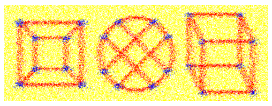

Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 12, no. 2, pp. 131–195 (2008)

Grid Drawings of Binary Trees: An Experimental Study

Adrian Rusu[1] *Confesor Santiago*[2]

[1]Department of Computer Science, Rowan University, Glassboro, New Jersey, USA
[2]Department of Electrical and Computer Engineering, Rowan University, Glassboro, New Jersey, USA

Abstract

In this paper we consider the class of binary trees and present the results of a comprehensive experimental study on the four most representative algorithms for drawing trees, one for each of the following tree-drawing approaches: **Separation-Based**, **Path-based**, **Level-based**, and **Ringed Circular Layout**. We establish a simpler, more intuitive format for storing binary trees in files and create a large suite of randomly-generated, unbalanced, complete, AVL, Fibonacci, and molecular combinatory binary trees of various sizes. Our study is therefore conducted on randomly-generated, unbalanced, and AVL binary trees with between 100 and 50,000 nodes, on Fibonacci trees T_n for $n = 1, 2, ..., 45, 46$ (143 to 46,367 nodes), on complete binary trees of size $2^n - 1$ for $n = 7, 8, ..., 15, 16$ (127 to 65,535 nodes), and on molecular combinatory binary trees with between 133 and 50,005 nodes. Our study yields 70 charts comparing the performance of the drawing algorithms with respect to ten quality measures, namely **Area**, **Aspect Ratio**, **Size**, **Total Edge Length**, **Average Edge Length**, **Maximum Edge Length**, **Uniform Edge Length**, **Angular Resolution**, **Closest Leaf**, and **Farthest Leaf**.

None of the algorithms has been found to be the best in all categories. This observation leads us to create an adaptive system that determines the type of a binary tree and then selects an algorithm to draw the tree depending upon the specified quality measures. Currently, our adaptive tree drawing system recognizes all six types of binary trees and all ten measures included in our experimental study. Under our settings, our adaptive tree drawing system outperforms any system using a single binary tree drawing algorithm.

Submitted:	Reviewed:	Revised:	Accepted:	Final:
April 2007	June 2007	August 2007	October 2007	November 2007

Published: June 2008

Article type: Regular Paper

Communicated by: S. Kobourov

E-mail addresses: rusu@rowan.edu (Adrian Rusu) santia09@students.rowan.edu (Confesor Santiago)

1 Introduction

Trees are ubiquitous data-structures, arising in a variety of applications such as Software Engineering (class and module hierarchies), Business Administration (organization charts), Knowledge Representation (isa hierarchies), and Web-site Design and Visualization (structure of a Web-site).

Visualizing a tree can enhance a user's ability in understanding its structure. Hence, a lot of research has been done on visualizing trees, which has produced a plethora of tree-drawing algorithms (See for example, [8, 9, 10, 11, 13, 16, 15, 14, 22, 25, 26, 32, 33, 35, 34]). The majority of these algorithms have been developed with the primary target of minimizing the area of the drawing, so, in addition to their practical evaluation on area, it is of interest to evaluate how these algorithms perform on other important aesthetics.

Several experimental studies for drawing graphs are available (See for example, [6, 3, 4, 5, 17, 18, 19, 36]). However, we are not aware of any experimental study done to evaluate the practical performance of tree-drawing algorithms. Given the importance of trees, and the large amount of research that has been done on developing techniques to visualize them, we believe that this is a big omission. As a first step, in this paper, we present an experimental study of some well-known algorithms for drawing binary trees. These algorithms represent some of the *distinct* approaches that have been used to draw binary trees without distorting or occluding the information.

A *binary* tree is one where each node has at most two children. In contrast to graphs, every tree accepts a *planar drawing*, i.e. without any crossings. Therefore, most tree-drawing algorithms achieve this aesthetic. A *straight-line* drawing has each edge drawn as a single line-segment. Straight-line drawings are considered more aesthetically pleasing than polyline drawings.

The issue of *resolution* of a drawing has been extensively studied, motivated by the finite resolution of physical rendering devices. The resolution of a drawing is defined as the minimum distance between two vertices. The quality measures **Area**, **Aspect Ratio**, **Size**, **Total Edge Length**, **Average Edge Length**, **Maximum Edge Length**, **Uniform Edge Length**, **Angular Resolution**, **Closest Leaf**, and **Farthest Leaf** of a drawing depend on its resolution, hence two drawings can be compared for these measures only if they have the same resolution. *Grid-based* algorithms, i.e. algorithms that place all the nodes of a drawing at integer coordinates, guarantee a minimum of one unit distance separation between nodes in the final drawings, and allow the drawings to be displayed in a display surface, such as a computer screen, without any distortions due to truncation and rounding-off errors.

A drawing of a tree T has the *subtree separation* property [8] if, for any two node-disjoint subtrees of T, the enclosing rectangles of the drawings of the two subtrees do not overlap with each other. Drawings with the subtree separation property are more aesthetically pleasing than those without the subtree separation property. The subtree separation property also allows for a focus+context style [31] rendering of the drawing, so that if the tree has too many nodes to fit in the given drawing area, then the subtrees closer to focus can be shown in

detail, whereas those further away from the focus can be contracted and simply shown as filled-in rectangles.

All algorithms in our experimental study produce planar straight-line grid drawings and exhibit the subtree separation property.

This work is comprised of an experimental study, which originally appeared in an abbreviated form in [29], and an adaptive tree drawing system, which originally appeared in [28]. The contributions of this work can be summarized as follows:

- We have developed a general experimental setting for comparing the practical performance of drawing algorithms for binary trees. Our setting consists of (*i*) a simpler, more intuitive format for storing binary trees in files; (*ii*) save/load routines for generating binary trees to files and for uploading binary trees from files, respectively; (*iii*) a large suite of randomly-generated, unbalanced, complete, AVL, Fibonacci, and molecular combinatory binary trees of various sizes; (*iv*) ten quality measures: area, aspect ratio, size, total edge length, average edge length, maximum edge length, uniform edge length, angular resolution, closest leaf, and farthest leaf.

- Within our experimental setting, we have performed a comparative study of four representative algorithms for planar straight-line grid drawings of binary trees, one for each of the following *distinct* approaches: separation-based algorithm by Garg and Rusu [16], path-based algorithm by Chan et al. [8], level-based algorithm by Reingold and Tilford [25], and ringed circular layout algorithm by Teoh and Ma [33]. As the specific algorithms chosen are intended to be representative of their respective approaches, we expect the results to generally apply to other algorithms using the same approach.

- Our comparison highlights how more than twenty years of research in this field have produced increasingly better algorithms. Our investigations include some interesting findings:

  - A contradiction to the popular belief [20] that, in practice, Reingold-Tilford algorithm should be generally accepted as the method of choice for drawing binary trees. Even though this algorithm achieves some important aesthetics, it scores worse in comparison to the other chosen algorithms for almost all ten aesthetics considered in our study.
  - The intuition that low average edge length and area go together is contradicted in only one case.
  - The intuitions that average edge length and maximum edge length, uniform edge length and total edge length [36], and short maximum edge length and close farthest leaf go together are contradicted for unbalanced binary trees.

