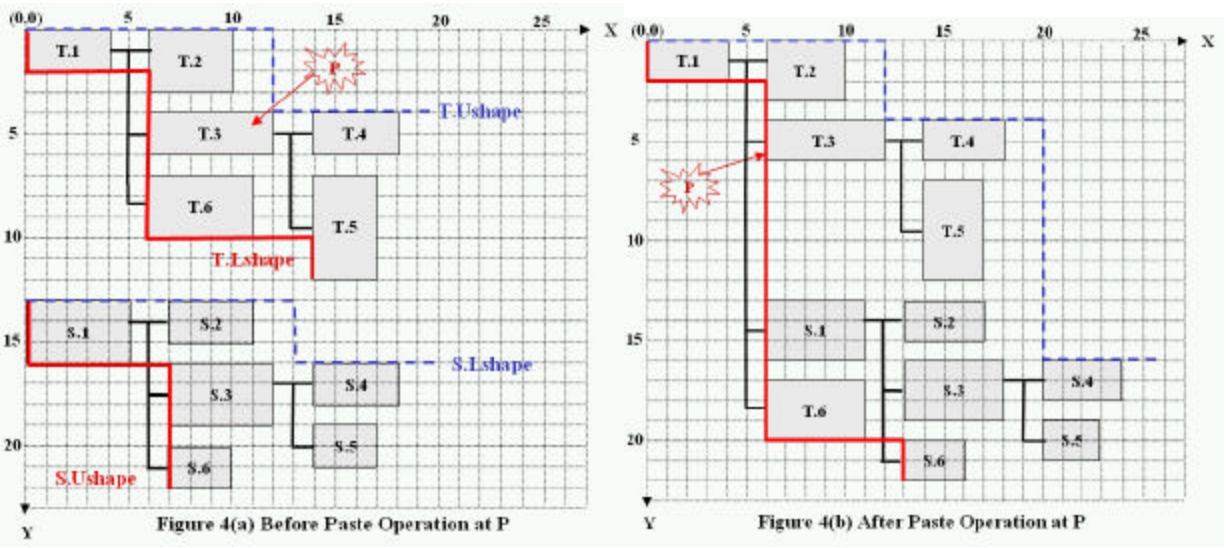

Figure 4. Subtree Pasting Operation

Tree (Figure 4)	Upper Shape (U)	Lower Shape (L)	Lower Shape (L)
Tree T before paste (T_B)	**< (12, 0), (8,4)>**	**< (0,2), (6,8), (14,2) >**	**< (6,2), (8,10), (6, 12) >**
Tree S before paste(S_B)	**< (13, 0), (7, 3)>**	**< (0,3), (7, 6) >**	**< (7,3), (13,9) >**
T after paste (T_A)	**< (12, 0), (8,4), (6, 16) >**	**< (0,2), (6, 18), (13, 2) >**	**< (6,2), (7, 20), (13, 22) >**

We now present the essential features of an algorithm for updating the shape functions resulting from a paste (or Cut) operation. There are three key sub-algorithms incorporated in the algorithm for a comp lete Cut or Paste operation.

Algorithm 1 (Change Propagation): If P is the parent of some node (child subtree) whose shape changes, then the shape functions for P (for the subtree rooted at P) must be recomputed (See Algorithms 2 and 3). No additional change is necessary to the shape functions of any other child of P. Once the shape functions for P have been recomputed, then this algorithm is repeated at the parent of P, etc., terminating only when the root of the tree is reached. **Runtime Note:** Clearly the worst case running time is O(N), where N is the number of nodes in a tree. The worst case would occur with trees where every node has a single child. However, if we assume a randomly chosen tree with N nodes, then the tree will tend to be balanced – each node has approximately the same number of children, say b. Thus Change Propagation will require O(dp) running time, where d denotes the depth in the tree where the first change occurs ($d \approx \log_b(N)$) and p denotes the worst case running time at each level on a path to the root. The running time at each level is essentially the running time of Algorithm 2 and 3 combined.

