2006

Experimental Analysis And Evaluation Of Tidy Tree Drawing Algorithms

Pankaj Mahajan
University of Central Florida

EXPERIMENTAL ANALYSIS AND EVALUATION OF TIDY TREE DRAWING ALGORITHMS

By

PANKAJ MAHAJAN
B.E. University of Pune, 2000

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the School of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2006

ABSTRACT

Tree Drawings have been used extensively in software engineering and many other business and computer applications. The basic structure of a tree allows for the organization and representation of complex information. Many commercial tools allow their users to draw or construct trees to represent a problem and/or its solution. Our focus is on dynamic trees - trees subject to frequent changes and redisplay in highly user-friendly interactive computer applications.

Tree presentations in such interactive tools have to be precise and maintainable, which means, the tree presentations should maintain a particular structure so that user's mental perception of the tree is not disrupted or changed drastically when modifications are made to the tree being manipulated. Minimal modifications to the tree should cause correspondingly minimal changes to the general layout of the tree drawing and such changes should be consistent with the original layout to enable the user to anticipate them and verify their correctness with minimal mental effort. Also, display properties, like Vext, Hext, aspect ratio and space utilization efficiency of the layout are important to the user as they influence efficient use of available drawing/visualizing space which in turn affects comprehensibility of the tree drawing in question.

In this thesis report, we analyze and compare three published algorithms, proposed by Workman-Bernard[1], S. Moen[3], and R. Cohen [2],to interactively manage the layout of graphically represented dynamic trees. We attempt to measure and analyze the performance of these algorithms based on their layout properties and their computational requirements. This research concludes that the Workman-Bernard (WB) algorithm when compared with its closest equivalent, Moen’s algorithm, produces trees with better layout at a significantly lower computation cost.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In computer science, a Graph is an abstract data structure that consists of a set of nodes and a set of edges that establish relationships (connection arcs) between the nodes. The graph abstract data type follows directly from the concept of graph as a mathematical object. In practice, some information is associated with each node and edge. A connected graph is one where in all the nodes in the graph are connected to every other node either directly or indirectly through the edge relations. Trees are connected graphs with no circuits, which mean there is one and only one direct path from one node to any other node in the tree and removal of any one edge will leave the tree unconnected to at least one of its nodes.

Tree representations are used to graphically represent many things such as database organization or structure, file and system management, user and groups, organizational structures even object attributes and methods. Almost everything that is hierarchical in nature can be demonstrated in a tree drawing. The lucidity that a graphical representation provides is very valuable and trees especially are one of the most simple-to-understand information structures.

1.1 Trees Drawing Basics

A tree consists of nodes and connectors connected in a non-circuitous manner. When drawing trees care has to be taken to allocate space for the connectors as well as the nodes they connect. Figure 1 shows a simple layout of a tree with parent and two child nodes. Any tree drawing possesses some common properties which will be discussed in this section.

When drawing trees there are two kinds of connectors can be used – *orthogonal connectors* and *angular connectors*. The term "connector" as we use it in this document refers to a connector that joins two nodes and indicates some relationship between the nodes it joins.

1.1.1 Orthogonal connectors and Orthogonal Trees

Connectors are composed of one or more straight line segments connected end to end to join two nodes. A connector is said to be *orthogonal* if and only if the line segments forming such connectors are aligned parallel with either the horizontal or the vertical display axes. Trees drawn using orthogonal connectors can be termed as 'Orthogonal Trees'. Each child is connected to its parent node using a connector branch emanating from the connector stem for the parent node. This typically means node placement needs extension of the stem vertically (or horizontally) and then branching perpendicular to the stem to allow the node to connect.

Figure 1 illustrates tree layouts using orthogonal connectors.

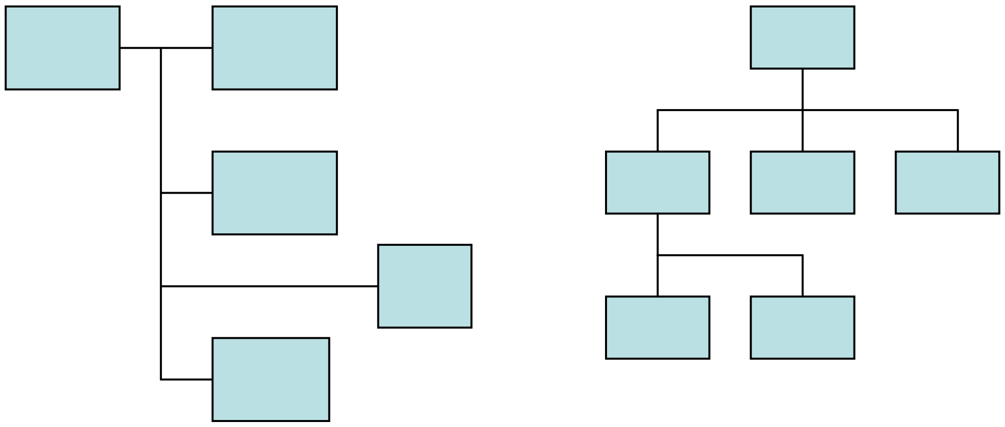


Figure 1. Tree layouts using orthogonal connectors

1.1.2 Angular Connectors and Angular Trees

Angular connectors are connectors represented by one and only one line segment that may or may not be aligned with the display axes. Figure 2 illustrates tree drawings using angular connectors.

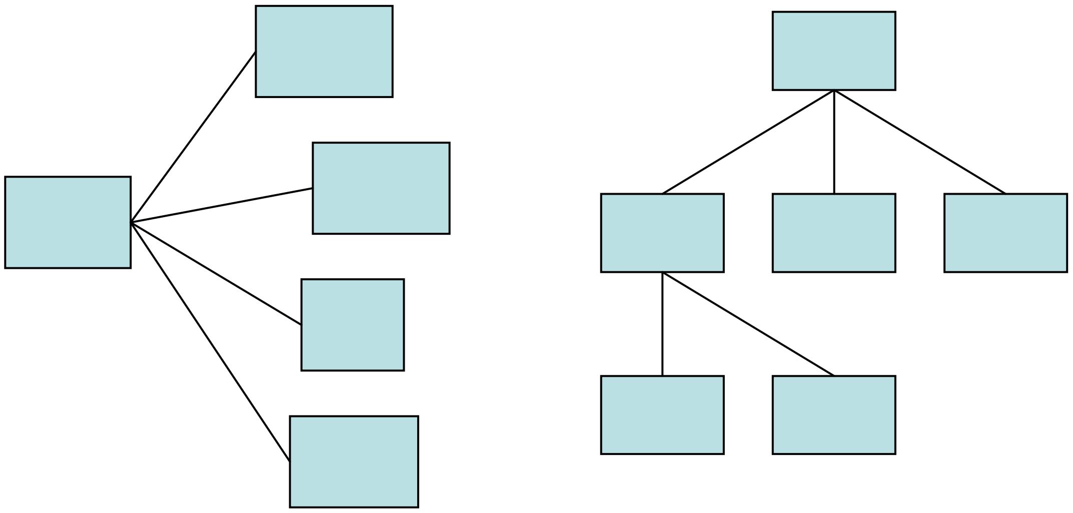


Figure 2: Tree drawings using angular connector

These connectors sometimes use curves instead of lines, however those cases will be beyond the scope of our research goal. The algorithms we consider in our experiment are those developed by Robert Cohen[2], Sven Moen[3] and D. Workman and M. Bernard (WB)[1]. Specifically we focus on trees (non-cyclic graphs). In WB and Cohen’s Algorithm, we will use orthogonal connectors, while Moen’s algorithm uses angular connectors.

1.1.3 Drawing Area

The drawing (tree layout) is limited or contained by the 2D plane enclosing the entire drawing. This plane shall be referred to as drawing area. The nonabstract equivalents of a drawing plane may be a canvas, a screen or a sheet of paper used to illustrate the drawing.

In abstract space, two dimensions of the drawing area are determined based on the space occupied by the drawing. If the dimensions cannot be predetermined, an expanding drawing area is used so as to accommodate any scale changes.

1.1.4 Drawing Resolution

Like any picture or painting, a tree drawing will also have a minimum finite drawing unit (FDU) area for the drawing (2D pixel size). This finite area is defined by the minimum distances between any two points on the actual drawing. For convenience, this can be further expanded to define finite minimum horizontal distance (*fduW*) and finite minimum vertical distance (*fduH*) for a graph. These units represent a logical unit for the tree drawing and it is not necessary to associate a value of physical dimension with this unit for analysis purpose. In applications of the algorithms, this parameter can be controlled to fit the layout within the physical limits of the drawing surface such as a canvas or sheet of paper. Figure 3[1] illustrates these geometric display concepts.

[1] This figure is reprinted from [1].

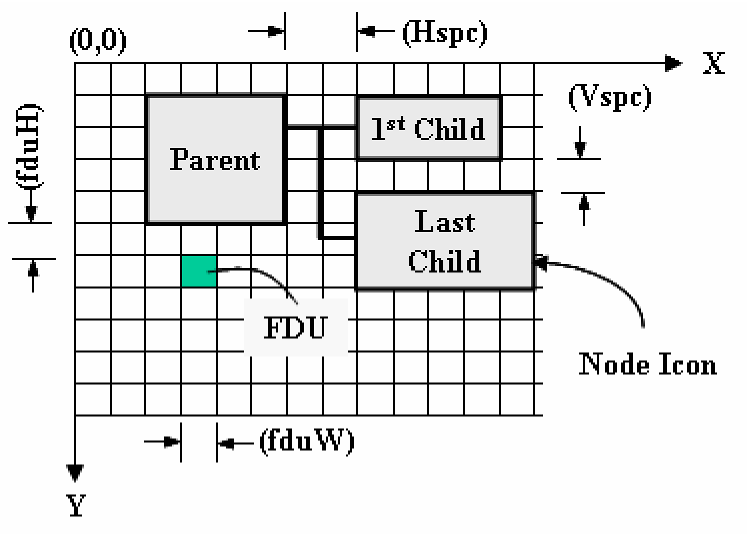


Figure 3: Drawing area, drawing resolution and coordinate axes.

1.1.5 Node Icon/Glyph

A node of a tree is represented on the tree drawing with a definite outline that varies from application to application. This outline of is called the *icon.* In real world implementations, the shape and size of the icon can be defined by a particular node subtype or by a general parameter when the node is created. Along with the actual icon, it is necessary to provide some spacing around the icon area to provide some physical isolation from the connectors and other icons in the tree drawing.

The smallest rectangular area enclosing the icon along with the icon spacing is called the ***node glyph or glyph***. However, for the purpose of our analysis, we restrict the icons and glyphs to rectangular shapes.

1.1.6 Node Spacing

When drawing the nodes, it is desirable for the nodes to be separated from each other in order to allow the viewer to perceive the nodes (icons) to be individual elements. For this purpose, each glyph includes horizontal and vertical spacing around the actual node icon. The layout algorithms can explicitly specify the rules to accommodate such spacing requirements or it can be left to the implementation to resolve this issue. In the discussion of Moen we will see explicit specification of these rules. Cohen and WB leave this issue to the implementation.

1.1.7 Connector Spacing and Connector Length

Connectors indicate relationships between various nodes in a tree. While drawing a tree, connectors are represented by orthogonal or angular connectors. These connectors are formed using one or more lines. It is necessary for a viewer to be able to view these connectors and so they have to be represented by lines of convenient lengths. These connector lengths can either be explicitly provided as a parameter, decided upon by the layout algorithm, or decided upon by the implementation. Also, to avoid overlapping and thus obscuring the connector relation, they need to be properly isolated with white-space called the *connector region*.

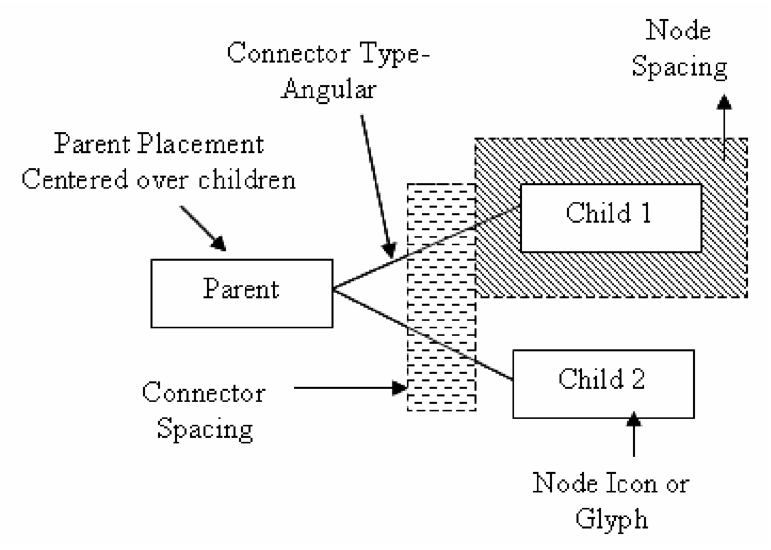


Figure 4: Basics of Tree Drawing Layout

For orthogonal connectors, node spacing around the icon is sufficient to include spacing for the connectors. This spacing is shared equally by the parent node and its children.

1.1.8 Other Terminology

The ***block area*** of a tree drawing is the area occupied by the smallest bounding rectangle that encapsulates the entire tree drawing. The block area of a tree is always greater than or equal to the ***tidy area*** occupied by smallest ***bounding polygon*** that encapsulates the tree drawing. These two concepts are illustrated in Figure 5. A bounding polygon is called a ***bounding shape*** or simply a ***shape***.

Frequently shapes can be expressed as the union of two piece-wise linear functions: a ***upper shape function*** *($y = U(x)$)* and the ***lower shape function*** *($y = L(x)$)* of a given tree. When bounding polygons can be represented by two shape functions, distinct benefits result in the speed and simplicity of the layout algorithms we shall study. In short, the manipulation of trees reduces to algebraic manipulation of their bounding shape functions – they become mathematical objects as well as information objects.

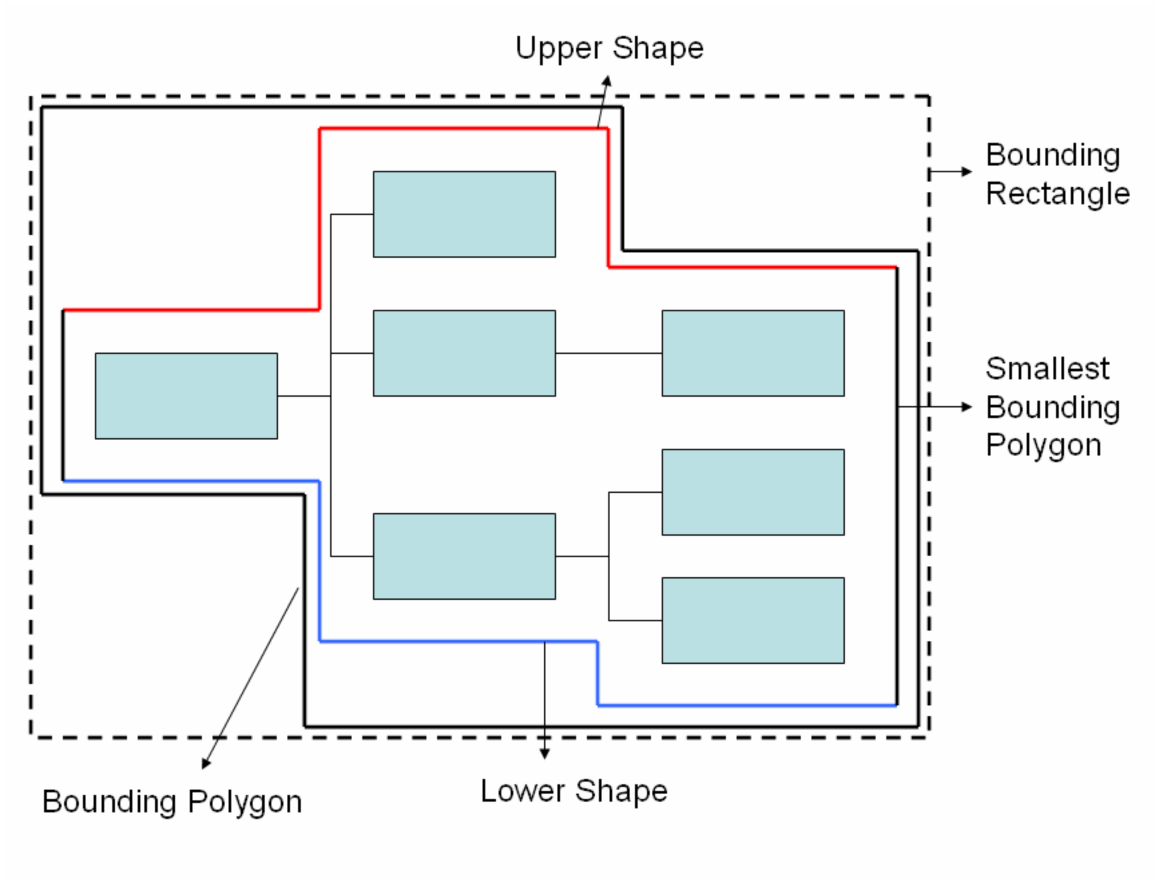


Figure 5: Bounding Polygon and Bounding Rectangle

Trees are composed of a parent or root node and a collection of child subtrees. The collection of these child subtrees is referred to as the ***forest of children*** for the given parent node. The count of trees in the forest of children is referred as the ***child-count***. For trees comprising a single root or leaf node, the forest of children will be an empty collection. Though the child-count may be arbitrary in general, ***binary trees*** are defined as trees for which the child-count is at most 2 for every node. Clearly, ***leaves*** can be defined as nodes with a child-count of 0.

The ***root node*** of a tree is unique and is the only node that has no parent.