  - The performance of a drawing algorithm on a tree-type is not a good predictor of the performance of the same algorithm on other tree-types: some of the algorithms perform best on a tree-type, and worst on other tree-types.
  - For three of the seven types of trees considered, the algorithm with the best theoretical worst-case bound produces worse area in practice than algorithms with worse theoretical worst-case bounds, or algorithms for which no theoretical bounds are available.
  - With regards to area, of the four algorithms studied, three perform best on different types of trees.
  - With regards to aspect ratio, of the four algorithms studied, three perform well on trees of different types and sizes.
  - Not all algorithms studied perform best on complete binary trees even though they have one of the simplest tree structures.
  - The level-based algorithm of Reingold-Tilford [25] produces much worse aspect ratios than algorithms designed using other approaches.
  - The path-based algorithm of Chan et al. [8] tends to construct drawings with better area at the expense of worse aspect ratio.

- Using the results of our experimental study we have designed an adaptive tree drawing system comprised of all four algorithms we experimented. Since the performance of an algorithm with respect to a specific quality measure was found to be dependent on the type of binary tree, our system analyzes the tree to classify it as a specific type and then selects an algorithm to draw it with respect to the user-specified quality measures.

The rest of the paper is organized as follows. The four algorithms being compared are described in Section 2. Details on the experimental setting are given in Section 3. In Section 4, we summarize our experimental results, and perform a comparative analysis on each chosen aesthetic of the performance of the four algorithms. In Section 5, we present our adaptive tree drawing system. Conclusion and future work are discussed in Section 6.

2 The Drawing Algorithms Under Evaluation

We have tested four different algorithms for producing planar straight-line grid drawings of binary trees. The four algorithms can be classified into four categories on the basis of their approach to constructing drawings:

- **Separation-Based**: In the *Separation-Based Approach*, a divide-and-conquer strategy is used to recursively construct a drawing of the tree, by performing the following actions at each recursive step:

- *Find a Separator Edge or a Separator Node*: A *separator edge (node)* of a tree T with $degree(T) = d$ is an edge (node), which, if removed, divides T into at most d smaller, partial trees. Every tree contains such an edge or a node [15, 35]. In the first step, these algorithms find a separator edge or a separator node.
- *Divide Tree*: Divide the tree into several partial trees by removing at most two nodes and their incident edges from it (including the separator edge or the separator node) (See Figure 1(a)).
- *Assign Aspect Ratios*: Pre-assign a desirable aspect ratio to each partial tree.
- *Draw Partial Trees*: Recursively construct a drawing of each partial tree using its pre-assigned aspect ratio.
- *Compose Drawings*: Arrange the drawings of the partial trees, and draw the nodes and edges, that were removed from the tree to divide it, such that the drawing of the tree thus obtained is a planar straight-line grid drawing. These drawings of partial trees are arranged either one next to the other (horizontal composition) as in Figure 1(b), or one above the other (vertical composition) as in Figure 1(c).

Several separation-based algorithms have been designed [13, 16, 15, 30]. Even though both the algorithms of [13] and [16] achieve the worst-case theoretical bound of $O(n)$ area, the algorithm of [13], being a top-down algorithm, always constructs the worst-case drawing. Being algorithms developed for drawing general trees, [15] and [30] have not been considered. We have therefore chosen to evaluate the $O(n)$-area bottom-up algorithm of [16] (we call it *Separation*). For our study, we have used the same implementation as the one used by Garg and Rusu in [16].

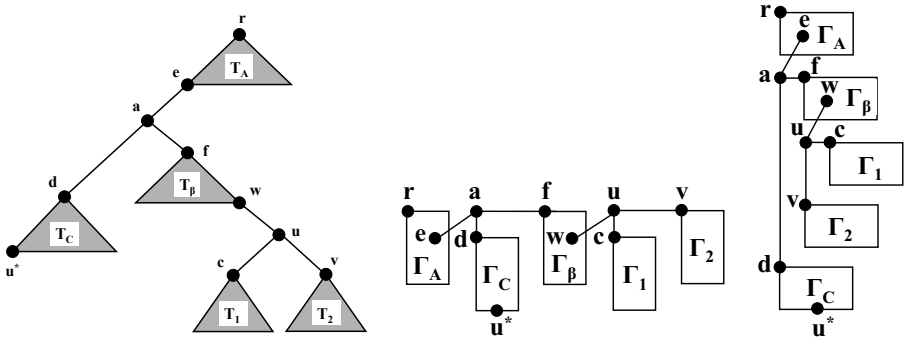


Figure 1: Separation-based approach [16]: (a) General case when the separator edge (node) is not on the leftmost path. (b) Horizontal composition. (c) Vertical composition.

- **Path-Based**: The *Path-Based Approach* uses a recursive winding paradigm as follows: first lay down a small chain of nodes from left to right until near a distinguished node v, and then recursively lay out the subtrees rooted at the children of v in the opposite direction.

  Several path-based algorithms have been designed [8, 14, 32].

  For our study, we have implemented the $O(n \log\log n)$-area algorithm developed by Chan et al. [8] (we call it *Path*). The reasons we have chosen this algorithm for our study are that it defines the majority of path-based techniques, produces the best worst-case theoretical bound on area for path-based algorithms (near optimal), and provides the user some control over the aspect ratio without sacrificing area. An illustration of *Path*'s drawing techniques is shown in Figure 2.

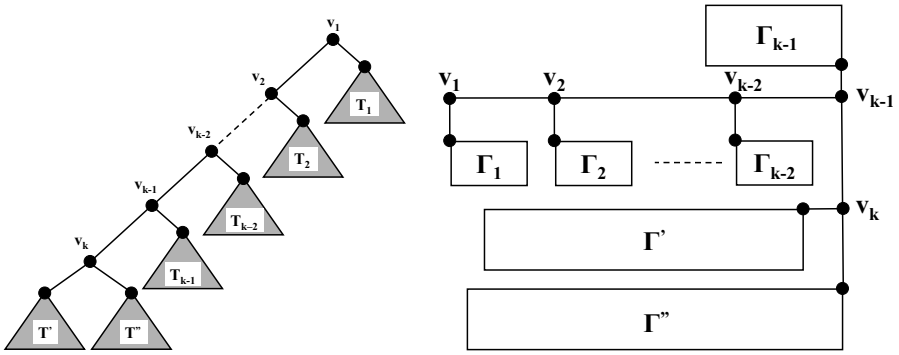


Figure 2: Path-based approach [8]: (a) General case: the tree is divided into subtrees $T_1, T_2, \ldots, T_{k-2}, T_{k-1}, T', T''$. (b) Non-upward composition.

- **Level-Based**: The *Level-Based Approach* is characterized by the fact that in the drawings produced, the nodes at the same distance from the root are horizontally aligned [7, 25, 37]. Since the algorithms of [7] and [37] do not exhibit the subtree separation property, for our study we have implemented the widely-used recursive algorithm developed by Reingold and Tilford [25] (we call it *Level*). This algorithm uses the following steps: draw the subtree rooted at the left child, draw the subtree rooted at the right child, place the drawings of the subtrees at horizontal distance 1, and place the root one level above and halfway between the children. If there is only one child, place the root at horizontal distance 1 from the child. An illustrative diagram of how *Level* places each node is shown in Figure 3.