Algorithm 2(Child Placement): The algorithm for (re-)positioning the child subtrees of a given root or parent node when a new child subtree is added, removed, or changed can be characterized as building a ***forest of trees*** relative to some given origin point – the ***forest origin***. Figure 5 illustrates the Forest concept. In Figure 5a, a Forest is depicted with origin (X_F, Y_F) and an arbitrary (child) tree C_k with origin (X_k, Y_k). The normal shape functions for C_k have their origin relative to the origin of C_k. These functions are depicted as U_k and L_k. The first operation we need to perform is to extend the shape functions for C_k so that they are relative to the origin of the Forest, (X_F, Y_F). This can be done by U_k^F = ScalarAdd(Cat(UShape(X_k-X_F,0),U_k), $Y_k - Y_F$) and L_k^F = Cat(LShape(0,$Y_k - Y_F$), ScalarAdd(L_k, X_k-X_F)). Note that the extended shape functions, U_k^F and L_k^F, are computed by composing Cat() and ScalarAdd() in different orders but have the same form with the roles of X and Y reversed. This symmetry in shape operations used to compute the upper and lower shape functions is one of the benefits of using the same basic representation for these functions, but with the roles of X and Y reversed. **Runtime Note:** the running time for computing U_k^F and L_k^F is O($|U_k^F|$) and O($|L_k^F|$), respectively; that is, the number of steps in each of these functions.

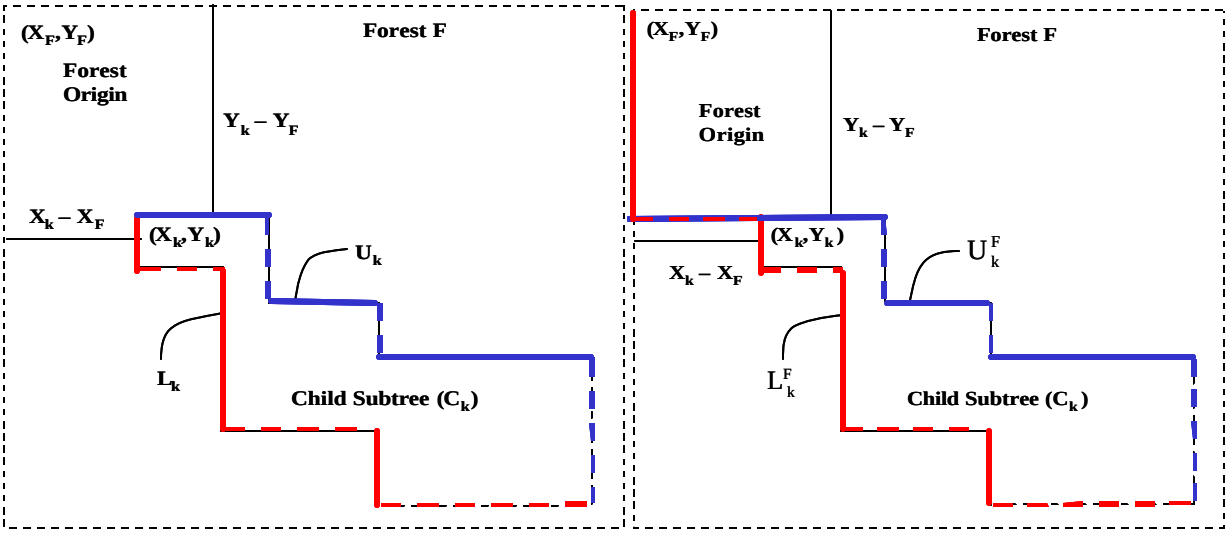

Figure 5a. Shape Function (U_k and L_k) of a Child in a Forest
5b. Extended Child Shape Functions relative to the Forest Origin.

By extending the shape functions for individual trees within a Forest (Figure 5b), we are able to define shape functions, U_F and L_F, for a Forest, F. The algorithm for constructing U_F and L_F can now be given.

(1) Let P be an existing node, the parent of a tree, T, we are about to construct. The tree that results will be a simple matter of placing P relative to the forest composed of its child subtrees, C_1, C_2, …, C_n. If n = 0, then T = P and the shape functions are computed according to Op-1 above. If n > 0, continue with step (2).