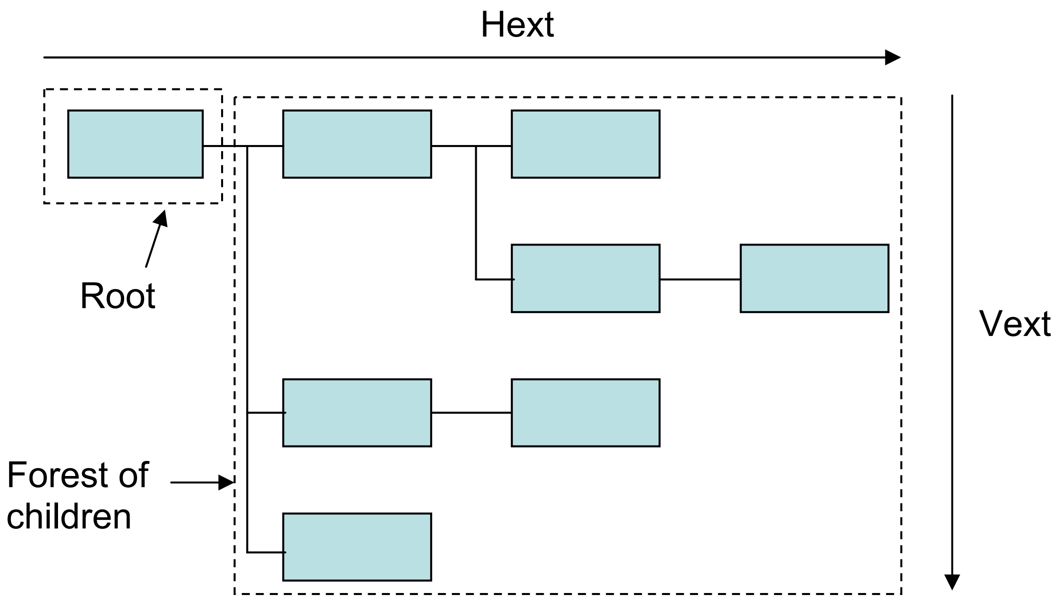


Figure 6: Root, Forest of children, Hext and Vext of tree drawing

The ***tree height (h)*** is measured as the number of nodes along the longest path from the root of the tree to any of the leaf nodes of the tree. This metric is also referred to as ***depth(d)***, usually to indicate the placement of a node within the tree structure.

The ***horizontal extent (Hext)*** of the tree drawing is always a multiple of *fduW* and measures the width of the smallest bounding rectangle of the tree drawing. Similarly, the ***vertical extent (Vext)*** is a multiple of *fduH* and measures the height of the smallest bounding rectangle of the tree drawing.

The ***display origin*** of a tree is the point of placement of the tree in the 2D display area and is defined to be the XY-coordinates of the upper left corner of the root node of the tree. These concepts are illustrated in Figure 7. The child forest also has an origin and this point is algorithm specific.

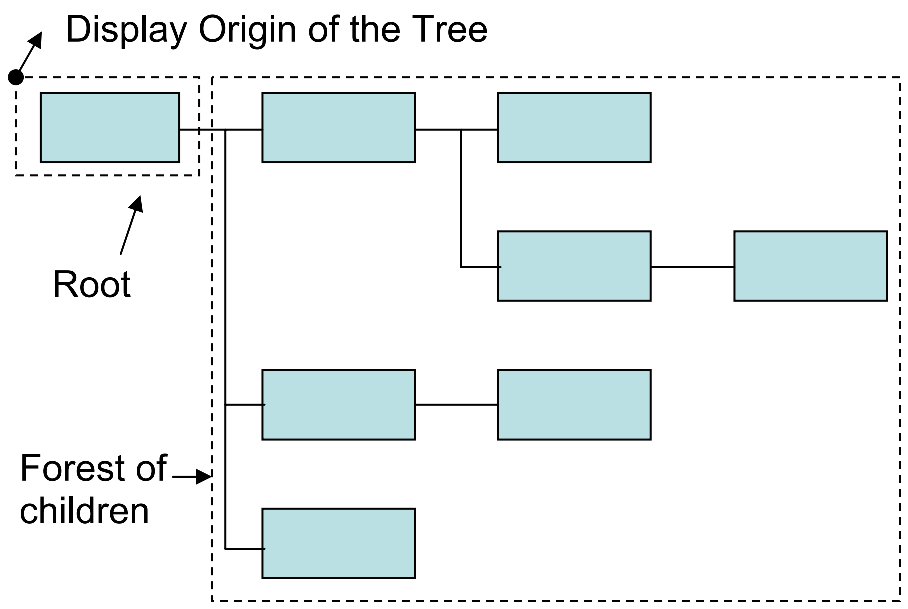


Figure 7: Display Origin of a Tree and a Child Forest

1.2 <u>Tidy Trees</u>

A ***tidy tree*** is a tree drawing wherein the nodes and connectors of the tree are placed together as closely as possible without overlapping or intersecting; that is, no node glyph intersects any other node glyph or connector, and the same is true for connectors (excluding the nodes they connect). Tidy trees are desirable because they occupy the least display area without obscuring the structure of the tree and any information that might be displayed in nodes and on connectors.

With consideration of a viewer's ability to perceive graphical details, excessive reduction of tree size will diminish overall readability. Conversely, excessively over sizing the drawing will cause some portions of the tree drawing to become hidden off-screen or out of view. The invisibility of these portions reduces information imparted to the viewer and thus diminishes understandability. Thus a tidy tree drawing attempts to maximize both readability and understandability.

The general approach to tidy tree layout taken by all the algorithms studied in this research can be summarized by two simple rules: (a) maintain a bounding polygon for each subtree, and (b) position the child subtrees of a given parent as closely as possible without their bounding polygons intersecting or overlapping. The algorithms we examined (Cohen, Moen, Workman-Bernard) vary in the way the bounding polygon is constructed and how the parent node is placed relative to the forest of its children. These variations have significant impact on the tidy area of the resulting layout, on other esthetic properties of the display that can affect appeal to the viewer, and on the computation resources required to produce the display.

Before introducing each of the three layout algorithms identified above it should be noted that these algorithms are not exhaustive of the possible variations to the general approach to tidy layout. We will return to this point in our discussion of future research on this topic.

1.3 Cohen's Algorithm

This algorithm was proposed in the 1995 by Robert Cohen[2]. The algorithm, though generalized for graphs, provides basic understanding and terminology for tree drawing. Specifically the paper is aimed at dynamically drawing a special family of graphs that includes trees. It points to the need of the algorithm to make local modifications to a graph without drastic changes to the structure of the entire drawing.

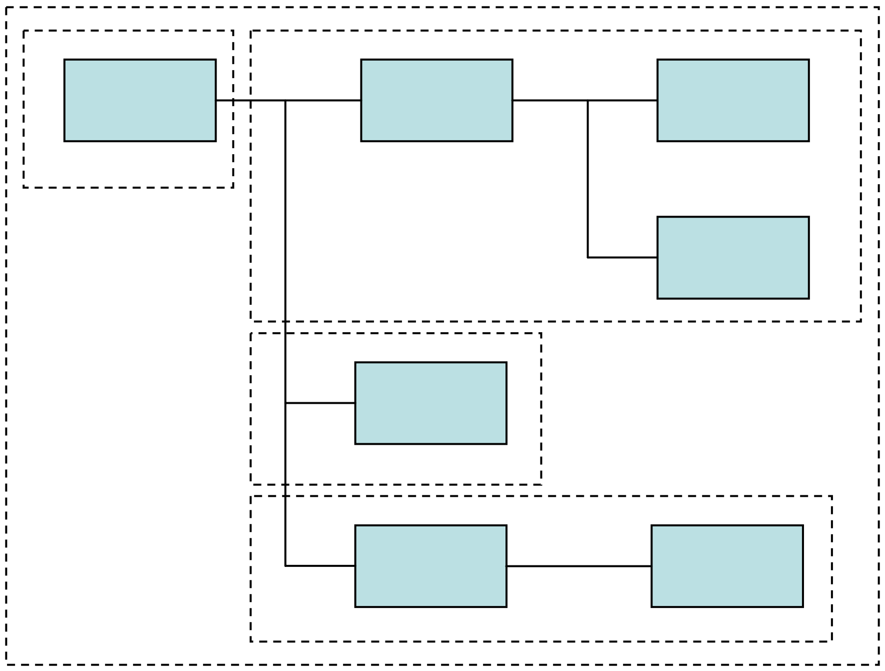


Figure 8: Cohen or Canonical Layout

It provides a baseline standard to compare with any other tidy tree algorithm because it uses a rectangle as its bounding polygon – this result in the largest possible tidy area, but is computationally the most efficient. The principle of this algorithm is to allow no overlap between the bounding rectangles of child sub-trees. The algorithm focuses on simplicity and speed of the layout and does not attempt to make the most efficient use of display space occupied by tree.

1.4 Moen's Algorithm

Sven Moen published his algorithm in 1990[3]. His algorithm was one of the first such algorithms designed to reduce tidy area below the upper bound established by Cohen's algorithm. Moen incorporated other esthetic qualities into tree drawing by enforcing two distinguishing features. Except for the special case of binary trees, Moen requires that the parent node always be placed centered over the forest of children. Second, Moen requires that all nodes at the same depth share the same X-coordinate in display space. Figure 9 illustrates Moen's general layout scheme.

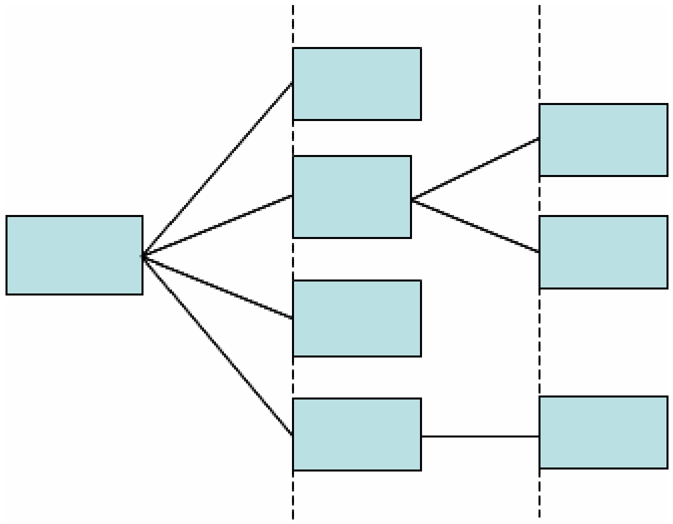


Figure 9: Moen Layout Example

The specific layout rules of Moen's algorithm were adapted from an earlier algorithm due to Reingold and Tifford [4]. They are listed below:

1. The origin of the parent node should always be to the left of the origin of its child forest;
2. The origin of all nodes at the same depth should share a common X-coordinate;
3. In binary trees, the right child should always be positioned above (smaller Y coordinate) its parent and left child should always be positioned below it parent;
4. A tree and its mirror image should be drawn to reflect each other;
5. A sub-tree should always look the same regardless of where it occurs.

In addition to the above rules, Moen's algorithm uses angular connectors. Node connectors have their own spacing requirements and will be discussed in greater detail in a later section.

Moen's placement algorithm employs two piecewise linear functions that form a bounding polygon for each subtree. Because these functions more closely approximate the actual display area occupied by the nodes and connectors of a subtree, Moen's algorithm is able to layout the child forest of a given parent in a much smaller space than that required by Cohen's algorithm. But like Cohen's algorithm, the parent is centered over the forest of children (except for binary trees).

1.5 Workman - Bernard Algorithm

The Workman-Bernard algorithm has a long history and evolved from research into graphical tools for software design and code layout. The early work on such tools resulted in a system called GRASP (GRAphical Specification of

Programs)[5]. In 1996, Pothoven[6] laid much of the ground work for more efficient interactive tools for dynamic layout of trees. Several refinements and improvements of Pothoven's work culminated in the Workman-Bernard algorithm presented here.

Like Moen's algorithm, WB uses a tight-fitting bounding polygon for subtrees that can be represented by upper and lower shape functions. Unlike Moen's bounding functions, the WB shapes are actually mathematical step functions. This small mathematical simplification over Moen's bounding functions accounts for a factor of two speed-up in the efficiency of the child forest compaction part of the tree layout.

WB distinguishes itself further from Moen in the policy for parent placement relative to the forest of children. WB ensures that the origin of the parent node and the origin of the child forest share the same Y coordinate. In Moen's algorithm, the parent is placed centered with respect to its child forest.

The TRED tool is an example of a dynamic tree drawing application. It was designed to represent program structure, specifically Java program structure in a tree format. Java applications consist of packages, classes, and interfaces, some of which will have member objects, variables and methods. This hierarchy of program structure can be completely demonstrated and illustrated using tree structures. Root of the tree usually represents the application root or package hierarchy root. First level may represent packages, second may represent classes and interfaces. The classes, packages and interfaces are differentiated graphically by use of distinct icons for both of them. Thus a viewer can see the program structure and possibly modify it graphically rather than textually. TRED is intended to

1. Read in Java code files and directories and display underlying program structure,
2. Allow user to modify the program structure graphically.
3. Reflect changes back into Java code to the files and directories involved.

While working on the TRED project, we noticed the specific properties desirable in a dynamic tree drawing algorithm. It also invoked interest in understanding the properties of trees as subject to drawing and the performance of algorithms used to draw them.

1.6 Desirable Properties of Dynamic Tree Drawing Algorithms

From the author's experience on the TRED project, the following properties of dynamic layout algorithms were found to be of particular interest.

Response time – The graph drawing is laid out for visualization by a human user. The time required for redisplay after an incremental change has to be small enough to maintain the illusion of real time update. Excessive delays in the redraw time increases the probability the user will lose mental context and point of focus. Typical incremental operations that require redisplay include *create*, *insert, delete, cut* and *paste*. Translating these changes into particular layout geometry may incur additional cost, but these costs can be assumed to be constant across all algorithms discussed here.

Display space utilization – The display space is defined by the physical medium used to display the tree e.g. a computer screen, paper, etc. An ideal dynamic tree drawing algorithm will avoid wasting available display space, especially at the cost of reducing the amount of information visible after a change is made. Display properties relating to display esthetics include – horizontal extent, vertical extent, aspect ratio (Hext/Vext), display area, wasted display area, compaction ratio (tidy area/block area), and node density (node area/ tidy area).

2 RELATED WORK

2.1 Cohen algorithm

The Cohen algorithm is the most obvious and straight-forward of the placement algorithms studied here.

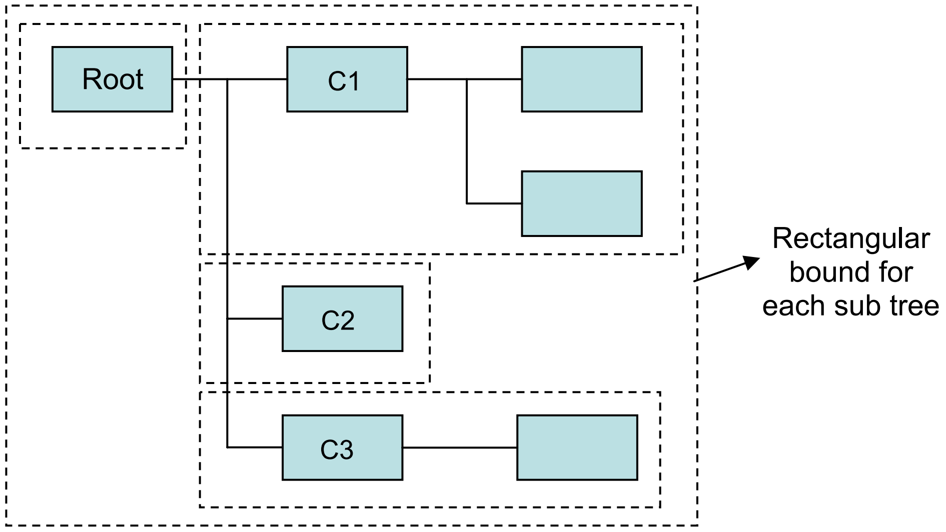


Figure 10: Cohen Layout

The algorithm uses a rectangle as the bounding polygon for every sub-tree. If, A and B are any two siblings, their bounding rectangles must be disjoint. Though the Cohen layout does not specify shape function specifications, it can be surmised that the shape functions for these trees are thus unit step functions. The block area of the trees equals the tidy area.

The upper shape function is a single step marked by the upper bound of the tree. The lower bound is also a single step marked by the lower bound of the tree. Thus the placement is mostly not compact. However, sibling placement requires comparison of shape functions which being only a single value comparison takes constant time. For simplicity the connector space is included in the border spacing around the icon.

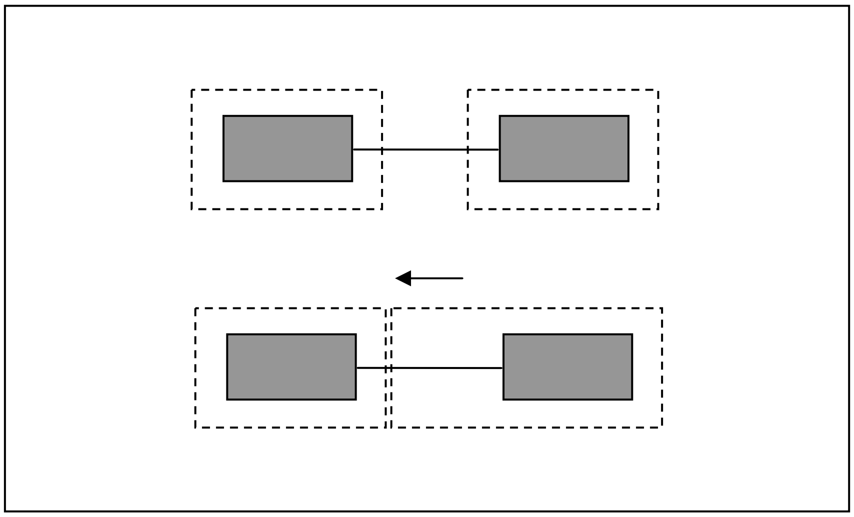


Figure 11: Inclusion of connector space in node area

2.2 Moen algorithm

The Moen algorithm was proposed by Sven Moen in his paper (reference marker). The algorithm attempts to provide a pseudo symmetrical placement of the nodes in an attempt to provide the 'Δ' shape for the tree. As an extension of the Reingold and Tilford algorithm (reference marker), it attempts to address practical tree requirement of non-uniform sized nodes. While Reingold and Tilford drawings are vertical with parent being above children, Moen algorithm tree drawings are oriented horizontally. This is because real world trees are typically nodes typically contain text of some sort, the aspect ratio of most text fonts allows more information to fit in the display area if the text us oriented horizontally. This is because most common contents of node information represented are text strings. As a common tidy tree rule, Moen algorithm processes sub-trees independently. Please note however that rule 2 and rule 5 of Reingold and Tilford cannot be implemented together given non-uniform sized nodes (reference marker).