- **Ringed Circular Layout**: The algorithms based on the *Ringed Circular Layout Approach* place a node and all its children in a circle [9, 22, 26, 33]. For our study, we have developed and implemented a binary tree

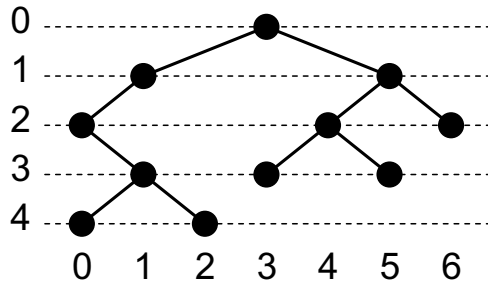


Figure 3: Grid diagram illustrating an example of drawing a tree with *Level*.

adaptation of the algorithm developed by Teoh and Ma [33], which was designed for general trees and is an improvement over the other Rings-based algorithms (we call it *Rings*). In the original algorithm, equal-sized circles corresponding to children are placed in concentric rings inside of the parent circle, around its center, thus trying to minimize the space wasted inside of the interior of the parent circle (see Figure 4). Since we only consider binary trees, we have developed *Rings* to take advantage of the basic properties of a binary tree. *Rings* places the children of a node in either the same vertical or horizontal channel, starting with the same horizontal channel at the root (depth 0), and alternates between vertical and horizontal channel placement for every following depth in the tree. In addition, the length of the edge connecting a subtree to its parent is set to $depth(subtree(v))+1$, where $depth(subtree(v))$ is the depth of the subtree rooted at node v. This ensures that enough space is made available to draw the rest of the subtree, which is consistent with other rings-based algorithms.

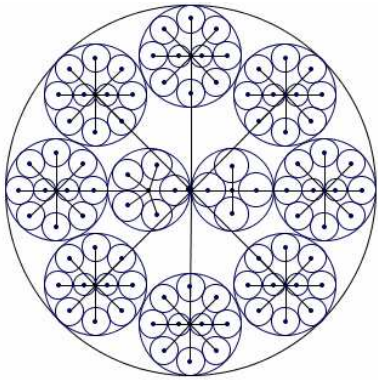


Figure 4: In *Rings*, the outer rings contain the larger subtrees, and the interior rings contain the smaller subtrees. For binary trees, nodes are always placed orthogonally, on the opposite sides of the root.

3 Experimental Setting

Our experimental setting consists of (*i*) a simpler, more intuitive format for storing binary trees in files; (*ii*) save/load routines for generating binary trees to files and for uploading binary trees from files, respectively; (*iii*) a large suite of randomly-generated, unbalanced, complete, AVL, Fibonacci, and molecular combinatory binary trees of various sizes; (*iv*) ten quality measures: area, aspect ratio, size, total edge length, average edge length, maximum edge length, uniform edge length, angular resolution, closest leaf, and farthest leaf.

3.1 Input File Format

Since trees have a simpler structure than graphs, we introduce a simpler, more intuitive format for storing binary trees in files. Each line in the input file represents a node, its left, and its right children, in this order, separated from each other by one space: node leftChild rightChild, where node is the key that uniquely identifies the node in the tree, leftChild is the key for the left child of node, or # if node has no left child, and rightChild is the key for the right child of node, or # if node has no right child. The following restriction applies to all the nodes, except the root of the tree: a node must occur as a child for another node before being itself defined.

For example, assume we have a binary tree defined using a preorder traversal as follows: 0, 1, 3, 4, 2, 5. This tree would be represented in its corresponding graph file format as follows:

0 —- 1;
0 —- 2;
2 —- 5;
1 —- 3;
1 —- 4;
3;
5;
4;

In our new format, this tree may be represented in its corresponding file in any of the following two ways:

0 1 2	or	0 1 2
1 3 4		2 5 #
2 5 #		1 3 4
3 # #		3 # #
4 # #		5 # #
5 # #		4 # #

Since the node with key 3 has not occurred as a child of any node before it was defined as having no children of its own, the following is not a proper representation of the tree:

0 1 2
2 5 #
3 # #

4 # #
1 3 4
5 # #

We have implemented in C++ simple load and save routines for transferring binary trees between the computer memory and files, using the format described in this subsection. The source code for the save and load routines can be downloaded at: http://elvis.rowan.edu/segv/BTree_Exp_Study/.

Because of its simplicity, we skip the details of the pseudocode for these routines here, but is available in [27].

3.2 Test Suite

We have generated a large test suite consisting of binary trees of various types and sizes. We define the tree categories such that the trees contained in each category do not have significant variations, and have rather similar structures. We then performed our experimental study on this test suite.

Our test suite consists of five binary trees for each of the following types and sizes:

- *Randomly-generated binary trees* (see Figure 5):

  Each randomly-generated binary tree T_n with n nodes was generated by generating a sequence $T_0, T_1, \ldots, T_n$ of binary trees, where T_0 is the empty tree, and T_i was generated from T_{i-1} by inserting a new leaf v_i into it. The position where v_i is inserted in T_{i-1} is determined by traversing a path $p = u_0 u_1 \ldots u_m$ of T_{i-1}, where u_0 is the root of T_{i-1}, and u_m has at most one child. More precisely, we start at the root u_0, and in the general step, assuming that we have already traversed the sub-path $u_0 u_1 \ldots u_{i-1}$, we flip a coin. If "head" comes up, then if u_{i-1} has a left child c, then we set $u_i = c$, and move to u_i, otherwise we make v_i the left child of u_{i-1}, and stop. If "tail" comes up, then if u_{i-1} has a right child c, then we set $u_i = c$, and move to u_i, otherwise we make v_i the right child of u_{i-1}, and stop.

  We do not generate equal likely random trees because we want the trees to have relatively similar structural properties.

- *Unbalanced binary trees* (see Figure 6):

  We consider a binary tree T_n with n nodes as *unbalanced* if its height is greater than $n/\log n$.

  A binary tree T_n with n nodes is *unbalanced-to-the-left* (*unbalanced-to-the-right*) if it is unbalanced, and, in addition, the number of left (right) children in T_n is greater than its number of right (left) children.

  Each unbalanced-to-the-left (unbalanced-to-the-right) binary tree T_n with n nodes was generated in a similar way to the randomly-generated binary