(2) Define the origin of a Forest, F, by $(X_F, Y_F) = (X_P+W_P, Y_P)$. Initialize $F = F_1 = \{C_1\}$ to contain the first child subtree, C_1, by setting the coordinates of C_1 to coincide with the origin of F. The algorithm proceeds iteratively by adding C_{k+1} to F_k to obtain F_{k+1}, for $2 \le k \le n$. If we let L_F^k and U_F^k denote the bounding shape functions for the Forest, F_k, the Forest obtained after adding k child subtrees, then $L_F^1 = L_1$ and $U_F^1 = U_1$, the shape functions for C_1. The iterative step is (3).

(3) For $2 \le k \le n$ do the following:

(a) Set $\delta = Vspc + MaxElt(\ Diff(\ \Lambda_F^{k-1}, U_k))$, where Λ_F^{k-1} is the UShape function obtained from L_F^{k-1} and U_k is the UShape function associated with C_k.

(b) Compute the origin (X_k, Y_k) for C_k as follows: $(X_k, Y_k) = (X_F, Y_F + \delta)$. Add C_k to the Forest, F; that is, $F_k = F_{k-1} \cup \{ C_k \}$.

(c) Compute the new shape functions for F_k as follows: $U_F^k = Min(U_F^{k-1}, U_k^F)$ and $L_F^k = Min(\ L_F^{k-1}, L_k^F\ )$, where U_k^F and L_k^F are the extended shape functions defined by C_k at its new origin. Specifically recall,

$U_k^F = ScalarAdd(Cat(UShape(X_k\text{-}X_F,0),U_k), Y_k - Y_F)$ and $L_k^F = Cat(\ LShape(0,Y_k - Y_F), ScalarAdd(L_k, X_k\text{-}X_F))$.

In these expressions, $X_k\text{-}X_F = 0$, and $Y_k - Y_F = \delta$. Thus U_k^F reduces to just $ScalarAdd(U_k, \delta)$ and L_k^F reduces to $Cat(\ LShape(0,\delta),\ L_k\ )$. Since the first step of L_k has a relative x-coordinate of 0, then the result of the Cat() operations simply increases the δy_1 component of the first step by δ.

Runtime Note: The running time for (3a) and (3c) is $O(\ |U_k^F| + |L_k^F|)$, thus the running time for (3) does not exceed $O(2b*d') = O(d')$, where d' is $max(max\{\ |U_k^F|\ |\ 2 \le k \le b\ \}, max\{\ |L_k^F|\ |\ 1 \le k \le b\text{-}1\ \})$, but under our assumption of a balanced tree, $d' \approx \log_b(N)$.

These computations are traced for the Paste operation illustrated in Figure 4 and are summarized in Table 1 below.

Table 1. Computation of Bounding Shape Vectors and Child Placement

k	U_k	L_k	U_k^F	L_k^F	d	U_F^k	L_F^k	Λ_F^k
1	**<(6,0)>**	**<(0,3)>**	**<(6,0)>**	**<(0,3)>**		**<(6,0)>**	**<(0,3)>**	**<(6,3)>**
2	**<(14,0)>**	**<(0,2), (8,6)>**	**<(6,0),(8,4) >**	**<(0,6), (8,6)>**	**4**	**<(14,4)>**	**<(0,6), (8,6)>**	**<(8,6), (6,12)>**
3	**<(13,0), (7,3)>**	**<(0,3), (7,6)>**	**<(6,0),(8,4), (6,16)>**	**<(0,16), (6,6)>**	**13**	**<(13,13), (7,16)>**	**<(0,16), (7,6)>**	**<(7,16), (13,22)>**
4	**<(7,0)>**	**<(0,3)>**	**<(6,0),(8,4), (6,16)>**	**<(0,20), (6,2)>**	**17**	**<(7,17)>**	**<(0,20)>**	**<(7,20)>**

Cut operations use the same computations with slightly different preliminaries. If a subtree S, rooted at node R with parent P, is removed from T, then node R is unlinked from its parent P and its siblings (if any). The computations of Algorithm 2 are then applied to the remaining children of P. The computation defined Algorithm1 then propagates the change all the way to the root of the new tree.