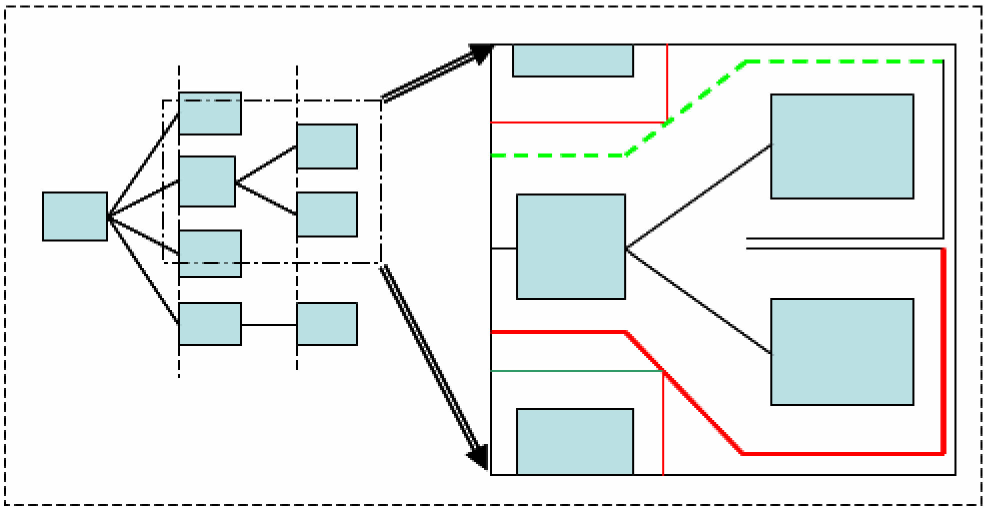


Figure 12: Moen Layout

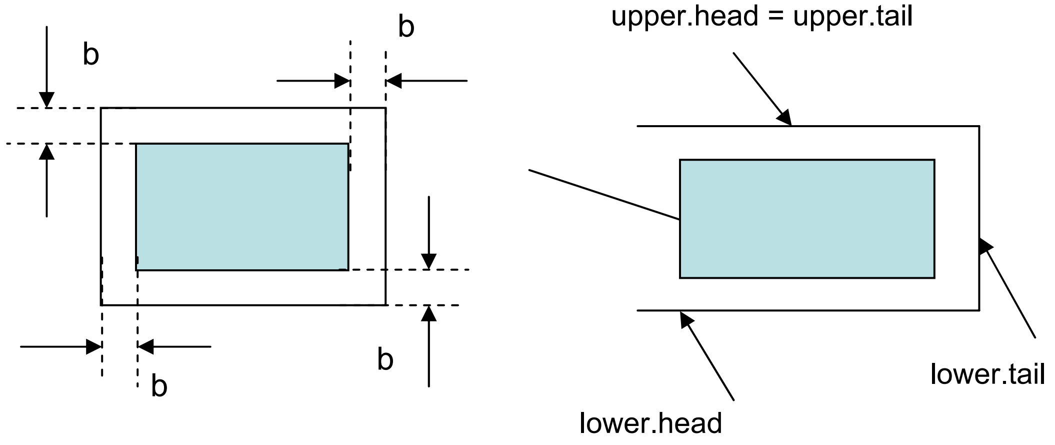
2.2.1 Moen Shapes

The main difference between Moen and Reingold & Tilford algorithms is the use of bounding shapes to achieve a tighter layout. Moen uses shapes as shown in green and red in fig. 12 to represent the profile of the trees. The sets of green and red lines of lines indicate the upper and lower shape polylines respectively. The profile of an entire tree will be stored at the root of that tree. This is used to compute distances between siblings while adding sub-trees or rearranging after updates. The shapes are maintained inside polyline structures, which hold a list of line segments expressed as a pair of (deltas) relative coordinates with respect to current position.

2.2.2 Functions in Moen’s Algorithm

2.2.2.1 Layout leaf

The creation of the leaf shapes involves providing horizontal and vertical spacing around the node icon. This represents the drawing space for the leaf node. The leaf’s shape however is left open on the left side and the right side is covered by the tail of the lower shape.



(a) Leaf and Border (b) Leaf Contour

Figure 13: Shape Functions (a) leaf shape, (b) non-leaf and connector shape

2.2.2.2 Merge

The parameters to this function are two polygon structures. The polygon structures may hold the shapes for two sibling trees. In case of merging with more than two children, one of the parameters would represent the shape of one or more sibling trees collectively already merged together. The function determines the minimum vertical offset at which the lower sibling can be placed with respect to its upper sibling so as to avoid overlap between their drawing spaces. For this it uses the offset function on each individual line in the shapes, upper shape of the lower sibling and lower shape of the upper sibling.

Once the offset has been determined, the shapes are merged together after placing the lower sibling at the vertical offset with respect to the upper sibling generating a collective shape. The collective upper shape reflects the upper sibling's entire upper shape and any portion of the lower sibling's upper shape that extends beyond the horizontal extent of the upper sibling. The collective lower shape reflects the entire lower shape of the lower sibling and any portion of the upper sibling's lower shape that may extend beyond the horizontal extent of the lower sibling. In addition to this, the bridge function is used where ever necessary to connect the two upper shapes or the two lower shapes which ever the case might be.

2.2.2.3 Bridge

The purpose of the bridge function is to connect two shapes where there is an abrupt change caused as a result of one of the shapes reaching its maximum horizontal extent. The function determines the projection of the termination point onto the other corresponding shape of the other sibling. For example, bridging two lower shape would be initiated, when the lower sibling has a lower horizontal extent as compared to its

2.2.2.4 Offset

Calculates the vertical offset between to given line segments. Parameters include the current offset, current x position, and coordinates representing the line segments as a pair of derivations (dx and dy).

2.2.2.5 Join

This method performs a merge operation on all the forests rooted at the parameter node t. It constructs a combined shape for all the sub-trees by using the merge and offset functions.

2.2.2.6 Attach_parent

This includes the parent/root node's dimensions in the shape. The procedure involves extending the shape of the forest rooted at that node one horizontal space leftwards. The nodes upper shape is then connected to this extended shape with a line segment, originating from a point at one vertical space above the nodes top right hand corner.

2.2.2.7 Layout

The Layout function traverses the tree down to the leaves attempting to produce the layout of the leaves first. These leaf layouts are progressively build up and stored in each node. The upper level nodes, produce the shapes by using the join and merge functions described earlier.

2.2.2.8 Unzip

This operation allows breaking the forest shape down into its component shapes – i.e. shapes of the children comprising the forest

2.2.2.9 Zip

This operation groups children together so as to form the forest shape out of the shapes of the children comprising the forest.

2.2.2.10 Branch

This method is used to add a sub-tree to an existing tree. The method has to be called on the expected sibling or the expected parent. It causes the data structures to initiate a change in the tree structure and then call the zip operation.

2.2.3 Placement using Moen's algorithm

Trees are laid out in a bottom up fashion by the Moen's Algorithm. The child trees are assembled before the parent tree and these children trees are assembled together to form the forest of children. The forest of the children provides the necessary data to put together the parent tree. The following describes the procedure used to place a sub-tree using a Moen layout algorithm. The operation described here is performed during the layout function of the algorithm. Before this operation starts, following conditions exist.

1. The sub-tree C and its siblings have being processed by the *layout* method. They all have a complete upper shape and a lower shape.
2. The root's position and its shape have not been finalized yet, however size of all the nodes is known, including that of the root.
3. The siblings are attached to their parent by the function *attach_parent*.

Once these conditions are achieved, a parent tree can be put together.

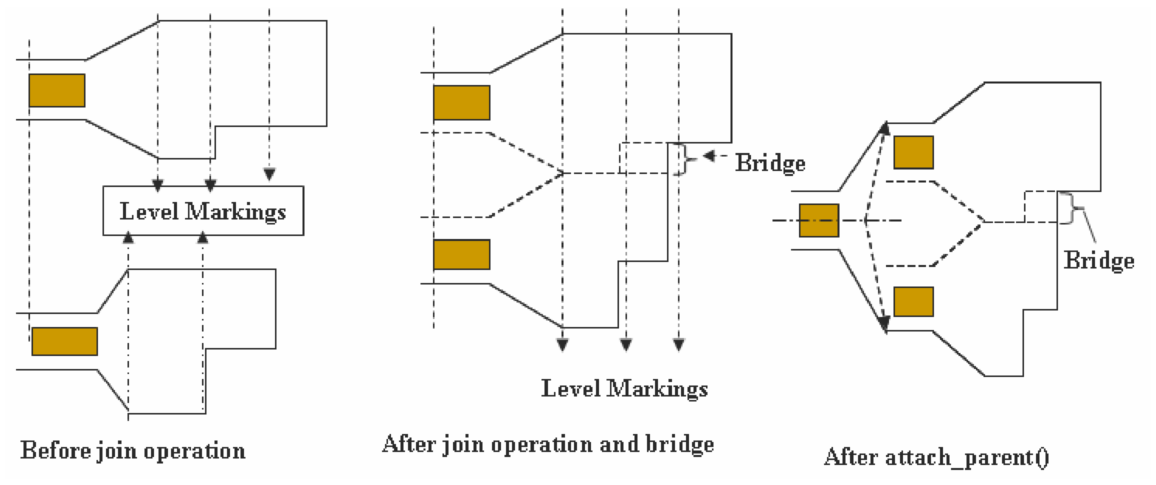


Figure 14: Sub-tree Placement using Moen

Figure 6 shows a child C being added as a child to a tree that is newly being created. The forest of the siblings of C is indicated as F.

Step1: According to Moen's algorithm, all the nodes at the same level start at the same X coordinate. Procedure used to achieve this is not described in the paper. The process may have been considered trivial and is omitted from the paper. Therefore, the best possible procedure is used to complete the task of establishing the common X coordinate levels for different tree levels.

1. The different X coordinate levels for each sub tree are maintained in its parent as a vector – let this vector be called the X-vector.
2. The parent also holds a vector indicating the minimum horizontal displacement necessary at each level within itself at each node level – let this vector be the Min-vector.

To ensure all the nodes in F and C that appear at the same level also appear on the same horizontal coordinates, a compare function is used to identify the minimal horizontal displacement required at each level.

Let, $F_{\min}$ and $C_{\min}$ be the min vectors for F and C respectively. Let, $F_{\min_k}$ represent the kth element in the vector $F_{\min}$. Similarly, $C_{\min_k}$ represent the kth element in the vector $C_{\min}$.

A new minimum displacement vector is computed as,

$F_{\text{newmin}} = \{y \mid y_k = \max(F_{\min_{k-1}}, C_{\min_{k-1}})$, where $y_0 = F_0$.

Using this new displacement vector the new X-vector (F_{newmin}) will be recomputed for the new forest F_{new} inclusive of C. Similar process is used for cut and delete operations.

Step 2: Once X_{new} is computed, the vector is stored in its parent updating the X vector for the parent. The change is then propagated from the parent node to all its siblings and to its parent. Each of the sibling node as well as the parent node will update the X-vector it holds as well as pass it down to all its children so they can update their X-vectors. Updating X vectors causes change in the shapes which will be recomputed at every level. Once the X-vectors are balanced and shapes are updated accordingly, only then can relative vertical placement occur.

Step 3: The vertical placement of each node is done through comparison of shapes between sibling's nodes or trees. A sub-tree ready for placement within the tree will have complete upper and lower shapes. The placement starts with the root, starting at a logical 0 vertically and horizontally. The nodes will be assigned Y coordinates relative to that of the root. To do this, the forest of children of the root is assembled together first. This assembly involves (starting with first child of the tree) comparison of the upper sibling's lower shape and the following siblings lower shape. Comparison is done on a step by step basis. Each piece of the poly-line is compared to corresponding piece of poly-line from the other shape for intersection and minimum vertical distance between the two pieces, so as to place the lower sibling's upper shape below the upper sibling's lower shape with no intersections. The no-overlap condition between the shapes guarantees no-overlap of the trees as the trees are completely enclosed within their shapes.

Step 4: Once a child tree is placed with respect to its upper sibling, the process is repeated with the following child, using the combined shape of the siblings above it as reference.

When all the child-trees are placed forming a forest of children, the parent is placed centered on the forest of children. The centering is not performed on the aggregate Vext of the forest of children, but rather on the Vext of the first level of the forest of children.

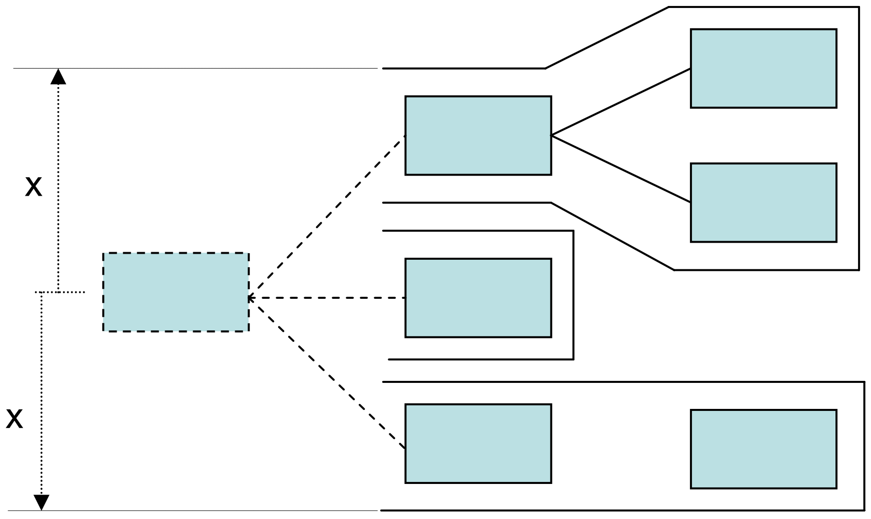
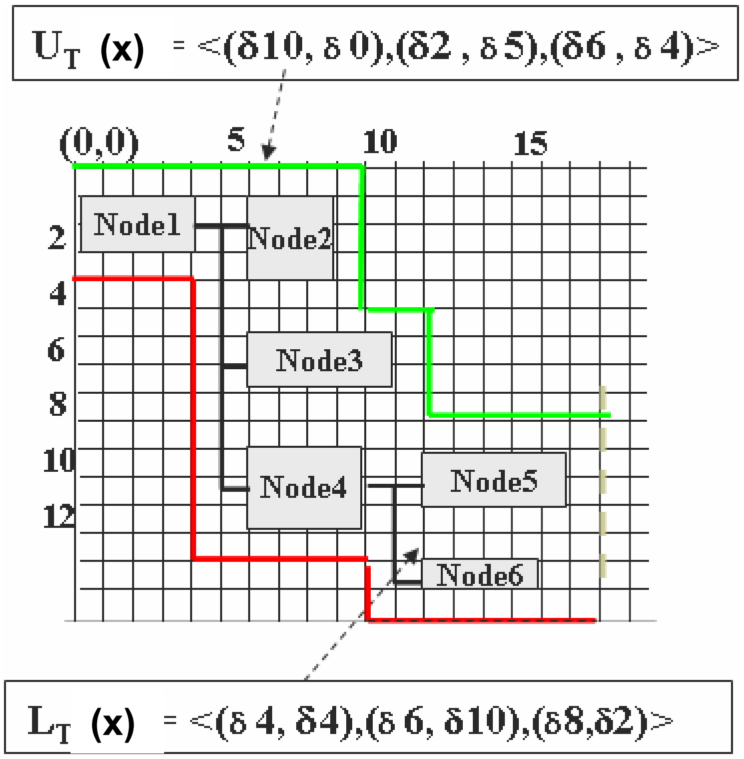


Figure 15: Center placement of parent

2.3 WB algorithm

Figure 16 shows an sample tree drawn using WB algorithm. The algorithm uses shape functions as bounding polygons for each sub-tree. Unlike Moen's Algorithm, WB algorithm uses discrete step functions to represent the shape of the trees. In fig. 16, the upper shape function is indicated in green while the lower shape of the tree is red. Theses shapes represent the entire tree. Each non-leaf node will preserve the shape of its forest of children for reuse for placement and replacement along the drawing surface.



U_T (x) = <(δ10, δ0),(δ2 , δ5),(δ6 , δ4)>
(0,0)
5
10
15
2
4
6
8
10
12
Node1
Node2
Node3
Node4
Node5
Node6
L_T (x) = <(δ4, δ4),(δ6, δ10),(δ8,δ2)>

Figure 16: Shape of Tree

By the requirement of the algorithm, all trees will be constructed from sub-trees and their shapes will not be affected by the context in which they appear. The connectors used for connecting parent to child/children and vice versa are not allocated with any special structure. Instead the space for the connectors is included in the space of each node. Thus every node will have a leading and a trailing space which will be occupied by any connectors connected to the node.

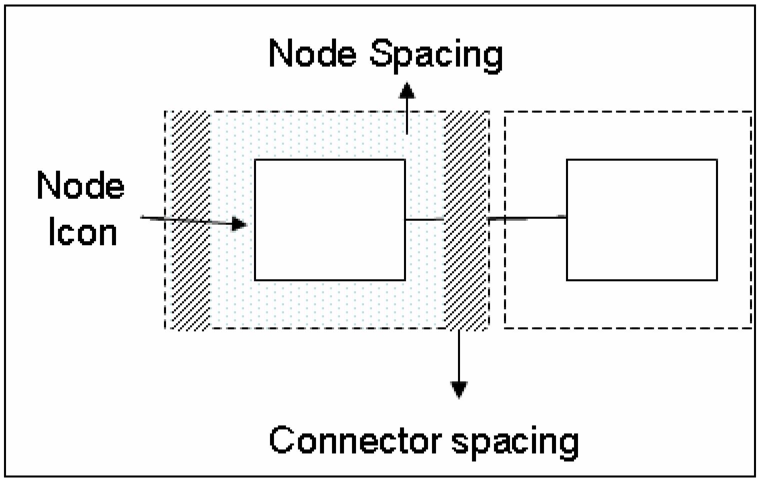


Figure 17: Layout of node and connector spacing

This arrangement eliminates need to process the connectors independently. The size of the connectors between parent and children is predetermined and constant. It can also be seen from figure 17, that the vertical spacing is also included in the node area. Thus, a global constant will determine the minimum vertical distance between any two nodes that appear aligned on a vertical axis.