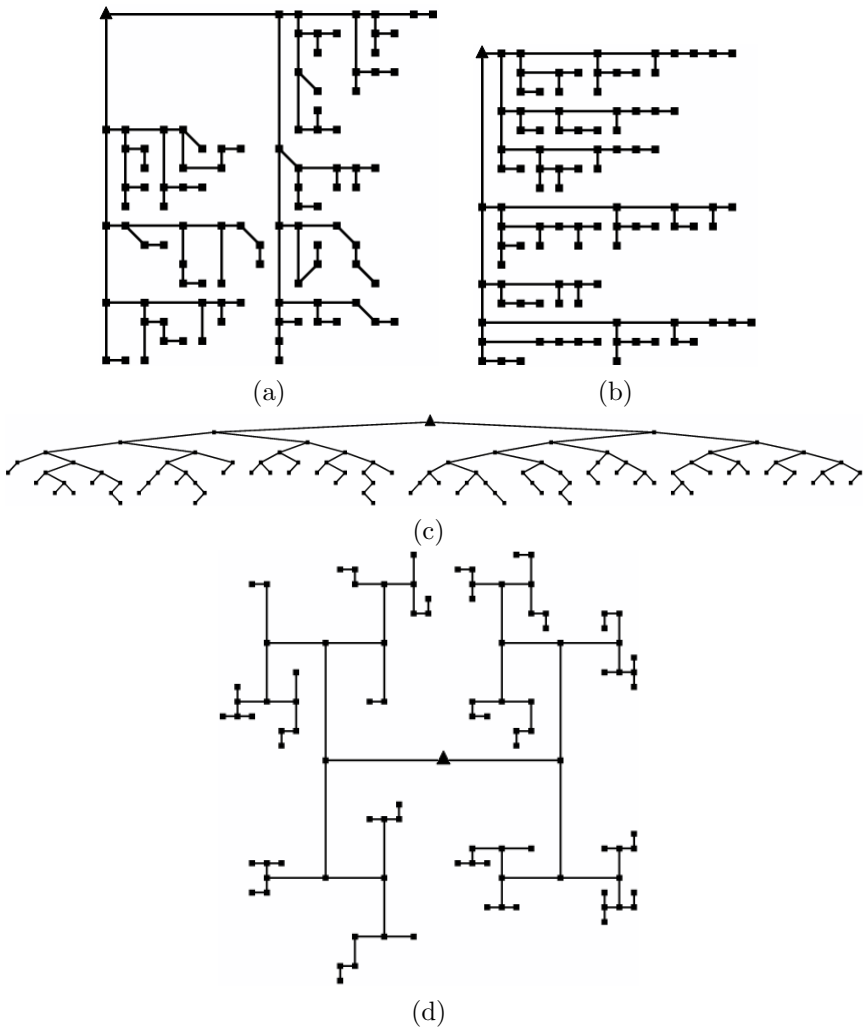


Figure 5: Drawings of a randomly-generated tree with 100 nodes, generated by the algorithms in our study: (a) Separation, (b) Path, (c) Level, and (d) Rings. The root is represented by the triangular node.

trees. The only difference occurs at the time of coin flipping: the probability of the coin coming "head" is set to be higher (lower) than the probability of the coin coming "tail".

We do not generate trees representative of all unbalanced trees because we want to have similarly structured unbalanced trees.

- *AVL trees* (see Figure 7):

  An *AVL tree* is a balanced binary tree where the height of the two subtrees of a node differs by at most one.

  Each AVL tree was generated by randomly inserting nodes in the tree using the same technique used for randomly-generated trees, and by using a generic method to maintain the tree's AVL property after each insertion.

The intuition behind generating random and unbalanced trees is that we want categories of trees with relatively similar structures, and different then the known categories of trees. It is highly unlikely that our random tree generation algorithm would generate an unbalanced tree since the likelihood of going left or right when inserting a new node is equal in the case of random trees and is heavily biased (toward right or left) in the case of unbalanced trees. We set the bias parameter to generate unbalanced trees which are near the unbalanced criteria (height greater than $n/logn$). For example, our dataset does not include trees which are structured as a left or right path.

Being unique trees for each type and size, we generated one tree for each of the following:

- *Complete binary trees* (see Figure 8):

  A binary tree T_n with n nodes is *complete* if every non-leaf node of T_n has exactly two children.

- *Fibonacci trees* (see Figure 9):

  A *Fibonacci tree* T_n is defined inductively as follows: T_0 is the empty tree, T_1 is the tree with one node, and T_n has as left subtree T_{n-1}, and as right subtree T_{n-2}. Note that a Fibonacci tree is the most unbalanced AVL tree allowed.

- *Molecular combinatory binary trees* (see Figure 10):

  These binary trees have a strong connection to "real-life" applications. The data was obtained from the study in [21] by Dr. Bruce MacLennan at the University of Tennessee. Within this research, Dr. MacLennan used combinatory logic [12], a mathematical formalism based on network substitution operations suggestive of supramolecular interactions. Binary trees derive from the networking conventions of combinatory logic and visualization of these binary trees could improve the investigator's ability in interpreting the substitution operations involved in combinatory logic. The idea is to use molecular processes to implement the combinatory logic

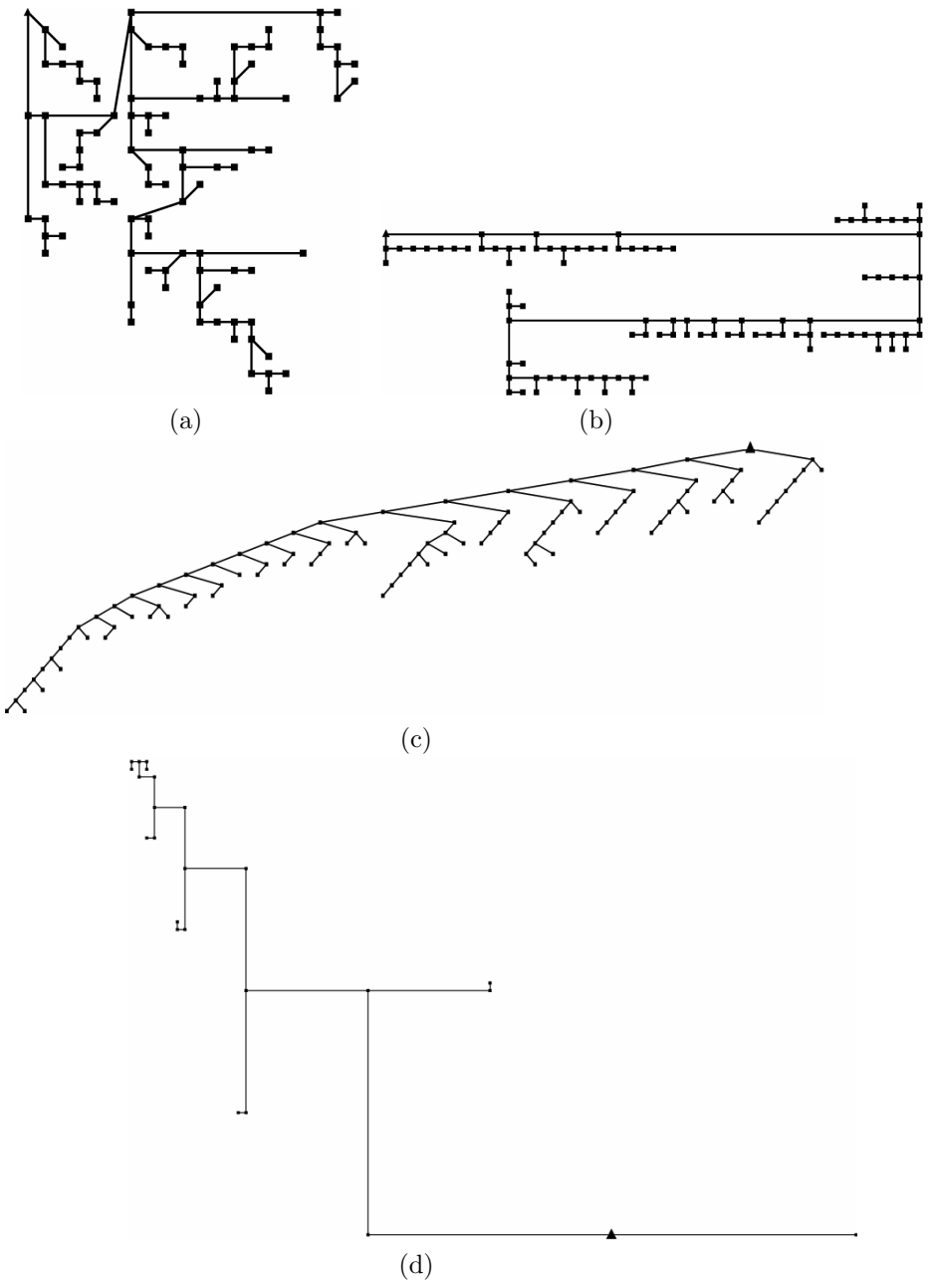


Figure 6: Drawings of an unbalanced-to-the-left binary tree with 100 nodes generated by the algorithms in our study: (a) Separation, (b) Path, (c) Level, and (d) Rings. For Rings, the tree is of size 25. The root is represented by the triangular node.