Algorithm 3 (Parent Update)
Once the Forest of children has been computed as described in Algorithm 2, the shape functions for the entire subtree rooted at the Parent, P,mu st be updated. This implies that each node stores the shape functions for the node itself and also the subtree rooted at that node. Let T denote the subtree rooted at P and let F denote the Forest of children of P. Then U_T = Cat(U_P, U_F) and L_T = Min(L_P, ScalarAdd(L_F, W_P)). **Runtime Note:** running time for Parent Update is $O(|L_F|)$ due to the Min() operation. The Cat() operation is O(1). By the analysis presented above for Algorithms 1, 2 and 3 it follows that Cut and Paste operations, under a balanced tree assumption, have a runtime cost of $O(\log_b(N)^2)$.

(Op-4) *Reading/Writing a tree to an external file*. Along with incrementally maintaining the shape vectors of a node, a count of the nodes in its subtree is also maintained. Thus when a tree is to be saved to an external file, the root of the tree contains the total number nodes to be written out. Using this count, an array of node pointers can be dynamically allocated to facilitate the output process. The array is filled in by performing a depth-first, first-to-last child traversal of the tree. A node's output index is assigned (stored in the node) and its pointer stored in the array at that index if it is a leaf node or if all its children have been assigned output indices. The root of the tree will always have the largest output index. Once the array has been constructed, the nodes are output to the file using the array – the node defined by the first element is written out first, the node (tree root) in the last element is written out last. When a node is written out, each of its five links (parent, upper sibling, lower sibling, first child, last child) are replaced by the output index of the corresponding node. Using this algorithm for writing out a tree it is not necessary to write out the shape vectors. They can be reconstructed when the tree is loaded back into memory by essentially reversing the output process. **Runtime Note:** Both output and input operations require O(N) time, where N is the number of nodes in the tree.

(Op-4) *Selecting a node.* The most basic tree operation is selecting a node to use as the basis for defining the above operations. Using a mouse, the user clicks the node to be selected for the next operation. The editor receives S = (X_{click}, Y_{click}), the coordinates of the node selected. Starting at the root of a tree, a simple recursive descent algorithm is used to locate the selected node. By taking the 2D-vector difference between S and a node's origin point, the relative coordinates of the selected point (S') can be computed. Using S' it is a simple computation to determine if the selected point lies within the bounding rectangle of the node itself, or using the node's shape vectors, whether the selected point lies within its subtree. If the selected point lies within the subtree, but not its root node, R, the computation is repeated for each child of R. **Runtime Note:** Clearly the worst case is O(N), but under a balanced tree assumption, Selecting a node requires roughly O(log(N)) operations.

4.0 Related Work

Early work on dynamic drawing of trees was given by Moen [9]. He provides an algorithm for dynamically constructing drawing of trees using subtree contours stored as polygons. The algorithm processes subtrees independently, placing subtrees of a node as close together as possible. Cohen et al [3] provide dynamic drawing algorithms for drawing several classes of graphs including trees. The key data structure for incremental editing consists of representing the tree as a collection of disjoint directed paths which must be recomputed after edit operations. Their algorithm does not attempt to preserve the mental map of the graph during updates.

Other authors have focused on dynamic drawings for special classes of graphs or other types of layout. North [10] provides a dynamic hierarchical graph drawing heuristic Dynadag, for general graphs. It is based on the static layout algorithm of Sugiyama, Tagawa and Toda [13]. Incremental algorithms have also been developed for orthogonal layouts [11]. Several drawing systems that are well known to the Graph Drawing community support dynamic drawings. Dynadag is included in the Dynamic layout manager of the GraphViz system [5] for handling hierarchical layouts incrementally. The daVinci [4] incremental graph layout is used to stabilize graph updates when nodes and edges have been added to a given graph. In Tom Sawyer Software Graph Layout Toolkit [7], the hierarchical and symmetric libraries support incremental layout. Graphlet [6] combines an editor and a library of dynamic layout algorithms. Incremental computation is performed using a 'gadgets-dependency graph' which maintains information on which objects must be redrawn. DiBattista et al [2] provides a comprehensive reference to incremental techniques for graph drawing.