2.3.1 Creation of a Tree

Trees are collections of sub-trees, the order of the sub-trees may or may not be specific or important, but there will be an order in which the tree will be assembled. The algorithm takes in assembled sub-trees and organizes them into non-overlapped formation by displacing each consecutive lower child by some distance *d*. The trees are thus created by inserting in the sub-trees as children of the root.

2.3.2 Generalized WB-Shapes

This shape function was derived from the legacy Shape functions. Instead of the dependent axis being different, generalized shapes use no specific axis as the basis. Instead the steps are constructed of step-length & step-height pairs. The step-length indicates the length of the step along x axis and step-height indicates the y axis coordinate for that step.

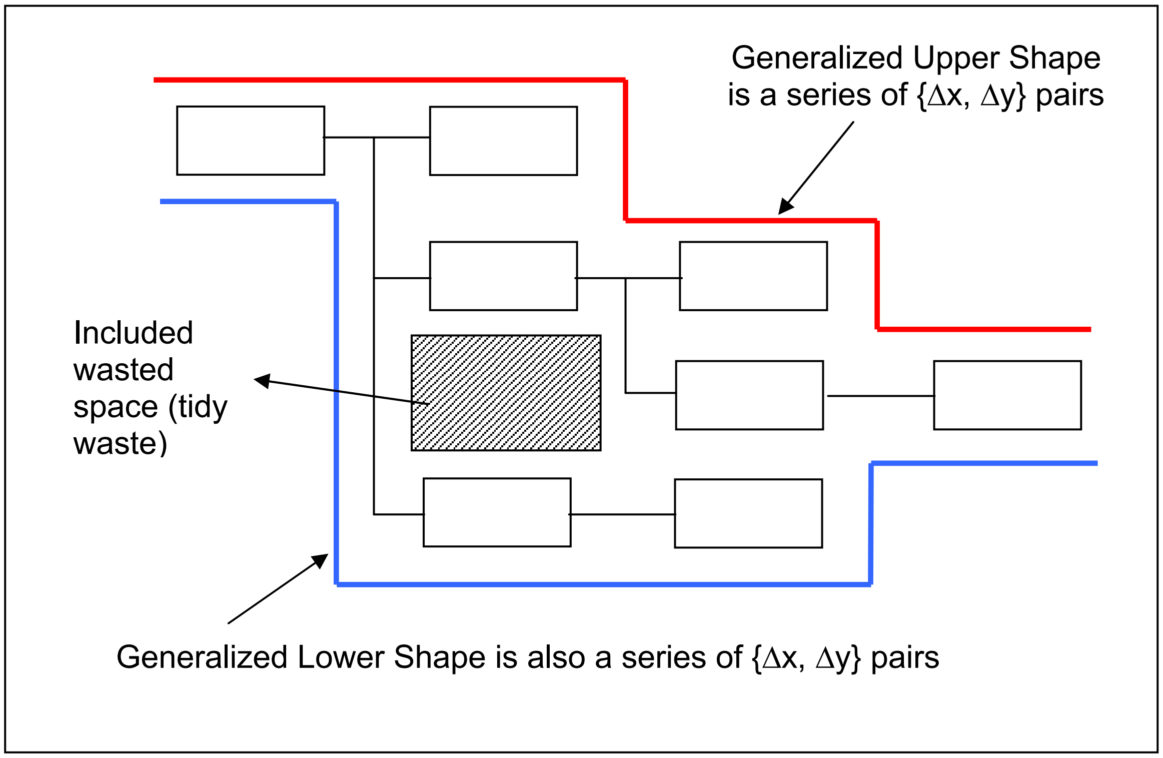

Generalized Upper Shape is a series of {Δx, Δy} pairs
Included wasted space (tidy waste)
Generalized Lower Shape is also a series of {Δx, Δy} pairs

Figure 18: Generalized WB – shapes – More precise fit

2.4 Comparison factors and parameters

The Tree drawing is a visual aid for representation and comprehension of information by a human user. This implies that a desirable tree drawing will not only be represented correctly, but also as compactly as possible without loss of readability. Also, the trees are more likely to be drawn or displayed on square-rectangular canvas or drawing surfaces like a computer monitor. Readability there for will directly correlate with the expected aspect ratio of 1. Another factor that affects Tree drawing readability is the amount of change inflicted over the entire tree by modifications to a certain part of the tree itself. Larger the overall change, the more difficult it will be for the user to comprehend the Tree Drawing. Therefore, it is desirable to have a minimal change over the entire drawing as far as possible.

3 PROPOSED RESEARCH

The analysis of the algorithms for generating and maintaining dynamic trees involves understanding trees – dynamic trees to be specific. The properties of trees influence the drawing parameters and hence the performance of the algorithms. It is very important to understand these properties first. Our formal goal – To analyze and compare performance of dynamic tree drawing algorithms - is therefore extended to understand the dynamic trees themselves.

The formal goal for the research is

1. To understand the specific tree drawing algorithms.
2. To analyze the computational performance of these algorithms while generating a tree from scratch.
3. To measure the drawing properties of the tree drawings with our focus on spatial layout properties concerning wasted space and aspect ratio of the drawing.
4. To analyze the dynamic performance of these algorithms in context of tree update operations like insertion, deletion and modification of nodes or trees or the root of the tree.
5. We are also interested in particular properties of tree drawings that are believed to represent aesthetically desirable properties for practical dynamic tree drawings.

Properties of interest

Tree Display Vext (recH) – This is the height of the tree drawing along the Y axis as measured in fduH.

Tree Display Hext (recW) – The width of the tree drawing along the X axis as measured in fduW.

Block area (AB) – Area of the smallest enclosing rectangle for the tree drawing.

Shape area (AS) or Tidy area – Area of bounding polygon of a tidy tree

The bounding polygon can be any data construct that outlines the profile of the tree drawing. In our case, Moen's algorithm uses Shapes and WB algorithm uses shape functions.

Cohen Area (AC) – Area of Cohen Tree drawing for same tree

Cohen area is used as the base reference as it is the simple and straightforward with minimal optimizations.

Tidy Waste (T.µS) – Unutilized area within bounding polygon of a tidy tree

Measures wasted space within the tree drawing's bounding upper and lower polyline or shape extents.

Block Waste (µB) – Unutilized space within the bounding rectangle of a tidy tree

Measures space wasted on the drawing surface.

Tidy Efficiency – (1- µS/AS) – Ratio of tidy waste to tidy area

This parameter represents the algorithms ability to utilize available space.

Block Efficiency – (1 – µB/AB) – Ratio of block waste to block area

Compaction ratio – (A_S/A_C) – Ratio of Shape area to Cohen area.

Measure of compaction achieved in comparison with Cohen (canonical) layout

Aspect Ratio – (recH/recW) – Aspect ratio of the tree drawing.

Average number of children per parent –

This is the average of the ratio of children per non-leaf node. In case of a unit tree (tree with only a single node) the root is a leaf and number of parents is 0.

3.1 Approach

In the practical applications of tree drawings, the size and shapes of tree nodes are usually variable. Ideally this research would consider all trees of all sizes and shapes in order to arrive at the exact estimate for practical applications of the algorithms being discussed here. However, the number of trees of any size involving non-uniform sized nodes is infinite as there seems. In order to down-size the problem to a manageable level, we consider trees with uniformed sized square nodes in this research.

Also, any specific example tree arrangement in practical use is determined by the information content in it. This causes most practical trees to have different tree arrangements and different sizes - in terms of number of nodes involved. For our research, we resolve this unknown by taking into consideration all the tree arrangements of a particular size. The cumulative/overall performance of the algorithms on such a forest of trees provides us with a representative performance for any practical tree of that size. The rate at which, the number of tree arrangements increase with each increment in total number of nodes in the trees, is very high. The number of tree arrangements reaches very large numbers quickly. Therefore we had to settle for an upper limit on the number of nodes the trees we analyzed.

3.2 Tool Requirements

The first task in this research was to verify and understand the algorithms involved. The second task was to enumerate the number of trees of a certain size. We discussed and implemented a couple of different techniques as we will discuss ahead. Once all the trees of a size are enumerated, we proceeded to drawing those trees and measuring the properties that we decided to measure and record our measurements for further analysis.

3.3 Tools

The TRED tool used WB algorithm to lay out the tree drawings involved. Experience on the TRED team and involvement in implementing WB algorithm provided the necessary understanding and validation of the WB algorithm. The algorithm description does not specify the data structures to be used allowing flexibility for implementation.

Though TRED performed quite well in drawing trees, it is incapable of producing or measuring the data we intended to measure. Specifically, for collecting data on trees of size N – all trees of size N had to be generated.

3.3.1 Initial approach to enumerating trees

This was a layman's approach to construct trees of size N by using all the Trees of size N-1. We take a tree of size N-1 as a base tree and add a child to each node of the tree to produce a tree of size N. Problems with this approach despite its simplicity was that it produced duplicate trees – trees with same tree structures can be created this way using different base trees. It required filtering and hence it was inefficient computation wise.

3.3.2 Binary code representation of trees

A very convenient way of representing a tree structure is by encoding it to its equivalent binary code format.

Trees structures with no information regarding appearance or display characteristics can be represented as a binary literal string. We call this string the binary code for the tree. To represent a tree in a binary format, 0's are used to represent nodes, while 1's represent the connectors. The order in which they appear is post order traversal of the tree. For example 00011 is a binary code for a tree with 3 nodes, where root has two children – i.e. two leaf nodes. The 1's can be interpreted as the number of connectors originating from the parent represented by the 0 immediately before the 1's toward the child trees or nodes represented earlier.

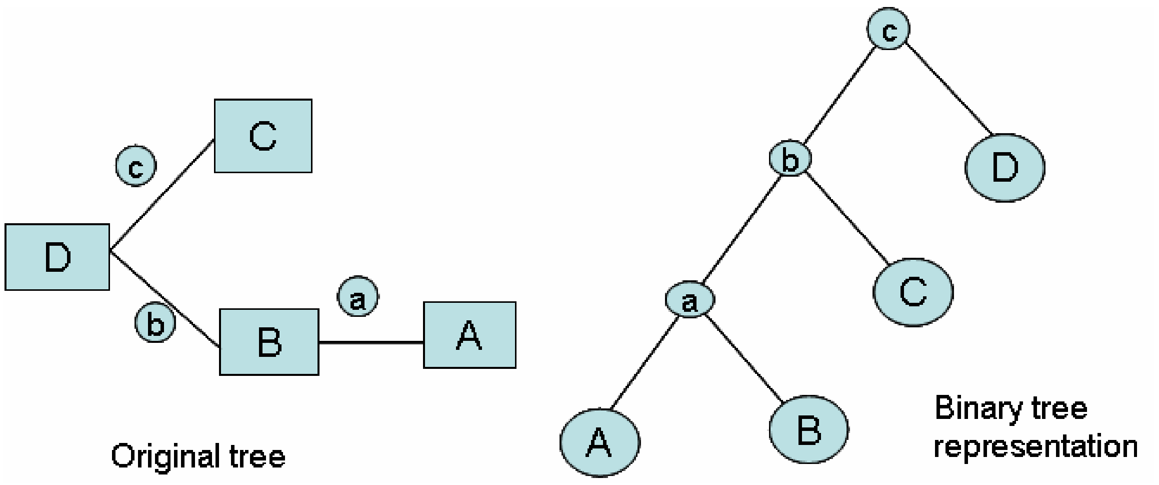


Figure 19: Tree and its equivalent binary representation

For example – Figure above shows a tree for the binary code 0010011.

First two 0’s represent “child 1” and “child1-child1”, and the following 1 represents the connector connecting them. Similarly the following two 0’s represent “child2” and “root” node. The last two 1’s represent the connectors originating from the “root” towards “child1” and “child2”.

Properties of a valid binary tree code are

1. Contain exactly one more “0” than the number of 1’s.
2. All valid tree codes will start with a “00” (“0” for unit tree) and end with a “1”.
3. Also parts of the code that represent valid sub-trees will also follow the same rule.
4. Any arbitrarily split left part of a valid binary tree code will at least have as many 0’s as 1’s. But the number of 1’s will always be lower than the number of 0’s.

Once you are trained to identify the sub-trees in the code, mostly through practice, it is possible to use an easier interpretation to the 1’s- they represent the number of sub-trees formed earlier that are children to the node represented by the preceding 0. This interpretation allows constructing trees from binary codes and hence is very useful for our purpose.

The procedure to encode any given tree to a string of binary numbers is as follows.

Start with the tree and an empty binary string.

1. Now traverse the tree bottom up, for every child of a given node append a 1 to the binary string.

2. After all the children sub trees are counted, append a zero to signify the parent itself.
3. Repeat the same procedure using the binary string thus produced with each child in the order they appear starting from the first child.

For leafs, no 1’s will be appended, just zero for the list itself.

3.3.3 Binary code tree enumeration algorithm

The Binary Code Tree Enumeration algorithm (BCTE) utilizes the binary representation and the binary decision tree approach to determine the entire set of trees with N nodes in the forest. The algorithm helps to reduce the complex task of identifying each individual tree arrangement for given size N to binary string processing. The method used is eliminating the prefixes that cannot produce valid tree codes down the binary decision tree.

The process involves a binary decision tree as shown in fig 23.

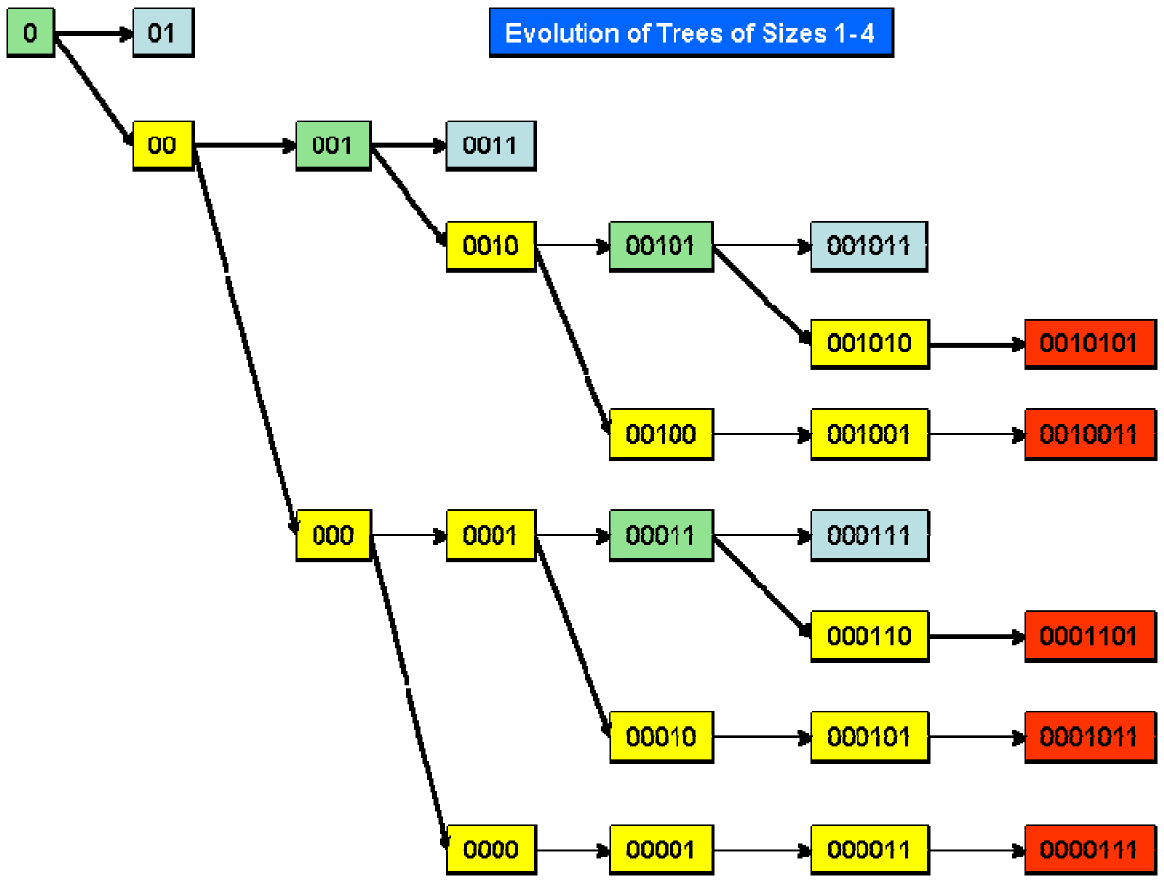


Figure 20: Binary decision tree for binary tree code generation

The figure shows binary decision tree depicting the tree code generation of binary tree codes for trees of size up to 4. The different colors indicate the type and status of node while the nodes are labeled with the prefixes of tree codes as we work down the tree. The red leaves indicate the terminal codes, which are valid codes for size = 4 trees.

At this point we define a few terms which will be used extensively in explaining this algorithm.

1. A ***forest*** is a sequence of valid subtrees with no root.
2. A valid ***subtree*** refers to the binary code for a tree of size, **M**,

where 1 <= **M** <= **N** (target size).

3. A ***leaf*** is the binary code for a subtree of size 1, namely, "**0**".
4. The ***trunk*** of a subtree is a substring of the form "$\mathbf{01^k}$", where k > 0, that occurs at the right end of a binary code, or is followed by a "0"; the zero of the trunk always denotes the parent of the subtree. The trunk defines a complete subtree if all digits to the left of the trunk form k complete subtrees.
5. **Terminal** is the term used for nodes that will be identified as binary codes that represent a tree of size N, where N is the maximum tree size and also the goal for the evolution tree being referenced.
6. **Dead** is the term used for nodes which are identified by the algorithm as non-expandable or dead leaves of the binary decision tree.
7. **Internal** is the term used to refer to all other nodes that occur between the root and the dead nodes or terminal nodes. These nodes may be valid codes for trees of size less than the goal size, but they are neither terminal nor dead.