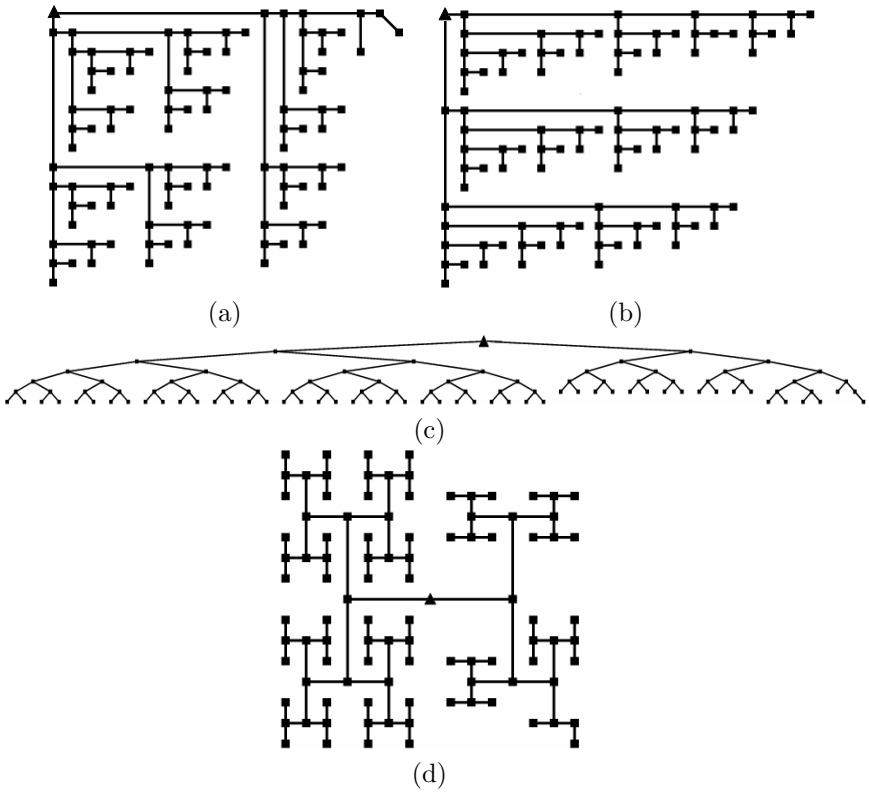


Figure 7: Drawings of an AVL tree with 100 nodes, generated by the algorithms in our study: (a) Separation, (b) Path, (c) Level, and (d) Rings. The root is represented by the triangular node.

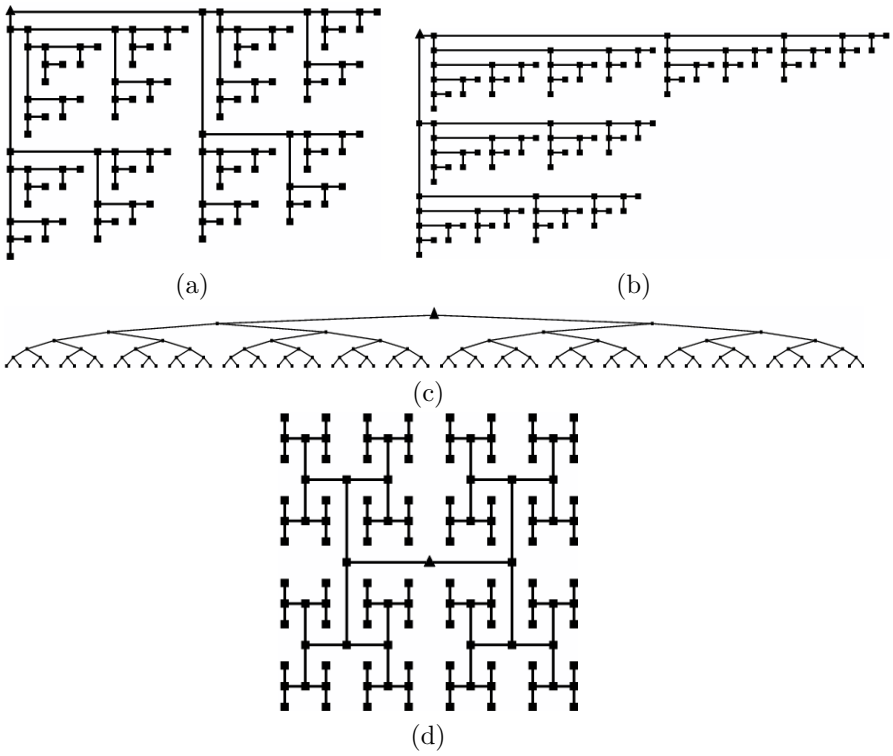


Figure 8: Drawings of the complete tree with 127 nodes, generated by the algorithms in our study: (a) Separation, (b) Path, (c) Level, and (d) Rings. The root is represented by the triangular node.