5.0 Conclusion

In this paper, we have presented a new technique for an incremental tree editor based on algorithms for manipulating shape functions. The boundaries of subtree objects are defined as monotonic step functions and are manipulated to determine the best placement of the subtrees. The framework was applied to hierarchical tree drawings with a horizontal, left-to-right layout. The drawing system supports a full complement of editing operations, including changing node size and shape and insertion and deletion of subtrees. In redrawing the tree after user changes, the system reuses global layout information and only localized subtree data need to be updated. This ensures that the drawing is stable and incremental changes do not disrupt the user's sense of context. The algorithms conserve the total display area allocated to child-subtrees and their runtime and space efficiency is good with all tree operations being worst case $O(N)$ in time.

The tree editor was described for a very specific layout of hierarchical trees. Our approach can be generalized to a family of Shaped-based algorithms where the position of the parent relative to children and relative positions of children to each other are based on layout parameters.

References

[1] Arefi, F., Hughes, C., Workman, D., Automatically Generating Visual Syntax-Directed Editors, *Communications of the ACM*, Vol.33, No. 3, 1990.

[2] Di Battista, G., Eades, P., Tamassia, R., Tollis, I., *Graph Drawing Algorithms for the Visualization of Graphs*, Prentice Hall, New Jersey, 1999.

[3] Cohen, R., DiBattista, G., Tamassia, R., Tollis, I., Dynamic Graph Drawings:Trees, Series-Parallel Digraphs, and Planar ST-Digraphs, *SIAM Journal on Computing*, Vol.24, No. 5, 970-1001, 1995.

[4] Frohlich, M., Werner, M., Demonstration of the interactive graph-visualization system da Vinci, *Proc. Symposim on Graph Drawing 1994, in Lecture Notes in Computer Science*, 894, Springer-Verlag, 266-269, 1995.

[5] Gansner, E., North, S., An Open Graph Visualization system and its applications to software engineering, *Software-Practice and Experience*, vol. 30, 1203-1233, 2000.

[6] Himsolt, M., Graphlet: Design and implementation of a Graph editor, *Software – Practice and Experience*, Vol 30, 1303-1324, 2000.

[7] Madden, B., Madden, P., Powers, S., Himsolt, M., Portable Graph Layout and Editing, *Proc. Symposim on Graph Drawing 1995, in Lecture Notes in Computer Science*, Vol. 1027, Springer-Verlag, 385-395, 1995.

[8] Misue, K., Eades, P., Lai, W., Sugiyama, K., Layout Adjustment and the Mental Map, *Journal of Visual Languages and Computing*, Vol. 6 No. 2, 183-210, 1995.

[9] Moen, S., Drawing dynamic trees, IEEE Software, Vol. 7, 21-28, 1990

[10] North, S., Woodhull, G., Online Hierchical Graph Drawing, *Proc. Symposim on Graph Drawing 2001, in Lecture Notes in Computer Science*, 2265, Springer-Verlag, 232-246, 2002.

[11] Papakostas, A., Six, J., Tollis, I., Experimental and Theoretical Results in Interactive Orthogonal graph Drawing, Proc. Symposium on Graph Drawing 1996, in Lecture Notes in Computer Science, Vol. 1190, Springer-Verlag, 101-112, 1997

[12] Pothoven, S., A Portable Class of Widgets For Grammar-Driven Graph Transformations, Computer Science Masters Thesis, University of Central Florida, 1996.

[13] Sugiyama, K., Tagawa, S., Toda, M., Methods for Visual Understanding of Hierarchical Systems, *IEEE Trans. On Systems, Man and Cybernetics*, Vol 11, No. 2, 109-125, 1981.

[14] D. Workman, GRASP: A Software Development System using D-Charts, Software – Practice and Experience, Vol. 13, No. 1, pp. 17-32, 1983.