-- Algorithm (N) --

Let p denote the current selected node – p must denote a node with status *new*.

```
If p.Terminal(N) Then   {  p.setStatus(Terminal); Output(p); goto (3); }
        else if p.Dead(N) Then  {  p.setStatus(Dead); goto (3); }
else{
```

```
            setStatus(Done);
            p.AddFirst();  //This creates a New child by appending a "0".
            p.AddLast();   //This  creates  a  New  child  by  appending  a  "1"
            p = p.FirstChild();
            goto (2);
        }
        For( q = p.Parent();  q !=  NULL;  q = q.Parent() )
              if( q.LastChild().Status() != New ) continue;
        else { p = q.LastChild();  //status == New
                    goto(2);
    }//else
Halt;      //All      binary      codes      have      been      enumerated.
Terminal(N){
            //1:  If the rightmost digit is "0" then, if N = 1 then return TRUE else
            return FALSE.
             //2:  If  Nzeros  <  N, return FALSE;
            //3:  If  Nsubtrees == 1 Then return TRUE else return FALSE.
            // Note:  Nsubtrees =  Nzeros - Nones
        }//Terminal

        Dead(N){
            //1: If Nzeros > N then return TRUE;
            //2: If Nzeros <= Nones then return TRUE; else return FALSE.
```

```
}//Dead


AddFirst(){
        //Create a new Node as the first child with the following parameters
        //1: Nsubtrees += 1;
        //2: Nzeros++
        //3: setStatus(New);
        //4: Code += “0”
        //5: No change to Nones
}//AddFirst
AddLast(){
        //Create a new Node as the first child with the following parameters
        //1: Nsubtrees—
        //2: No change to Nzeros
        //3: setStatus(New)
        //4: Code += “1”
        //5: Nones++
}
```

-- End of Algorithm--

Since, this algorithm has not been theoretically proven to function correctly; we attempt here to prove it. The algorithm in its multiple tests functioned flawlessly and produced a complete and correct list of binary tree codes for various sizes.

We begin by proving the validity of binary string rules we use to define a binary tree code.

Proof of Correctness

Lemma 1: Let T = R[S_1, S_2, …, S_n] denote a tree of size **N = Size(T)** with root node, R, and child forest S_1 … S_m, where m ≥ 0, and each S_k is itself a tree of the same form. When m = 0 (no children) T = R and N = 1. A binary code for T is a string, **Bin(T)**, of the form:

a) **0**, if m = 0 (N=1)

b) **Bin(S_1)Bin(S_2)…Bin(S_n)01^m,** if m > 0 (N > 1).

Then,

(1) Bin(T) is exactly 2N-1 in length.

(2) Zeros(Bin(T)) = Ones(Bin(T))+1

If Bin(T) = uv, for any non-null substring u, then Zeros(u) > Ones(u).

Proof of (1): By induction on N = Size(T). For N = 1, Bin(T) = **"0"** has length 1 = 2(1)-1. Assume (1) is true for all T where 1 ≤ Size(T) ≤ m. Consider a tree, T, where Size(T) = m+1. Since T has at least 2 nodes, we can write Bin(T) = Bin(S_1)Bin(S_2)…Bin(S_k)01^k, where k ≥ 1 is the number of child subtrees of the root, R. Now, for each j, where 1 ≤ j ≤ k, 1 ≤ Size(S_j) ≤ m and $m = \sum_{j=1}^{k} Size(S_j)$.

Thus by our induction hypothesis, Length(Bin(S_j)) = 2Size(S_j)-1. Therefore we have:

$$Length(Bin(T)) = 1 + k + \sum_{j=1}^{k}\left(2Size(S_j) - 1\right) = 1 + k - \sum_{j=1}^{k} 1 + 2\sum_{j=1}^{k} Size(S_j) = 1 + k - k + 2(m)$$
$$= 2m + 1.\ \text{But m} = \text{Size(T) - 1, so we have Length(Bin(T))} = \text{2(Size(T) - 1)} + 1 = \text{2Size(T) - 1.}$$

This completes the proof of (1).

Proof of (2): By induction on N = Size(T). For N = 1, Bin(T) = **"0"**, Zeros(T) = 1 and Ones(T) = 0. Assume (2) is true for all T where 1 ≤ Size(T) ≤ m. Consider a tree, T, where Size(T) = m+1. Since T has at least 2 nodes, we can write Bin(T) = Bin(S_1)Bin(S_2)…Bin(S_k)01^k, where k ≥ 1 is the number of child subtrees of the root, R. Now, for each j, 1 ≤ j ≤ k, 1 ≤ Size(S_j) ≤ m and $m = \sum_{j=1}^{k} Size(S_j)$. Thus by our induction hypothesis we have, Zeros(S_j) = Ones(S_j) + 1. Thus

$$Zeros(T) = 1 + \sum_{j=1}^{k} Zeros(S_j) = 1 + \sum_{j=1}^{k}\left(Ones(S_j) + 1\right) = 1 + k + \sum_{j=1}^{k} Ones(S_j) = 1 + Ones(T).$$

This concludes the proof of (2).

Proof of (3): For N = 1, Bin(T) = **"0"**. The only choice for **u** is **"0"**. Clearly Zeros(**u**) = 1 > Ones(**u**) = 0. Assume the statement is true for all trees, T, where Size(T) ≤ m. Consider T with Size(T) = m+1. Bin(T) = Bin(S_1)Bin(S_2)…Bin(S_k)01^k and consider the possible choices of **u**. If **u** cuts Bin(S_1), then since Size(S_1) ≤ m our induction hypothesis implies Zeros(**u**) > Ones(**u**). If **u** cuts at the boundary between Bin(S_i) and Bin(S_{i+1}) then Zeros(**u**) =

Sum($j \leq i$) { Zeros(S_j) }. But since Size(S_j) $\leq$ m our induction hypothesis implies Zeros(**u**) = Sum($j \leq i$) { Zeros(S_j) } > Sum($j \leq i$){ Ones(S_j)} = Ones(**u**). If u cuts in the middle of S_i for some $_i$ > 1, then **u** = Bin(S_1)…Bin(S_{i-1})**u’,** where **u’** is a prefix of Bin(S_i). Again, the induction hypothesis applied to Bin(S_j), j < i, and Bin(S_i) yields the desired result by following an argument similar to ones already given above. The last case to consider is where **u** cuts the trunk of Bin(T). By our induction hypothesis Zeros(Bin(Sj)) > Ones(Bin(Sj)) for each j. Thus Zeros(**u**) $\geq$ 1 + k + Sum(j$\leq$ k){Ones(Bin(Sj))}.

But Ones(**u**) = Sum(j$\leq$ k){Ones(Sj)} + p, where p $\leq$ k and u include p 1’s from the trunk of Bin(T). Thus we have Zeros(**u**) – Ones(**u**) $\geq$ 1 + k + Sum(j$\leq$ k) {Ones(Bin(Sj))} – Ones(**u**) = 1. It follows that Zeros(**u**) > Ones(**u**) and the proof is complete.

Observation. If B is any binary string that satisfies (2) of Lemma 1, then B must be of odd length. If Size(B) = 1 and B satisfies (3) of Lemma 1, then B = “0”.

If Size(B) > 1 and satisfies (1)-(3) of Lemma 1, then the suffix of B must be of the form “01^k”, for some k > 0. This follows because if

Lemma 2. Let B be a binary string. If B satisfies properties (2)-(3) of Lemma 1, then B = Bin(T) for some tree, T, of size N, where N = Zeros(B) and thus (1) of Lemma 1 also holds..

Proof. If B is any binary string that satisfies (2) of Lemma 1, then B must be of odd length. If Size(B) = 1 and B satisfies (3) of Lemma 1, then then B = “0”. If Size(B) > 1 and satisfies (2) and (3) of Lemma 1, then the suffix of B must be of the form "01^k", for some k > 0. Otherwise, if B = "u0" for some non-null u, then (2) implies Zeros(u) = Ones(u), a contradiction to (3). So, we can assume that B = "$u01^k$", for some substring u. Once again, (2) implies Size(u) ≥ Zeros(u) ≥ k. In fact, Zeros(u) = Ones(u) + k. In the rest of the proof, we write u = u[0] = x[0]y[0]z[0], where x[j] is a string over the alphabet {σ}. The symbol σ will count as if it were a single zero (**0**), but represents a valid binary code of some tree of size less than N, where Size(B) = 2N-1. y[j] is a string of the form $\sigma^{\mathbf{p}}01^{\mathbf{p}}$, for some **p** > 0; this is a valid binary code for a tree of the form R[S_1, S_2, … $S_{\mathbf{p}}$], where σ represents the binary code for S_i. z[j] is the remaining suffix of u[j].

If Ones(u) = 0, then u = 0^k and B is the binary code for a tree with k+1 nodes.

If Ones(u) > 0, then let $1^{\mathbf{p}}$ be the leftmost block of 1s. Then u = xyz, where Ones(x) = 0, y = $0^{\mathbf{p+1}}1^{\mathbf{p}}$ and Size(z) ≥ 0. Observe that y is the binary code of a tree with p nodes. Replace y by a single $\underline{\mathbf{0}}$ in u obtaining x$\underline{\mathbf{0}}$z. Replacing y by $\underline{\mathbf{0}}$ has the effect of reducing the number of zeros and ones in u by exactly **p**. Consequently, the relation, Zeros(u) = Ones(u) + k, still holds after the substitution. This process can be repeated until Ones(u) is depleted leaving u = $\mathbf{a}^{\mathbf{k}}$, where each **a** is a **0** or a $\underline{\mathbf{0}}$, and each $\underline{\mathbf{0}}$ represents the root of a subtree of size at least 2. The removal of each y substring decreases the length of u by 2**p**. The reduction leads to a binary code of the form: "$\mathbf{a}^{\mathbf{k}}\ 01^k$". This is the binary code of a tree with k sub trees. Thus B encodes a tree T with N nodes where Size(B) = 2N-1.

Theorem1. Algorithm 1 enumerates the binary codes of all trees of size N and halts in a configuration where the leaves of the genealogy are either dead or terminal. The terminal leaves represent the complete set of valid codes for trees of size N. The dead nodes that end with "1" cannot be the prefix of any valid tree code. Dead nodes that end with "0" represent a proper prefix of valid codes of some trees with size greater than N. Interior nodes either represent the binary code of some tree of size less than N, or the proper prefix of some code for a tree of size greater than or equal to N.

Proof. We prove by induction on depth(p) for interior node p that Lemma-1(3) always holds, and conditions (1) and (2) holds for interior nodes that are complete codes for trees of size less than N.

Case: depth(p) = 0.

In this case, p must be the root node of the evolutionary tree. Since we have assumed p is an interior node, we know that Terminal(p) and Dead(p) are both false. Thus, N > 1. But Bin(p) = "0" and Zeros(Bin(p)) = 1 > Ones(Bin(p)) = 0. Thus conditions (3), (2) and (1) hold for tree size = 1 = Zeros(Bin(p)).

Case: depth(p) = M+1.

Our induction hypothesis is that for all interior nodes, q, where depth(q) <= M, the statement of Theorem 1 holds. Let p denote an interior node where depth(p) = M+1 = Size(Bin(p))-1.

For p to be interior, p.Terminal(N) is false and p.Dead(N) is false. **"p.Terminal(N) is false"** implies:

a) Bin(p) ends in "0" and N > 1; or

b) Bin(p) ends in "1" and Zeros(Bin(p)) < N; or

c) Bin(p) ends in "1" and N = Zeros(Bin(p)) and

Zeros(Bin(p)) – Ones(Bin(p)) > 1.

Assume(a), then Dead(N) is false implies.

Zeros(Bin(p)) <= N and Zeros(Bin(p)) > Ones(Bin(p)). Clearly Bin(p) satisfies (3) of Lemma 1 for u = Bin(p) and v = λ. Since Bin(p) ends in "0" by assumption, then Bin(p) = "w0", for some string w = Bin(q), where q is the immediate parent of p. Since depth(q) = M, by our induction hypothesis, q must be an interior node for which the statement of Theorem 1 holds. Thus Bin(q) is the binary code of a tree of size N' < N, or Bin(q) is the binary code of a tree of size N' >= N.

If Bin(q) = w is the binary code of a tree of size N' < N, then it satisfies all conditions of Lemma 1, and in particular condition (3). Thus Bin(p) satisfies condition (3) of Lemma 1. Lemma 1 also implies that N' = Zeros(w) = Ones(w) + 1. It follows that Zeros(Bin(p)) = Ones(w) + 2 = Ones(Bin(p))+2. Thus "Bin(p)1" is a binary string satisfying conditions (1),(2) and (3) of Lemma 1 and by Lemma 2 is a valid tree code for some tree of size N = Zeros(Bin(p)). We conclude that Bin(p) is a proper prefix of the code for some tree of size N >= Zeros(Bin(p)).

If Bin(q) = w is a proper prefix of a code for a tree of size N' >= N, then it too satisfies condition (3) of Lemma 1, but does not satisfy condition (2). Even so, this is sufficient to ensure by our induction hypothesis that Lemma 1(3) holds for Bin(p) and that Zeros(Bin(p)) > Ones(Bin(p)) + 1. Thus "Bin(p)1^k", for k = Zeros(Bin(p))-Ones(Bin(p))-1 >= 1 satisfies all conditions of Lemma 1 and thus by Lemma 2 is a valid code for some tree of size Zeros(Bin(p)). Thus Bin(p) is the proper prefix of a valid binary code of some tree of size N >= Zeros(Bin(p)).

Assume (b), then Dead(N) is false implies:

Bin(p) = "w1", where w = Bin(q) and q is the immediate parent of p, and Zeros(Bin(p)) < N and Zeros(Bin(p)) > Ones(Bin(p)). An argument similar to that given for the previous case establishes that Bin(p) satisfies condition (3) of Lemma 1, and that Bin(p) is still the proper prefix of a binary code for some tree of size N > Zeros(Bin(p)).

Assume (c), then Dead(N) is false implies:

Bin(p) = "w1", where w = Bin(q) and q is the immediate parent of p, and Zeros(Bin(p)) = N and Zeros(Bin(p)) > Ones(Bin(p)). An argument similar to that given for the previous case establishes that Bin(p) satisfies condition (3) of Lemma 1, and that Bin(p) is still the proper prefix of a binary code for some tree of size N >= Zeros(Bin(p)).

This concludes the proof of Theorem 1 for interior nodes of the evolutionary tree.

Now suppose p.Terminal(N) is true. Then one of the following conditions must be true about p:

(d) N = 1 and Bin(p) = “0”;

(e) N > 1, Bin(p) = “w1” and N <= Zeros(Bin(p)) and Zeros(Bin(p)) = Ones(Bin(p))+1.

Assume case (d). For N = 1 there is only one tree of that size and its binary code is “0”.

In this case the algorithm halts by marking the root node as Terminal and the root becomes the only node of the evolutionary tree. Thus Theorem 1 holds true for this case.

Assume case (e). In this case N = Zeros(Bin(p)). If N > Zeros(Bin(p)), then N > Zeros(w) = Zeros(Bin(p)). But w must be Bin(q) where q is the immediate parent of p and must be an interior node. So depth(q) = M and by our induction hypothesis, Bin(q) satisfies condition Lemma 1(3). We conclude that Bin(p) also satisfies Lemma 1(3).

But since q is interior, q.Terminal(N) is false. This implies Zeros(w) <= N, a contradiction. Thus N = Zeros(Bin(p)) and Bin(p) satisfies conditions (3), (2) and (1) of Lemma 1. Thus by Lemma 2, Bin(p) is the binary code of a tree of size Zeros(Bin(p)) = N.

If p denotes a node for which p.Dead(N) is true, then we can conclude the following facts about p: p.Terminal(N) is false and (Zeros(Bin(p)) > N or Zeros(Bin(p)) <= Ones(Bin(p)).

The Tree code generator was developed using the BCTE algorithm to generate all distinct binary codes to represent all possible trees arrangements of a particular size. Please note that the binary code length follows the formula (2N-1) for trees of size N. Therefore the code generator aims to produce valid binary strings of length (2N-1).

The collection of these strings is non-duplicating and represents the complete set of trees of size N.

3.3.4 Binary code to Tree converter

Once we had the Binary tree code generator, we were able to generate complete sets of binary codes for trees of given size. However, these were just the codes representing tree arrangements and hence were not relevant to our research. To be meaningful, these codes had to be converted into real trees, trees that could be measured for properties. We designed an interesting solution to this problem. The Binary code to Tree converter, is a very short piece of code that utilizes a stack of trees and parses a binary tree code and generate the tree it represents with default data wherever needed. Once drawn a tree is available for measurement as long as needed.

3.3.5 Forest generation – Linear code method

Putting together the binary code generator and the binary code to tree converter, we started producing and measuring trees and forests of trees of specific sizes. The trees generated in this linear code method were generated on a per tree basis. Once a tree is completely converted, its properties were measured. They were recorded as both individual tree properties as well as aggregated in the forest properties for the particular size of trees. The tree would then be discarded.

This method is very stable and uses resources linearly. However, there is a lot of duplication of effort. For example, while generating trees of size 6 the sub trees of size 2, 3, 4 and 5 are recomputed every time they occur in codes for size 6. Considering that almost all large trees are comprised of smaller sub trees, this re-computation causes enormous wastage of computing power and time due to duplicated effort.

3.3.6 Tree generation – Lookup method (Linear programming)

Another method of generating trees is by lookup. In this method, we generate larger trees as set combinations of smaller trees. For example a tree of size 5 can be represented as {(1,1,1,1), (1,1,2), (1, 3), (2,2)}. Each set combination represents the collection of sizes of the sub-trees constituting a size 5 tree. Each element in the set can be replaced with each of the distinct tree of the size indicated by the number. Thus, the set 1, 2 and 4 represent unique trees; however in set 3, the second element can be represented by two trees of size 3 and thus produces 3 trees. Different trees can be generated by permuting the order in which the children appear.

The tool we developed using this approach generated smaller trees before generating larger trees. Smaller trees are easier to generate and information from these small trees is organized and utilized in formation of larger trees. In particular, the shapes or shapes of the smaller trees can be utilized to generate larger trees. This approach is very efficient tree generation wise as there is almost no over head due to duplicated effort however it has the overhead of permuting through various tree sets and maintaining all trees the smaller than N, where N is the size of the tree being generated. It is also more complex to implement and requires a huge amount of memory capacity, especially for larger tree sizes.

In our experience, this method was nonlinear and needed increasingly higher amounts of computer time and memory.

3.4 Time Complexity analysis

3.4.1 Cohen's Algorithm

For purpose of analysis we consider a d-ary complete tree with N nodes and height h.