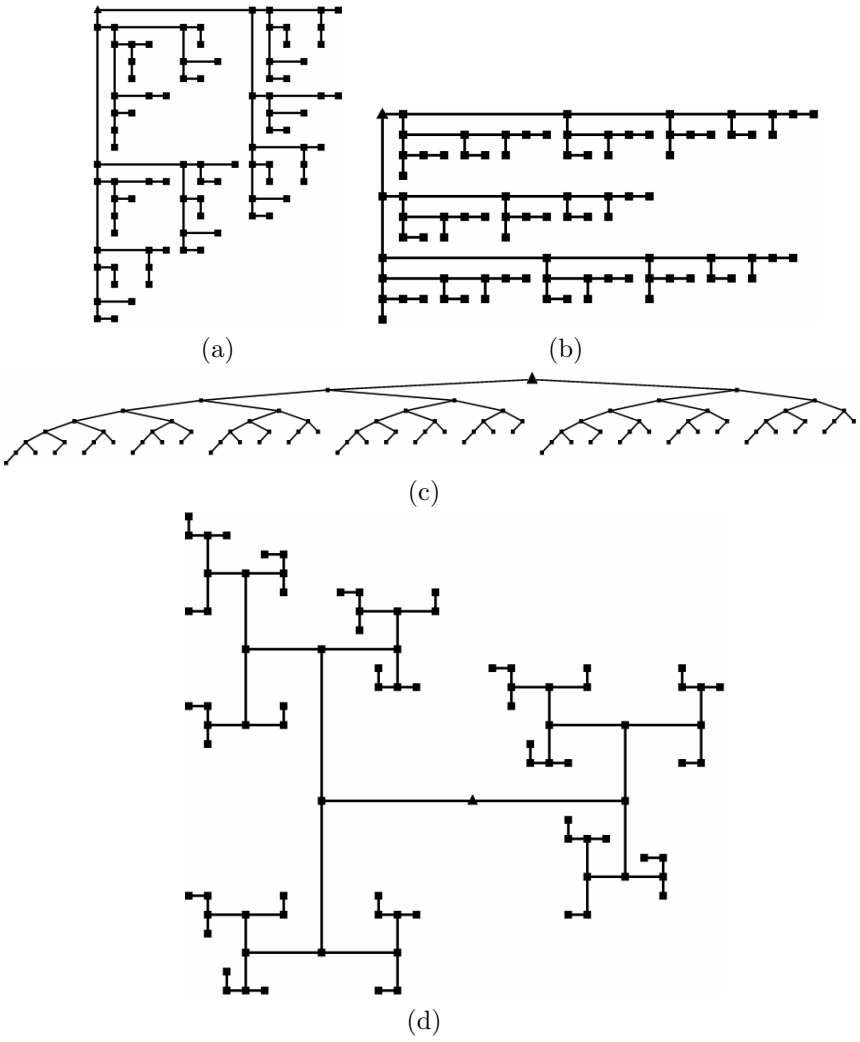


Figure 9: Drawings of the Fibonacci tree with 88 nodes, generated by the algorithms in our study: (a) Separation, (b) Path, (c) Level, and (d) Rings. The root is represented by the triangular node.

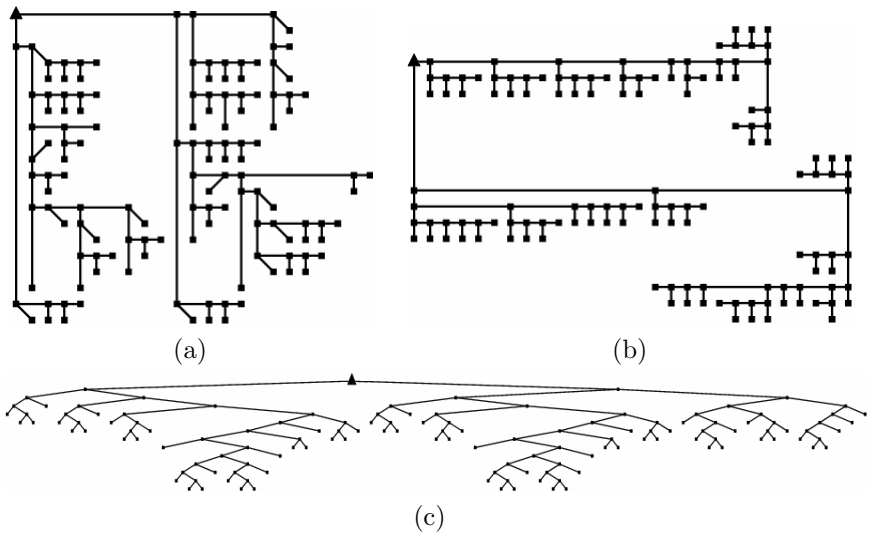


Figure 10: Drawings of a molecular combinatory binary tree with 133 nodes generated by the algorithms in our study: (a) Separation, (b) Path, and (c) Level. For Rings, the area was too large to include. The root is represented by the triangular node.

tree substitution operations, so that the molecular reorganization of the trees results in the desired structure or process.

We have generated random, unbalanced-to-the-left, unbalanced-to-the-right, and AVL binary trees with sizes between 100 and $50,000$, Fibonacci trees T_n for $n = 1, 2, ..., 45, 46$ (143 to $46,367$ nodes), complete binary trees of size $2^n - 1$ for $n = 7, 8, ..., 15, 16$ (127 to $65,535$ nodes), and molecular combinatory binary trees with between 133 and $50,005$ nodes. All these tree files are available for download at: http://elvis.rowan.edu/segv/BTree_Exp_Study/.

3.3 Quality Measures

The following eight well-known quality measures have been considered:

- **Area**: the number of grid points contained within the smallest rectangle with horizontal and vertical sides covering the drawing.
- **Aspect Ratio**: the ratio of the smaller and the longer sides of the smallest rectangle with horizontal and vertical sides covering the drawing.
- **Size**: the longest side of the smallest rectangle with horizontal and vertical sides covering the drawing.
- **Total Edge Length**: the sum of the lengths of the edges in the drawing.

- **Average Edge Length**: the average of the lengths of the edges in the drawing.
- **Maximum Edge Length**: the maximum among the lengths of the edges in the drawing.
- **Uniform Edge Length**: the variance of the edge lengths in the drawing.
- **Angular Resolution**: the minimum angle between any two edges in the drawing.

It is widely accepted [1, 2, 23, 24] that small values of the area, size, total edge length, average edge length, maximum edge length, and uniform edge length are related to the perceived aesthetic appeal and visual effectiveness of the drawing. In addition, an aspect ratio is considered *optimal* if it is equal to 1.

We have also considered two new quality measures, specially designed for trees:

- **Closest Leaf**: the smallest Euclidean distance between the root of the tree and a leaf in the drawing.
- **Farthest Leaf**: the largest Euclidean distance between the root of the tree and a leaf in the drawing.

The aesthetics **Closest Leaf** and **Farthest Leaf** help determine whether the algorithm places leaves close or far from the root. It is important to minimize the distance between the root and the leaves of the tree, especially in the case when the user needs to visually analyze the information contained in the levels close to the root and levels close to the leaves, without the information in between. Such a case appears in particular for algorithms where a change at the top level (root) of the tree generates modifications at the bottom levels (leaves) of the tree (for example, usual operations - find, insert, remove - on binary search trees, splay trees, or binary B^+ trees).

4 Experimental Analysis

Let T_n be a binary tree with n nodes that is provided as input to the algorithms being evaluated.

Two of the algorithms chosen in this study, namely *Separation* and *Path*, allow user-controlled aspect ratio, i.e. the user may change the aspect ratio by providing some parameters as input to the algorithms. The other two algorithms, (*Level* and *Rings*), generate unique drawings for each value of n. In order to find the parameters for which *Separation* and *Path* perform the best on each of the aesthetics considered in our study, we used the studies in [16] and [27], respectively. These parameters were placed in a lookup table and used to find the best value for each combination of aesthetic, tree-type, and number of nodes.

4.1 Comparison Analysis

In order to compare the algorithms, we varied n between 100 and $50,000$ for randomly-generated, unbalanced-to-the-left, unbalanced-to-the-right, and AVL binary trees, T_n for $n = 1, 2, ..., 45, 46$ (143 to $46,367$ nodes) for Fibonacci trees, $2^n - 1$ for $n = 7, 8, ..., 15, 16$ (127 to $65,535$ nodes) for complete binary trees, and between 133 and $50,005$ for molecular combinatory trees. We compared the performance of the algorithms for each tree-type separately.