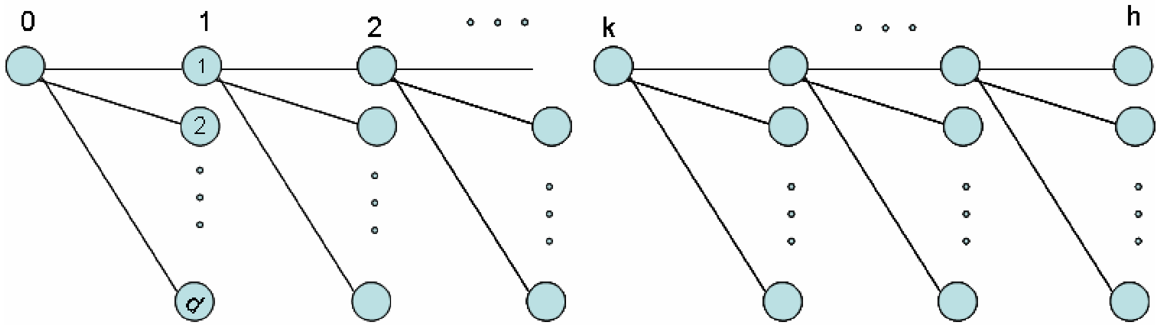


Figure 21: *d*-ary tree with N nodes and height h

At any arbitrary level k, we introduce an interactive tree operation. The time complexity of the algorithm to perform the interactive operation is of specific interest in this research.

For Cohen's algorithm, the cost of implementing the change consists of two parts, cost to implement the change in specific sub-tree and cost to propagate the change towards the root.

The upper limit on the number of steps in the bounding structure in Cohen's algorithm is 1. Therefore to implement the change will need comparison of the change among the forest of children of the parent where the change occurs will be d(1) = d. The cost to propagate the change towards the root will need recomputing the bounding rectangle for each progressive step towards the root. Each of this operation will need d comparisons.

Therefore, the total time complexity of the algorithm can be given as

$$T_{cohen} = d + \sum_{j=1}^{k} d = d(1+k)$$

The worst case for this algorithm is when the change occurs at a leaf, that is k= h.

$O(Tcohen) = O(d(1+h)) = O(d+\log_d N) = O(\log_d N)$

3.4.2 Moen's Algorithm

Again, consider the d-ary complete tree with N nodes and height h as done in previous analysis. We perform an interactive operation on a node at level k. For example a new sub-tree is added at level k.

The effort of interactive operation in a Tree drawing using Moen's algorithm consists of two parts - preparation and actual operation.

In cases involving removal of sub-trees like cut and delete, sequence of operations is altered to - actual operation and cleanup.

The preparation and cleanup actions are similar in the aspect of operation. The purpose of either is to confirm the tree to the algorithm's rule of common x start coordinate for all nodes at same level in the tree.

In the preparation phase, the min-vector of the forest rooted at k-1 is compared with the min-vector of the new sub tree being inserted. This involves (h-k+1) comparisons to determine the new min vector for the new forest. This new min is then propagated to the parent who in turn propagates it to its own parent till it reaches the root. The root then reverses the propagation to all its children including the forest being affected causing the min-vectors throughout the tree to be changed to reflect the change occurring in K’s forest.

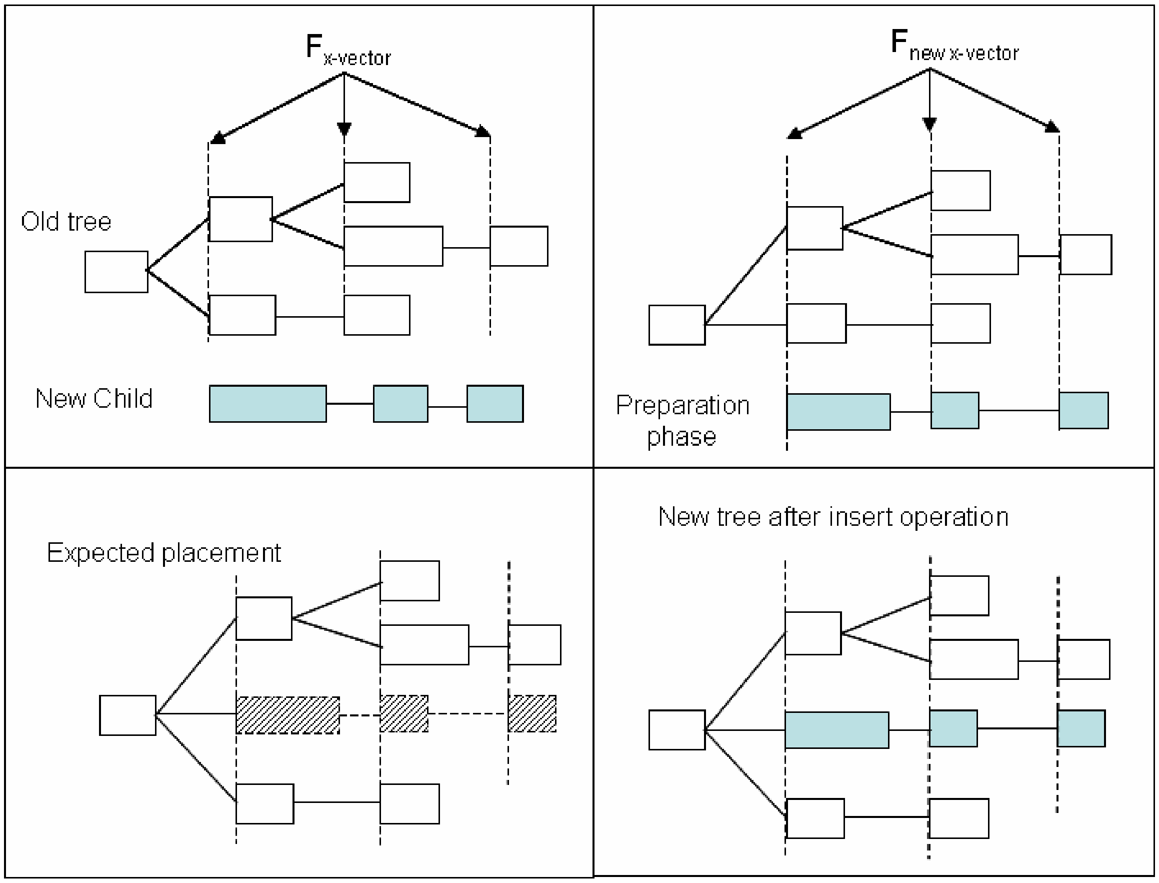


Figure 22: Moen's algorithm interactive operation

This upper limit on the cost of propagation to the root could be $\log_d(N)$ -1. The down propagation covers every node in the tree, therefore in any case N nodes are visited and their shapes updated.

Actual operation involves comparison of the newly generated shapes of the sub-trees in forest of K and the shapes of the new sub-tree. The comparison cost is dependent on the number of distinct steps in the shape. According to Moen's algorithm this number is (h-k+1) + (h-k). The first term indicates the shape component outlining the nodes themselves, while the second term is the profile for the connector space boundary. Hence the cost of placing the new sub-tree after comparing both the upper shape and the lower shape for d siblings in the forest of K is 2d(2(h-k)-1). Once the new sub-tree is placed vertically with respect to its siblings the forest shape generated during placement is used to generate the new shape for the parent K. This, in turn, needs the forest of parent of K to be rearranged and so on. This secondary propagation of change towards the root occurs k times and in the worst case logd(N)-1 times (k= h-1 as h= logd(N)).

The total cost of the actual operation phase can be represented by summing all these costs as

T(N) = T_{prep}(N) + T_{act}(N), where T_{prep}(N) is the cost of preparation and Tact(N) is the cost of actual operation.

$$T_{prep}(N) = N + (h - k + 1) = N + \log_d N_h - \log_d N_k + 1.$$

$$T_{act}(N) = \sum_{j=1}^{k}(2(d-1)\times(2(h-j)-1)+2)$$

$$T_{act}(N) = 2(d-1)\sum_{j=1}^{k}(2(h-j)-1)+2+\sum_{j=1}^{k}2$$

$$= 2(d-1)\sum_{j=1}^{k} 2(h-j) - 2(d-1)\sum_{j=1}^{k} 1 + \sum_{j=1}^{k} 2$$

$$= 2(d-1)\sum_{j=1}^{k} 2(h-j) - 2(d-1)k + 2(k)$$

$$= 4(d-1)\sum_{j=1}^{k} (h-j) - (2d-4)(k)$$

$$= 4(d-1)(h\sum_{j=1}^{k} 1 - \sum_{j=1}^{k} j)) - (2d-4)(k)$$

$$= 4(d-1)\left[hk - \frac{k(k+1)}{2}\right] - (2d-4)(k)$$

$$= 4(d-1)hk - 2(d-1)k(k+1) - (2d-4)(k)$$

But, $h= \log_d N_h$, $k = \log_d N_k$

$$= 4(d-1)\log_d N_h \log_d N_k - 2(d-1)\log_d N_k(\log_d N_k + 1) - (2d-4)(\log_d N_k)$$

The worst case occurs when the operation is performed on a leaf. Since, k=h the T_{act} becomes $2(d - 1)(\log_d N_h)^2 – 2d(\log_d N_h) + 4(\log_d N_h) = O[(\log_d N)^2]$.

However, the overall time complexity an interactive operation using Moen's algorithm becomes $O[T_{prep} + T_{act}] = O[N + (\log_d N_h)^2] = O[N]$

3.4.3 Observations

The algorithm implementations for WB-algorithm and Cohen's algorithm were easy and smooth. The reason being there was no specific data structure framework that was binding the implementation. Such was not the case with Moen's algorithm. The strong data structure definition and interrelated intricacies of the method were severe obstacles. Also, the algorithm is expressed as fragments of code rather than a formal definition.

Moen's Approximation

We were not able to recreate the Moen's algorithm code in its entirety and use the algorithm definition as standard. The WB-centered algorithm follows the same placement method in case of uniform sized nodes and hence an approximation of Moen's layout can be created from the WB centered node. The dimensions of node areas in our experiment are unit and the connector spacing is also unit. Thus any node area can be divided into icon space and connector space. Since both spaces are unit, half-unit hext of the glyph will be allocated to connector spacing and another half to the icon itself. We can approximate the connector spacing as allocated by Moen's algorithm, by adding an connector to connect all positive transistions in the lower shape and all negative transistions in the upper connector to form a close Moen approximation.

Figures below demonstrate how WB-centered can be used to approximate the Moen's Algorithm.

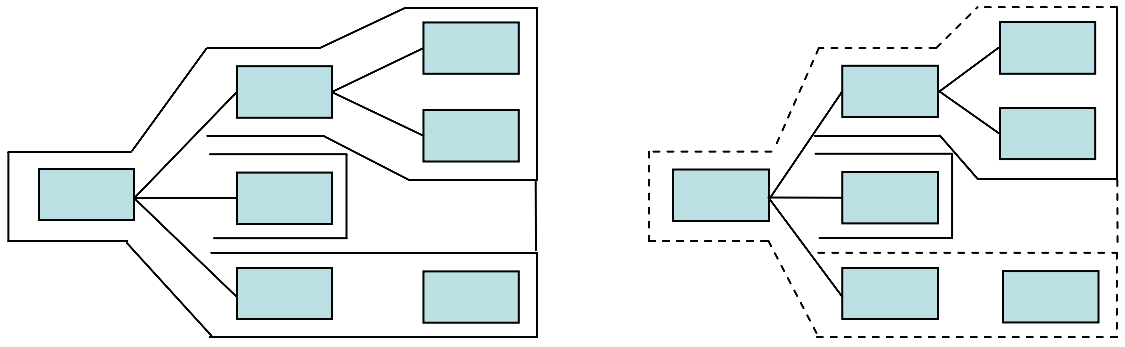
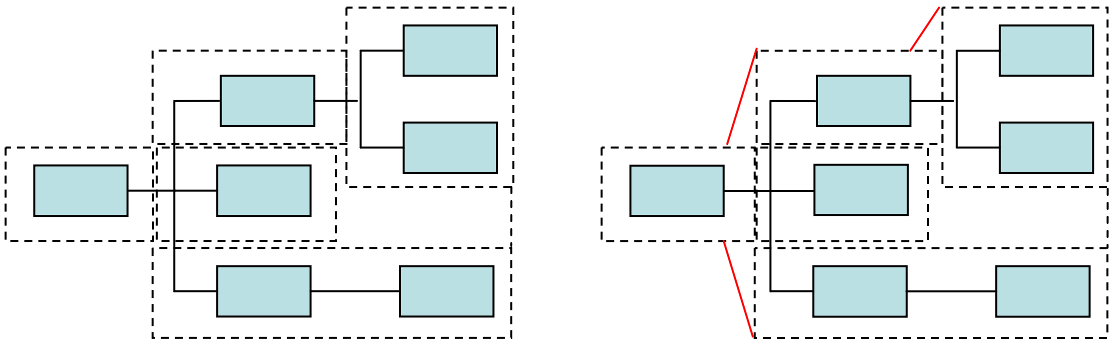


Figure 23: Moen's Layout- with and without added connector length



(b)

Figure 24: Moen's Layout, WB-Centered and Moen Approximation using WB-Centered

The WB- algorithms do not dictate an additional connector length, all connector placement and spacing is managed within the glyph area. Moen’s Algorithm does provide an additional connector length and it does so by using a computed value with some arbitrary constant reference. For close comparison, this constant if set to 0, will eliminate the independent connector lengths. With this property, the layout will be identical to WB layout, except for the non-orthogonal components in the shape functions of Moen’s algorithm.

Our approximation method attempts to consider the drawing area contained within these non-orthogonal components in the shape functions by producing a function we will refer to as – Moen’s approximation. Please note the adjusted shape functions are identical to Moen’s shape functions with added triangular regions at each positive or negative transition. Moen’s approximation is a constant that determines the contribution of the connector spacing towards the tidy area and tidy waste of the drawing layout.

Moen's Approximation (MA)= $\sum_{i=1}^{h} \left|\Delta y_i\right| * 0.5$,

where, k = number of steps in the respective shapes. Δy_i = Δy in step in shape at step i.

And, $\Delta y_i = \Delta y_{i,}$

where - all $\Delta y_i > 0$ for lower shapes and for all $\Delta y_i < 0$ for upper shapes.

For all other Δy_i, $\Delta y_i = 0$.

In general, the tidy area for a Moen layout can be approximated as

Moen's Tidy Area = WB-centered tidy area + MA * connector length.

Moen's Tidy Waste = WB-centered tidy waste + MA * connector length.

The data was collected in to understand the properties of the trees as a function of their size on the specific parameters of interest. Table below summarizes the data gathered.

3.5 Data Collected.

All units involved are FDUs accept where ratios are concerned.

Table 1: Block area (W*H) - Parent aligned with first child

Tree Size(N)	Average	Max.
1	1	*1*
2	2	*2*
3	3.5	*4*
4	5.6	*6*
5	7.85714	*9*
6	10.2381	*12*
7	12.75	*16*
8	15.4009	*20*
9	18.179	*25*
10	21.0753	*30*
11	24.0804	*36*
12	27.1881	*42*
13	30.3943	*49*
14	33.6616	*56*

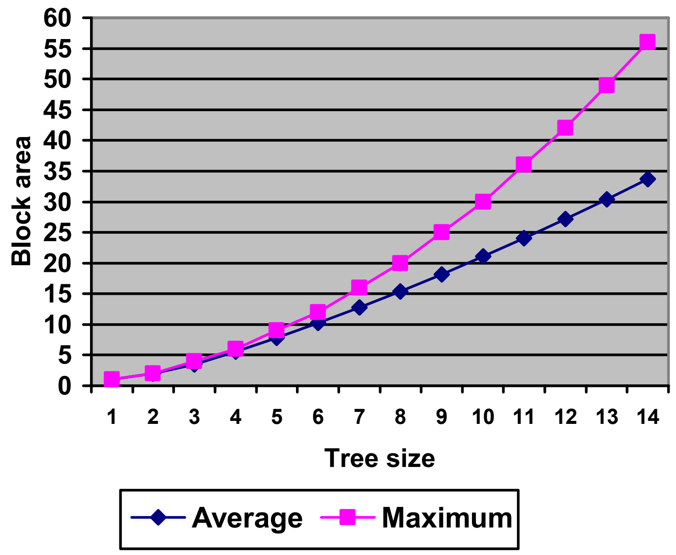


Table 2: Block Area (W*H) - Parent centered over forest of children

Tree Size(N)	Average	Max.
1	*1*	*1*
2	*2*	*2*
3	*3*	*3*
4	*4.6*	*5*
5	*6.57143*	*7*
6	*8.83333*	*10*
7	*11.3712*	*13*
8	*14.1702*	*17*
9	*17.2126*	*21*
10	*20.4823*	*26*
11	*23.9662*	*31*
12	*27.6533*	*37*
13	*31.5335*	*43*
14	*35.5945*	*50*

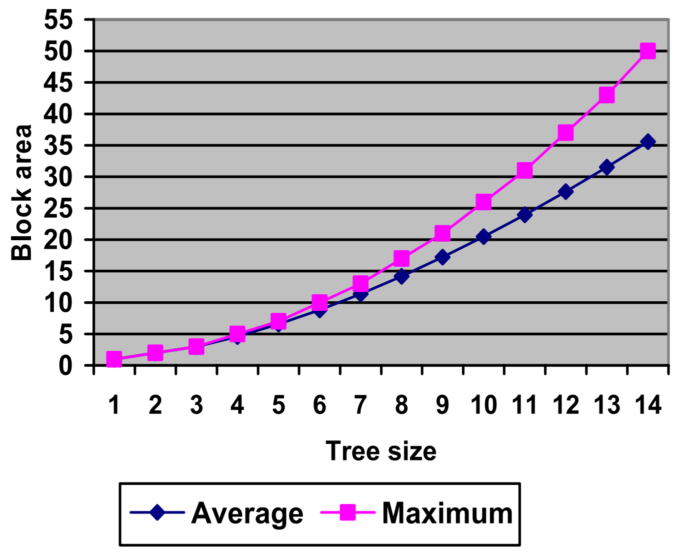


Table 3: Drawing Vext - Parent aligned with first child

First child aligned		
Tree Size(N)	Average	Max
1	1	1
2	1	1
3	1.5	2
4	2	3
5	2.42857	4
6	2.83333	5
7	3.2197	6
8	3.59674	7
9	3.96433	8
10	4.32498	9
11	4.67915	10
12	5.02781	11
13	5.37079	12
14	5.7099	13

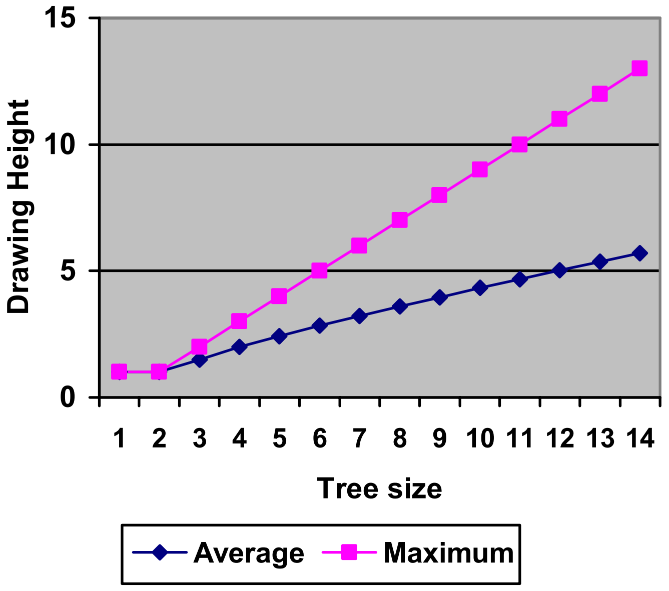


Table 4: Drawing Vext - Parent centered over forest of children

Centered		
Tree Size(N)	Average	Max
1	*1*	*1*
2	*1*	*2*
3	*1.5*	*2*
4	*2*	*3*
5	*2.42857*	*4*
6	*2.80952*	*5*
7	*3.15909*	*6*
8	*3.48835*	*7*
9	*3.8042*	*8*
10	*4.10911*	*9*
11	*4.40522*	*10*
12	*4.69402*	*11*
13	*4.97619*	*12*
14	*5.25119*	*13*

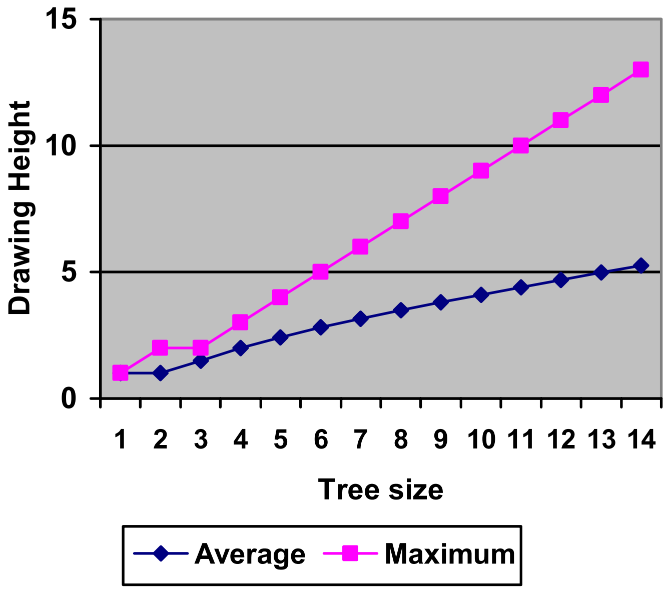


Table 5: Tidy Area – Parent aligned with first child

Tree Size(N)	First child aligned		
	Average	Min	Max
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6.04762	6	7
7	7.12879	7	9
8	8.24475	8	11
9	9.39231	9	13
10	10.5722	10	16
11	11.7839	11	19
12	13.0273	12	22
13	14.3028	13	25
14	15.6079	14	29

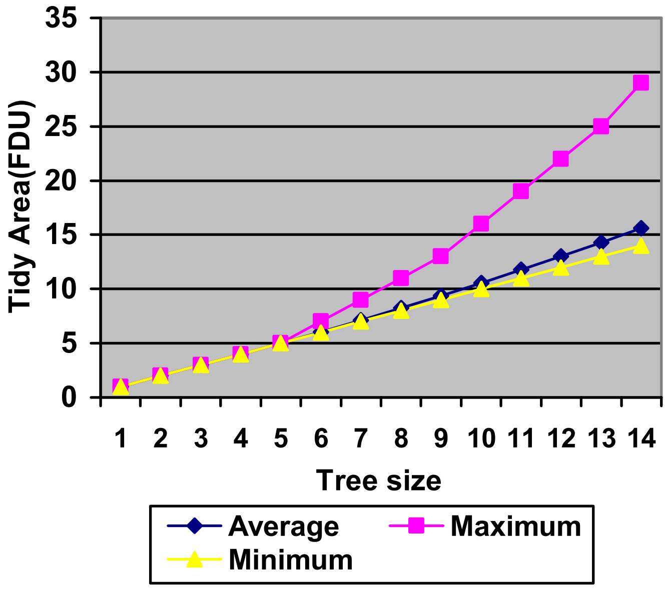


Table 6: Tidy area - Parent centered over forest of children

Tree Size(N)	*Centered*		
	Average	*Min*	*Max*
1	*1*	*1*	*1*
2	*2*	*2*	*2*
3	*3*	*3*	*3*
4	*4*	*4*	*4*
5	*5*	*5*	*5*
6	*6.04762*	*6*	*7*
7	*7.12879*	*7*	*9*
8	*8.24009*	*8*	*11*
9	*9.38113*	*9*	*13*
10	*10.5498*	*10*	*16*
11	*11.7466*	*11*	*19*
12	*12.9712*	*12*	*22*
13	*14.2205*	*13*	*25*
14	*15.4861*	*14*	*29*

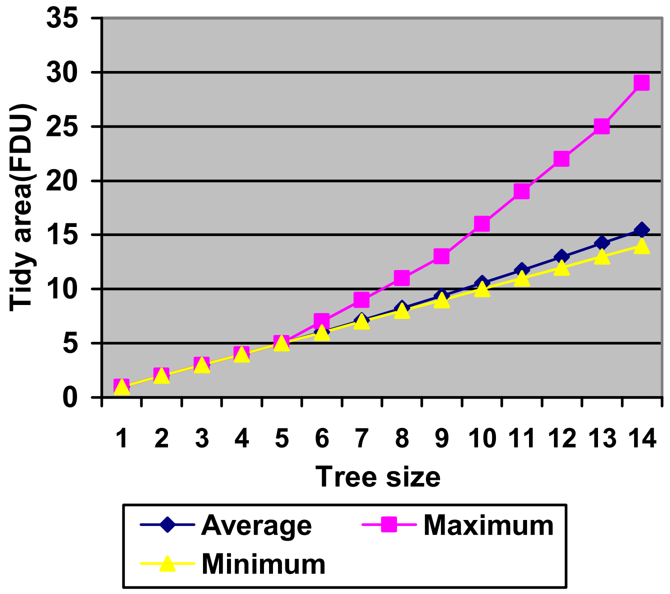


Table 7: Tidy Waste – Parent aligned with first child

Tree Size(N)	First child aligned	
	Average	Max
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0.047619	1
7	0.128788	2
8	0.244755	3
9	0.392307	4
10	0.572192	6
11	0.783935	8
12	1.02717	10
13	1.30103	12
14	1.60622	15

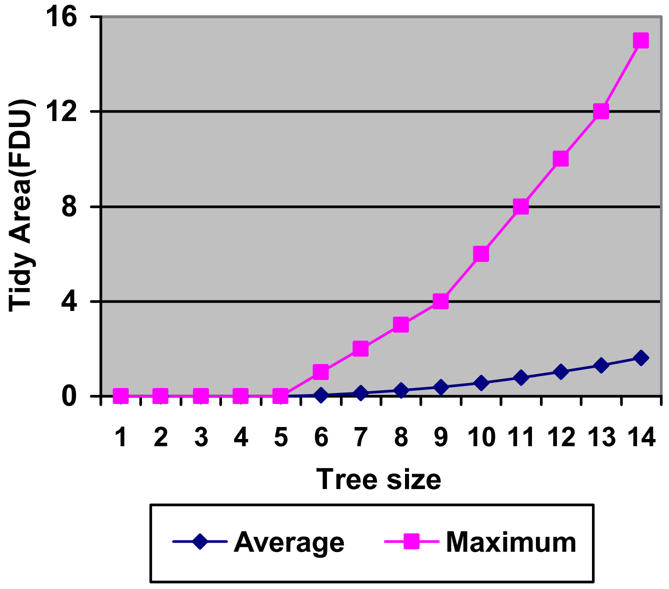


Table 8: Tidy waste - Parent centered over forest of children

Tree Size(N)	*Centered*	
	Average	*Max*
1	0	*1*
2	0	*2*
3	0	*3*
4	0	*4*
5	0	*5*
6	0.047619	*7*
7	0.128788	*9*
8	0.240093	*11*
9	0.381119	*13*
10	0.549774	*16*
11	0.746666	*19*
12	0.970722	*22*
13	1.22099	*25*
14	1.49556	*29*

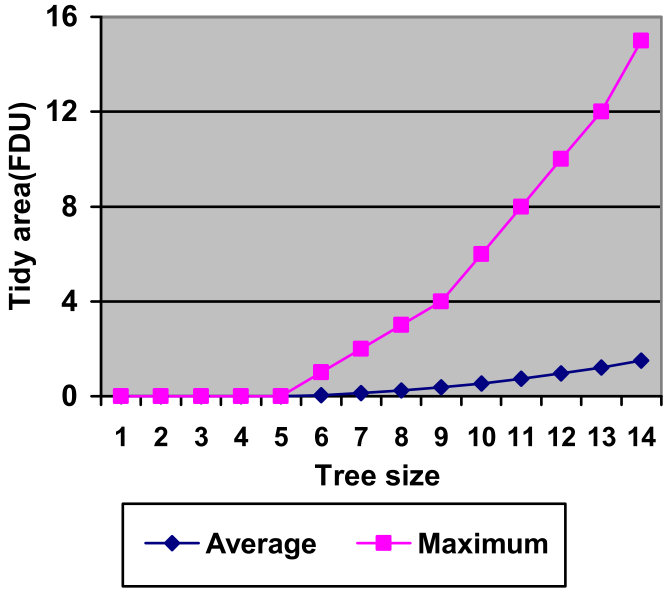


Table 9: Upper Shape Step Count – Parent aligned with first child

Upper Shape Steps (Min = 1)		
Tree Size(N)	Average	Max.
1	1	1
2	1	1
3	1	1
4	1.2	2
5	1.35714	2
6	1.5	3
7	1.63636	3
8	1.7669	4
9	1.89161	4
10	2.01091	5
11	2.12542	5
12	2.2357	6
13	2.34255	6
14	2.44311	6

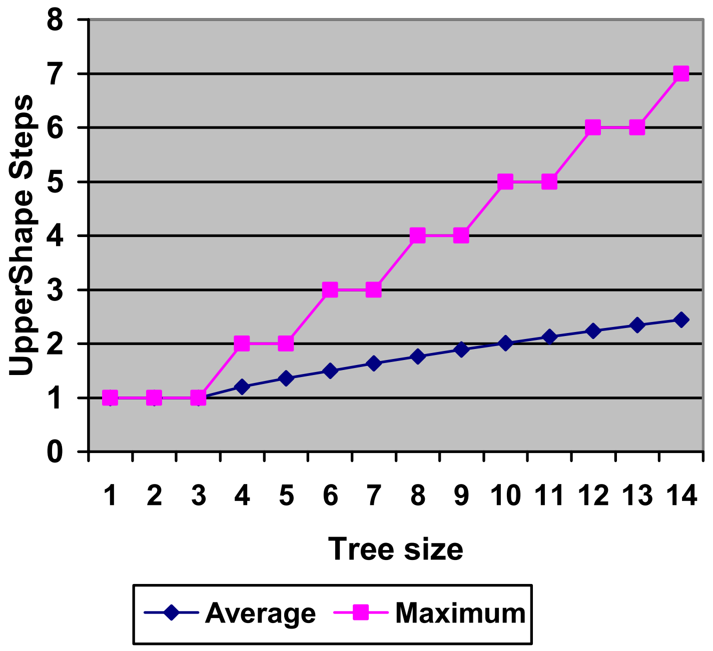


Table 10: Upper Shape Step Count – Parent centered over forest of children

Tree Size(N)	Average	Max.
1	*1*	*1*
2	*1*	*1*
3	*1.5*	*2*
4	*2*	*3*
5	*2.35714*	*3*
6	*2.64286*	*4*
7	*2.90909*	*5*
8	*3.15385*	*5*
9	*3.38252*	*6*
10	*3.59873*	*7*
11	*3.80377*	*7*
12	*3.99941*	*8*
13	*4.18754*	*9*
14	*4.37642*	*9*

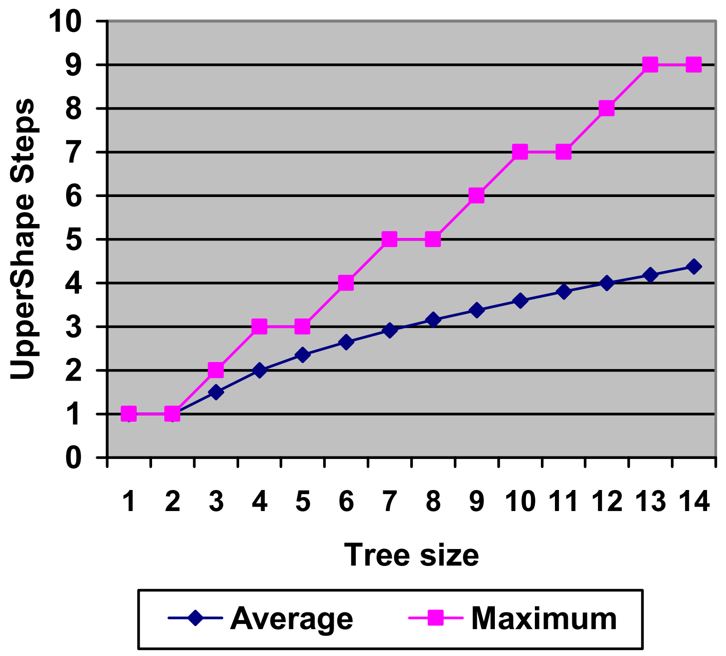


Table 11: Lower Shape step count– Parent aligned with first child

Lower Shape Steps		
Tree Size(N)	Average	Max
1	1	1
2	1	1
3	1.5	2
4	2	3
5	2.28571	3
6	2.57143	4
7	2.81818	5
8	3.05361	5
9	3.27133	6
10	3.47779	7
11	3.67335	7
12	3.86018	8
13	4.03883	9
14	4.20526	9

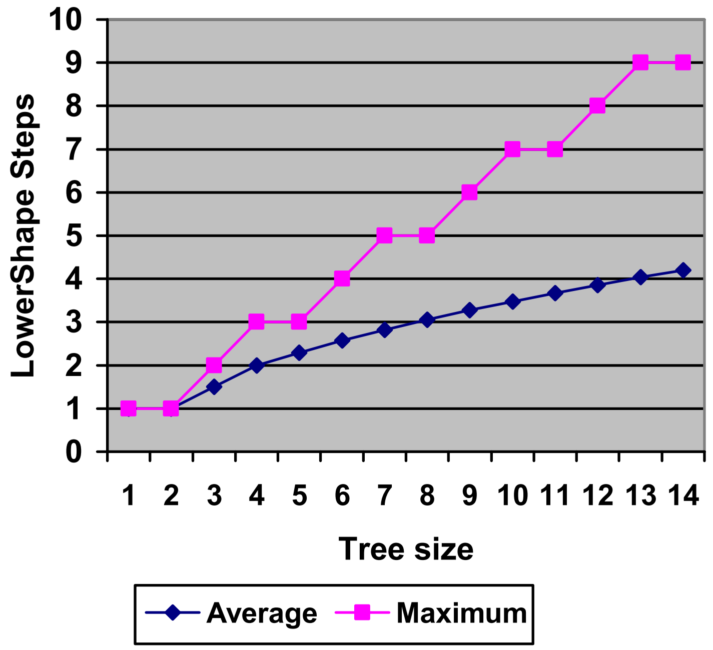


Table 12 - Lower Shape Step Count – Parent centered over forest of children

	Centered	
Tree Size(N)	Average	Max
1	*1*	*1*
2	*1*	*1*
3	*1.5*	*2*
4	*2*	*3*
5	*2.35714*	*3*
6	*2.64286*	*4*
7	*2.90909*	*5*
8	*3.15385*	*5*
9	*3.38252*	*6*
10	*3.59873*	*7*
11	*3.80377*	*7*
12	*3.99969*	*8*
13	*4.18767*	*9*
14	*4.3738*	*9*

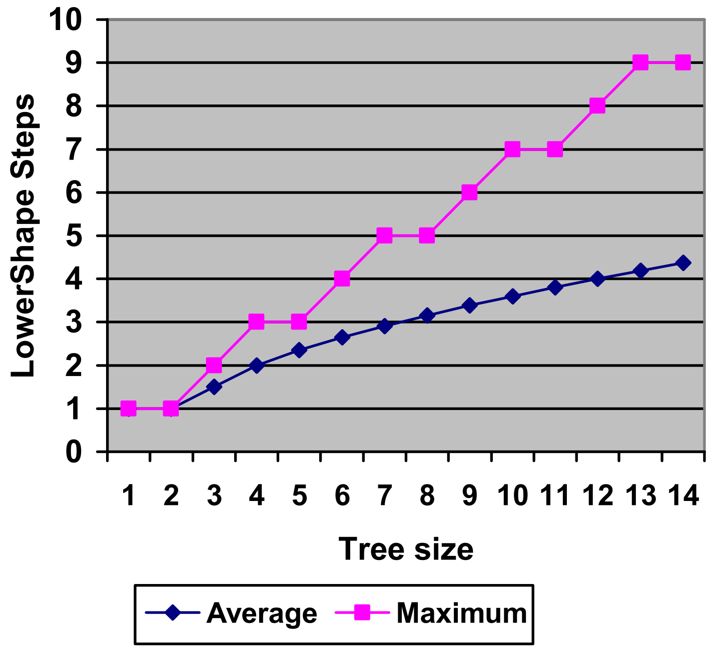


Table 13 - Average Child to Parent Ratio (b) for forest (Max = N-1, Min = 1)

Tree Size(N)	Child: Parent
1	0
2	1
3	1.5
4	1.4
5	1.64286
6	1.33333
7	1.64394
8	1.35664
9	1.54336
10	1.40395
11	1.54067
12	1.35203
13	1.55673
14	1.37484