The performance of *Separation* was not evaluated for aspect ratio because the desired aspect ratio is user configurable.

The performance of *Rings* was not considered in our comparisons for unbalanced and molecular combinatory binary trees because in these cases the area *Rings* produces grows exponentially and it quickly becomes prohibitive to use. For example, for an unbalanced tree with $1,000$ nodes, *Rings* produces a drawing with area of $7.65 \cdot 10^7$. Also, for molecular combinatory binary trees with 469 nodes, *Rings* produces a drawing with area of $2.15 \cdot 10^9$.

We do not create separate charts for the quality measure **Average Edge Length** because, in a binary tree with n nodes, the average edge length is always equal to the total edge length divided by $n - 1$. Therefore, we use the charts for the quality measure **Total Edge Length** to analyze the behavior of the algorithms for this aesthetic.

Figures 15-23 display the performance of *Level*, *Path*, *Rings*, and *Separation*. The x-axis of each chart shows the number of nodes.

The analysis of the performance of the four algorithms for each quality measure, and for each tree-type is summarized below:

- **Area**: (See Figure 15)
  - *Complete binary trees*: (See Figure 15(a)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. While the difference in the areas produced by *Rings* and *Separation* grows slowly, the difference in the areas produced by *Separation* and *Path* grows much faster. The same behavior is exhibited in *Level* and *Path*. For the last value of n considered ($n = 65,535$), *Level* produces a drawing having an area almost four times more than the drawing produced by *Path*.
  - *AVL trees*: (See Figure 15(b)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. *Rings* and *Separation* exhibit similar behavior, with *Rings* being slightly better. The differences in the areas produced grow slowly.
  - *Randomly-generated binary trees*: (See Figure 15(c)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. The performances of all the algorithms are worse than their respective performances on complete trees. In comparison to its behavior on complete trees, where it was the best, *Rings* exhibits the most dramatic change: its behavior is now the worst of all four algorithms. The area produced by *Level* grows rapidly in comparison to the area produced by *Path*,

being already three times more for the last value of n considered ($n = 50,000$).

- *Fibonacci trees*: (See Figure 15(d)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. *Rings* quickly becomes prohibitive, with area ten times more than the area of *Separation*, for $10,000$ nodes. The difference between the areas produced by *Separation* and *Path* grows slowly, while the difference between the areas produced by *Path* and *Level* grows much faster.
- *Unbalanced-to-the-left binary trees*: (See Figure 15(e)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. *Level* quickly becomes prohibitive, and it has been only partially plotted. *Path* slightly outperforms *Separation*.
- *Unbalanced-to-the-right binary trees*: (See Figure 15(f)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. *Path* produces excellent results for this type of tree. This is the worst case for *Separation*. *Level* rapidly becomes prohibitive, and it has been only partially plotted.
- *Molecular combinatory binary trees*: (See Figure 15(g)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. Even though *Path* is the best performing algorithm on both unbalanced and molecular combinatory binary trees, its behavior on molecular combinatory binary trees is much better: for $n = 50,000$, the area of molecular combinatory binary trees is almost half then in the case of unbalanced binary trees. *Level* rapidly becomes prohibitive to use, producing an area over $1,000,000$ for n of about $6,000$. This is the best case for *Path*.

- **Aspect Ratio**: (See Figure 16)
  - *Complete binary trees*: (See Figure 16(a)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. Quite interestingly, the behavior of *Path* and *Rings* is very similar. Neither algorithm always produces drawings with aspect ratios close to optimal. For example, if $n = 2^{14} - 1$, the best aspect ratio *Path* produces is around 0.5. *Rings* produces optimal aspect ratios when $n = 2^i - 1$, with i an odd number, and aspect ratios close to 0.5, with i an even number. The aspect ratios of the drawings produced by *Level* are very low (the highest value is close to 0.06), decreasing rapidly as n increases.
  - *AVL trees*: (See Figure 16(b)) Order of performance: *Separation*, *Rings*, *Path*, *Level*. *Rings* exhibits a very interesting pattern: its aspect ratios are either 0.5 or optimal. The performances of *Path* and *Level* decrease dramatically, with *Level* quickly producing very small aspect ratios, and *Path* producing aspect ratios less than 0.01 for 50,000 nodes.
  - *Randomly-generated binary trees*: (See Figure 16(c)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. *Level* produces drawings

with better aspect ratios for trees with smaller number of nodes (the highest value is close to 0.1). Still, its behavior is unsatisfactory, as the value of aspect ratio decreases rapidly as n increases. *Path* and *Rings* have uneven behaviors. Most of their aspect ratios are over 0.8, and none are under 0.5.

- *Fibonacci trees*: (See Figure 16(d)) Order of performance: *Separation*, *Rings*, *Path*, *Level*. Interestingly, *Rings* exhibits exactly the same behavior as in the case of complete binary trees: optimal aspect ratios when $n = 2^i - 1$, with i an odd number, and aspect ratios close to 0.5, with i an even number. The behavior of *Level* is only significant for trees with small number of nodes.
- *Unbalanced-to-the-left binary trees*: (See Figure 16(e)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. Quite surprisingly, *Level* produces drawings with better aspect ratios than before and its performance decreases very slowly as n increases. Quite interestingly, the performance of *Path* is almost identical with the one for randomly-generated binary trees, with most of the aspect ratios over or close to 0.8, and no aspect ratio under 0.6.
- *Unbalanced-to-the-right binary trees*: (See Figure 16(f)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. *Level* exhibits similar behavior as in the case of unbalanced-to-the left binary trees. Interestingly, while for small values of n the performance of *Path* is close to optimal, it decreases rapidly as n increases. For example, for 50,000 nodes, *Level* produces better aspect ratios than *Path*.
- *Molecular combinatory binary trees*: (See Figure 16(g)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. *Path* produces close to optimal values until n about $6,000$. After this point, the values plummet, decreasing to 0.1 for $n = 50,005$. Very interestingly, *Level* always produces values close to 0.1. In our analysis, it was discovered that *Level* always produces drawings of width equal to n. Hence, for molecular combinatory binary trees, the height is almost always one-tenth of the width.

- **Size**: (See Figure 17)
  - *Complete binary trees*: (See Figure 17(a)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. *Level* grows at an exponential rate for all categories of binary trees and will not be mentioned. *Path*, *Rings*, and *Separation* grow fast and then reach an individual point where rate of growth is comparable.
  - *AVL trees*: (See Figure 17(b)) Order of performance: *Separation*, *Rings*, *Path*, *Level*. Strangely, the performances of *Rings* and *Separation* oscillate as best performance. *Separation* grows at a constant rate, while *Rings* values are scattered about the performance of *Separation*. The performance of *Path* is at its worst, becoming two and a half times larger than *Separation* at $n = 50,000$.