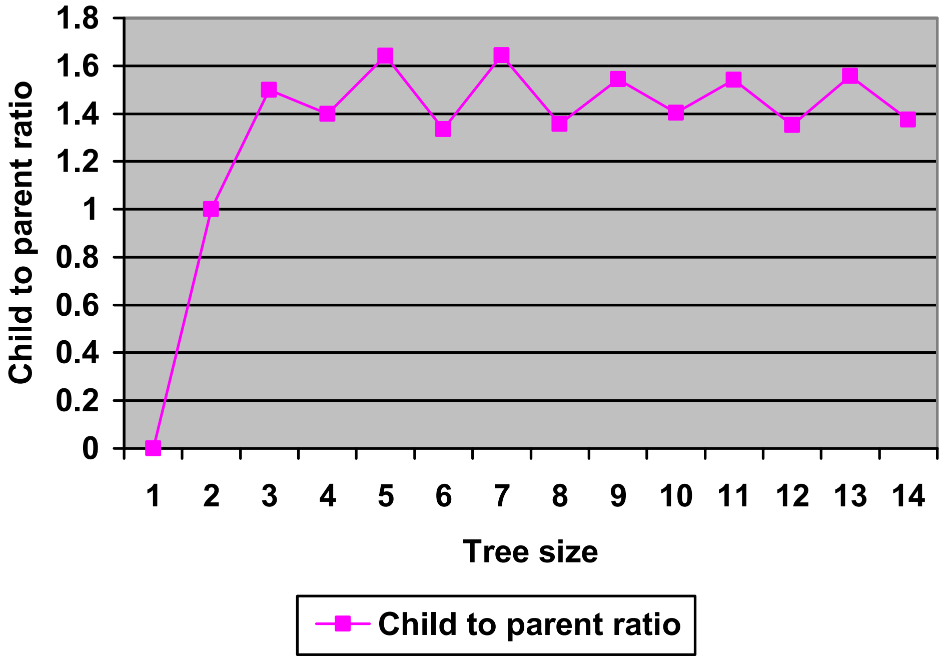


Table 14- Aspect ratio of Tree drawings

Tree Size(N)	First child aligned parent			Centered parent		
	Average	Min	Max	Average	Min	Max
1	1	1	1	*1*	*1*	*1*
2	0.5	0.5	0.5	*0.5*	*0.5*	*0.5*
3	0.666667	0.333333	1	*0.666667*	*0.333333*	*1*
4	0.75	0.25	1.5	*0.75*	*0.25*	*1.5*
5	0.788095	0.2	2	*0.788095*	*0.2*	*2*
6	0.822619	0.166667	2.5	*0.814683*	*0.166667*	*2.5*
7	0.853355	0.142857	3	*0.836941*	*0.142857*	*3*
8	0.882248	0.125	3.5	*0.855966*	*0.125*	*3.5*
9	0.908401	0.111111	4	*0.873022*	*0.111111*	*4*
10	0.93279	0.1	4.5	*0.888431*	*0.1*	*4.5*
11	0.955626	0.090909	5	*0.902733*	*0.0909091*	*5*
12	0.977309	0.083333	5.5	*0.916191*	*0.0833333*	*5.5*
13	0.997983	0.076923	6	*0.929003*	*0.0769231*	*6*
14	1.01827	0.071429	6.5	*0.941339*	*0.0714286*	*6.5*

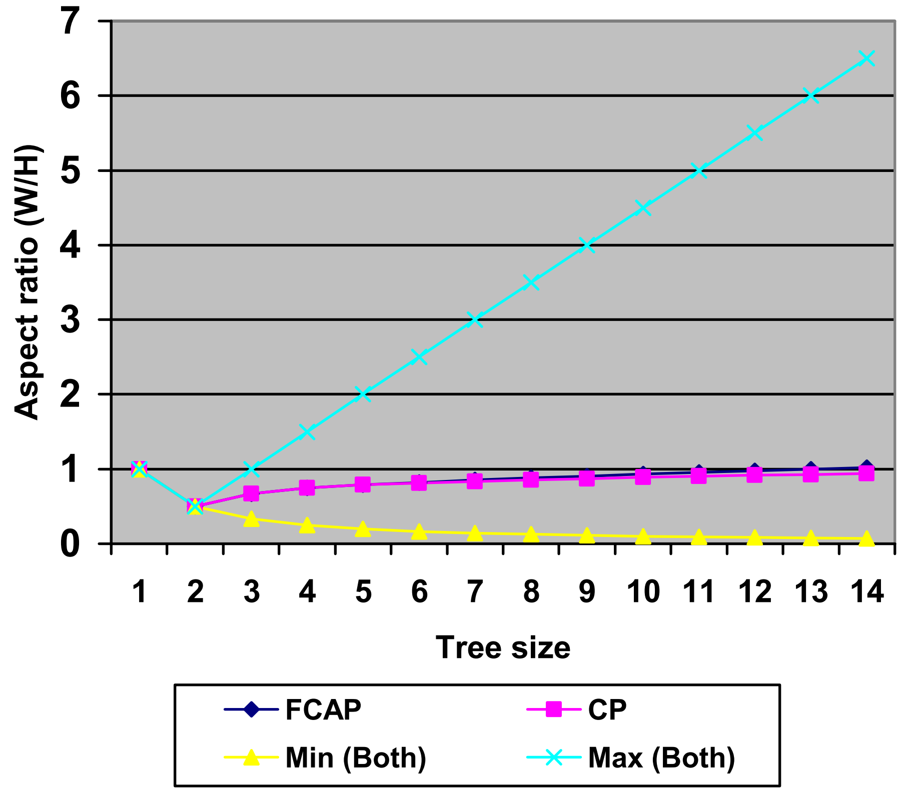


Table 15: Tidy area comparison – Moen vs. Parent centered over forest of children

Tree Size(N)	FCAP	*Centered*	
	WB	*WB-C*	*Moen*
1	1	*1*	*1*
2	2	*2*	*2*
3	3	*3*	*3.125*
4	4	*4*	*4.25*
5	5	*5*	*5.357143*
6	6.04762	*6.04762*	*6.505954*
7	7.12879	*7.12879*	*7.68561*
8	8.24475	*8.24009*	*8.89219*
9	9.39231	*9.38113*	*10.12624*
10	10.5722	*10.5498*	*11.38598*
11	11.7839	*11.7466*	*12.67223*
12	13.0273	*12.9712*	*13.98488*
13	14.3028	*14.2205*	*15.32094*
14	15.6079	*15.4861*	*16.67242*

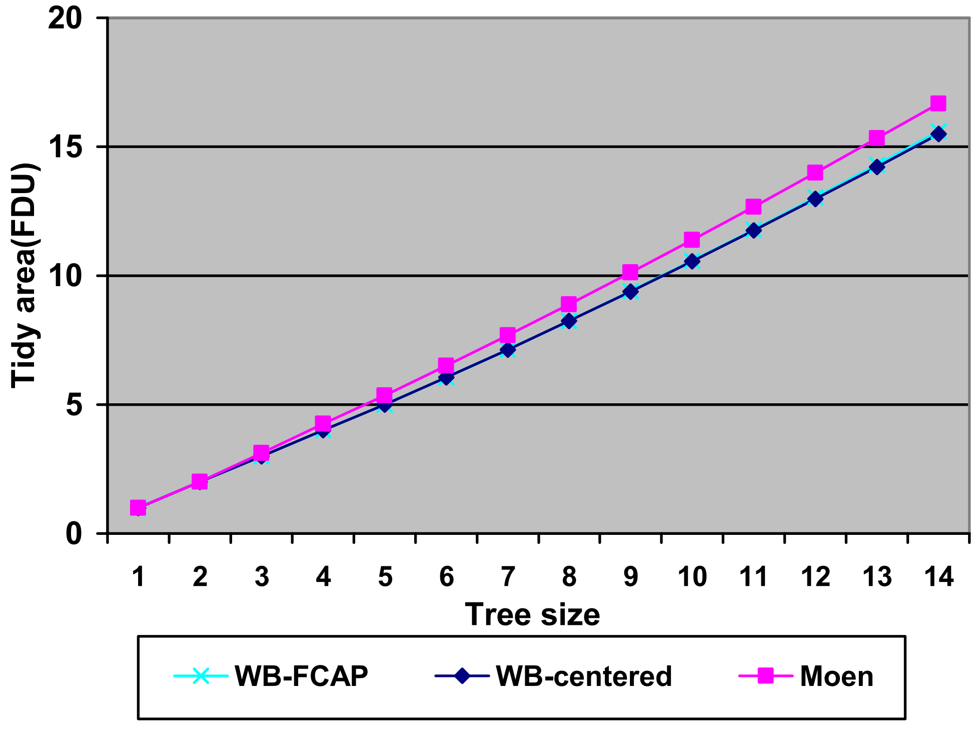
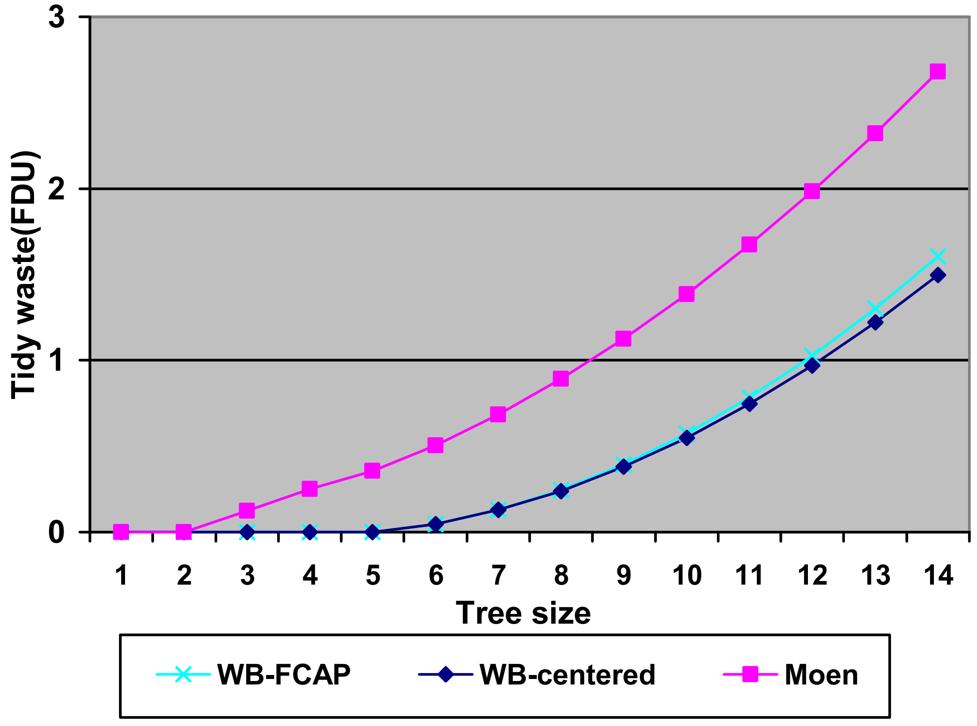


Table 16: Tidy waste comparison – Moen vs. Parent centered over forest of children

Tree Size(N)	*FCAP*	*Centered*	
	WB	***WB-C***	***Moen***
1	0	0	*0*
2	0	0	*0*
3	0	0	*0.125*
4	0	0	*0.25*
5	0	0	*0.357143*
6	0.047619	0.047619	*0.505953*
7	0.128788	0.128788	*0.685608*
8	0.244755	0.240093	*0.892193*
9	0.392307	0.381119	*1.126224*
10	0.572192	0.549774	*1.385954*
11	0.783935	0.746666	*1.672291*
12	1.02717	0.970722	*1.984397*
13	1.30103	1.22099	*2.32143*
14	1.60622	1.49556	*2.68188*



3.6 Discussion

Workman-Bernard algorithm utilizes the efficiency of discrete step functions for its bounding shapes. This simplifies comparison between bounding shapes. However, Moen's algorithm forces uniform start coordinates for each level in the tree. This also makes comparison simpler, unlike WB's unaligned levels which could complicate the comparisons. However, Moen's functions, as implemented in the paper, do not exploit this advantage, thus using the same point-to-point comparison as WB. Apart from the cost of comparison, response time in Moen's algorithm is increased by its requirement to align all levels in the tree. Taking O(N) time, Moen algorithm's performance remains almost linear, while WB performance improves with N due to its lower time complexity of $O(\log^2 N)$.

From the data gathered during this research we can perceive that the number of steps involved in the shapes (upper or lower) tend to grow at approx $\log_{1.4} N$. Therefore the $\log^2 N$ will be a dominant term in the time complexity of either algorithm for trees of size 3 or more. The *d* average degree of the tree (children per node) seems to vary between 1.3 and 1.7. This number will be the base of the log terms and hence is very significant. Due to limit on data available we are not able to identify the behavior of this curve for larger sized trees, however, we expect it to converge at some size of trees. However, that size maybe too large for practical requirements or practical tree drawings.

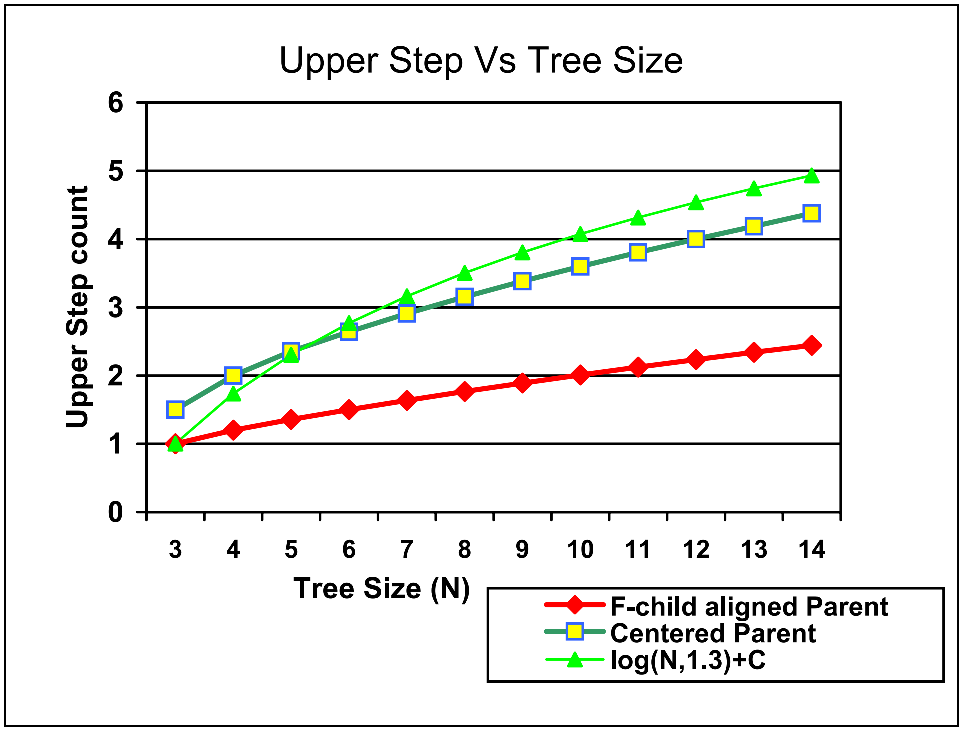


Figure 25: Comparison - Length of Shapes as function of N

It can be noticed that the problem size increases very fast per increment in tree size. The block area of the drawings using both the centered parent and the first aligned parent is almost the same for small sized trees that we consider here. It would be more helpful if we can extend the size of the data set we have by adding larger sized trees. Consideration of non-uniform sized nodes would improve the applicability of the experiment with practical implementations and applications. However, it is very difficult to do so due to the enormity of the problem. It is probably a problem for more programming oriented research in field of parallel or high performance computing. The data we have collected does not necessarily portray the actual performance of the algorithm however it is indicative of it, one might surmise that this research more or less highlights the basic characteristics of these algorithms.

4 CONCLUSION

The research conducted and analysis performed indicates that Moen's algorithm produces slightly more compact trees than the first child centered implementation of Workman-Bernard Algorithm. The cost of interactive updates in Moen have a time complexity of $O(N+\log_d^2 N)$, which is very high compared to Workman-Bernard algorithm's $O(\log_d^2 N)$. It is also possible to improve the compaction properties of Workman-Bernard algorithm by opting to place the root centered on the forest of its children. In case this strategy is used, Workman-Bernard's centered parent placement produces better results compaction wise as compared to Moen's algorithm. Moen's algorithm almost always will occupy more tidy area as compared to such a Workman Bernard configuration.

For application in practical interactive applications involving tree drawings where frequent changes to the tree arrangement are expected, Workman-Bernard algorithm is better suited as compared to Moen's algorithm as per our analysis.

For more precise results, additional research may be conducted. However, the flexibility and very nature of practical trees makes this an extremely difficult problem to be completely resolved by experimental methodology.

The research can be forwarded by addressing the main hurdles we faced here. The performance of the algorithms can be better understood by expanding the data set as collected by this research. The computational requirements are so intensive, that new methods to perform these measurements need to be improved in order to achieve a larger size of data, e.g. Parallel programming etc.

There is always a scope for non-uniformed sized nodes in tree arrangements, which will resemble more with real life usage of Drawing layout algorithms.

REFERENCES

1. David Workman, Margaret Bernard and Steven Pothoven, “An Incremental Editor for Dynamic Hierarchical Drawing of Trees”, International Conference on Computational Science, 2004:
2. Robert F Cohen and et. al “Dynamic Graph Drawing: Trees, Series-Parallel Digraphs and Planar ST-Digraphs”, Society for Industrial and Applied Mathematics, 1995.
3. Sven Moen, “Drawing Dynamic Trees”, IEEE Trans. Software Engineering, 1990.
4. E. M. Reingold and J. S. Tilford, “Tidier Drawings of trees”, IEEE Trans. Software Eng., 1981.
5. D. Workman, “GRASP: An Interactive System for Graphic Specification of Software,” Proceedings of the 1978 Southeast Regional ACM Conference, March 1978.
6. S. Pothoven, “A Portable Class of Widgets for Grammar-Driven Graph Transformations”, Computer Science Masters Thesis, University of Central Florida, 1996.
7. D. Workman, “Algorithms Analysis of Tree Layout,” research notes, School of EE and CS, University of Central Florida.
8. C. Wetherell and A. Shannon, “Tidy Drawings of Trees”, IEEE Trans. Software Eng. Sept 1979.