- *Randomly-generated binary trees*: (See Figure 17(c)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. *Rings* grows much faster than *Path* and *Separation*, in particular for high values of n. *Separation* grows at a constant rate and *Path* grows almost two times faster than *Separation*.
- *Fibonacci trees*: (See Figure 17(d)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. Ironically, *Path* starts as the best, but as n increases, so too does its rate of growth. Ultimately, the faster rate of *Path* enables *Separation* to outperform all in the end. *Rings* performs at its worst compared to all other categories of binary trees.
- *Unbalanced-to-the-left binary trees*: (See Figure 17(e)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. The performance of *Separation* and *Path* are very similar, and as n gets larger *Path* becomes slightly worse. *Level* again exhibits unsatisfactory performance.
- *Unbalanced-to-the-right binary trees*: (See Figure 17(f)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. The behavior of *Separation*, *Path*, and *Level* on this tree type are very similar to that on unbalanced-to-the-left binary trees.
- *Molecular combinatory binary trees*: (See Figure 17(g)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. The performance of *Path* starts out as the best algorithm, but is overtaken by *Separation* at n about $12,000$. The performance of *Path* is near linear, becoming two times larger than the results of *Separation* at approximately $n = 50,005$.

- **Total Edge Length** and **Average Edge Length**: (See Figure 18)
  - *Complete binary trees*: (See Figure 18(a)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The similarity between the plots for total edge length and area fits the intuitive notion that low total edge length and area generally go together. All of the observations made for area apply in this case, except when $n = 50,000$, *Path* performs better than *Separation*, and is very close to the performance of *Rings*.
  - *AVL trees*: (See Figure 18(b)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The results for *Rings* and *Separation* are very similar. Level performs poorly, producing results nearly four times greater than *Rings*. The intuition that low total edge length and area generally go together is confirmed in this case.
  - *Randomly-generated binary trees*: (See Figure 18(c)) Order of performance: *Separation*, *Rings*, *Path*, *Level*. The interesting finding here is that *Rings* and *Path* exhibit almost identical behavior. *Separation* performs a little better than these two, and the gap between the performance of *Level* and the performance of the other algorithms grows fast. It is also interesting to see that, with lower total edge length

than *Path* and *Level*, *Rings* constructs larger area then the two, thus contradicting the intuition linking total edge length and area.

- *Fibonacci trees*: (See Figure 18(d)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. The intuition that low total edge length and area generally go together is confirmed for the two best-performing algorithms (*Separation* and *Path*), but contradicted for the two worst performing algorithms (*Rings* and *Level*). The performances of *Path* and *Rings* are similar.
- *Unbalanced-to-the-left binary trees*: (See Figure 18(e)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. In this situation, the intuition that low total edge length and area go together is confirmed. For example, *Path* exhibits better behavior than *Separation* for total edge length, which complements the results from area. Also, *Level* and *Separation* exhibit almost identical behavior for total edge length, while *Level* produces an area exponentially higher than *Separation*.
- *Unbalanced-to-the-right binary trees*: (See Figure 18(f)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. The intuition that the performance on area and total edge length produce similar results is not confirmed; *Path* performed the best for area, here *Separation* is best. Within their order of performance, the behavior of the algorithms individually increase at a constant rate with respect to the number of nodes.
- *Molecular combinatory binary trees*: (See Figure 18(g)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. Again, the intuition in coupling total edge length and area is confirmed. Within their order of performance, the behaviors of *Path* and *Separation* individually increase at a constant rate with respect to the number of nodes, and the performance of *Level* grows exponentially.

- **Maximum Edge Length**: (See Figure 19)
  - *Complete binary trees*: (See Figure 19(a)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The performance of *Level* for maximum edge length grows very quickly in comparison to the other algorithms. The difference between the maximum edge lengths produced by *Rings* and those produced by *Separation* grows slowly, and the difference between *Separation* and *Path* grows fast for small trees but narrows rapidly for large trees. For the last value of n ($n = 65,535$), maximum edge length for *Path* is three times that of *Rings* and maximum edge length for *Separation* is twice that of *Rings*.
  - *AVL trees*: (See Figure 19(b)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The behavior of *Path* behaves in a non-linear fashion. Also, *Level* exhibits behavior much worse than the others. *Separation* and *Rings* produce very good results and the difference between their performances seem to grow very slowly.

- *Randomly-generated binary trees*: (See Figure 19(c)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. Again, *Level* produces unsatisfactory results, and quickly becomes prohibitive to use. The performance of *Rings* is better than the performance of *Path* up to n at about $30,000$, after which *Path* outperforms *Rings*. The behavior of *Separation* grows slowly.
- *Fibonacci trees*: (See Figure 19(d)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. *Separation* and *Path* have almost identical behavior and *Rings* grows much faster. Also, again, *Level* exhibits behavior much worse than the others.
- *Unbalanced-to-the-left binary trees*: (See Figure 19(e)) Order of performance: *Level*, *Path*, *Separation*, *Rings*. Quite surprisingly, *Level* produces almost constant maximum edge length. Moreover, it is very low. *Path* also produces steady results. *Separation* exhibits a much faster growing behavior.
- *Unbalanced-to-the-right binary trees*: (See Figure 19(f)) Order of performance: *Separation*, *Level*, *Path*, *Rings*. Very surprisingly, *Separation*, *Path*, and *Level* produce similar, low, almost constant maximum edge lengths.
- *Molecular combinatory binary trees*: (See Figure 19(g)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. To this point, the performances on molecular combinatory binary trees and unbalanced trees have been similar. Very interestingly, *Level* produces the worst maximum edge length, but for unbalanced binary trees *Level* produces the best results. Initially, *Path* produces the best results, until about $n = 6,000$, at which point *Separation* overtakes *Path* while its own rate of growth decreases.

- **Uniform Edge Length**: (See Figure 20)
  - *Complete binary trees*: (See Figure 20(a)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The performances agree with the intuition that the performance for total edge length and uniform edge length are similar. *Rings* and *Separation* produce very low, almost constant values. *Path* exhibits a very interesting non-linear behavior, with significantly worse values for the cases when the tree has $2^i - 1$ nodes, with i even.
  - *AVL trees*: (See Figure 20(b)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The performances of *Rings* and *Separation* are very good: they almost mimic their performances on complete binary trees. The results for *Path* do not have a consistent rate of change.
  - *Randomly-generated binary trees*: (See Figure 20(c)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The performances of *Rings* and *Separation* are very similar to those on complete binary trees.

With respect to the intuition low total edge length and uniform edge length go together, *Path* exhibits a contradicting behavior.

- *Fibonacci trees*: (See Figure 20(d)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. *Separation* outperforms all other algorithms, while maintaining a nearly constant value. *Path* slightly outperforms *Rings*.
- *Unbalanced-to-the-left binary trees*: (See Figure 20(e)) Order of performance: *Path*, *Level*, *Separation*, *Rings*. *Path* has gone from one of the poorer performing algorithms for uniform edge length to performing the best. *Level* and *Separation* almost perform exactly the same. *Rings* again exhibits unsatisfactory performance.
- *Unbalanced-to-the-right binary trees*: (See Figure 20(f)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. Surprisingly, *Path* and *Level* seemed to have the same performance as in unbalanced-to-the-left binary trees, where *Separation* became noticeably better.
- *Molecular combinatory binary trees*: (See Figure 20(g)) Order of Performance: *Separation*, *Path*, *Level*, *Rings*. *Level* performs very poorly. *Path* quickly becomes prohibitive. Interestingly, the intuition that total edge length and uniform edge length go together is not confirmed in this case: while the total edge length for *Separation* is slightly larger (almost identical) than the one for *Path*, the uniform edge length for *Path* grows much faster than the one for *Separation*.

- **Angular Resolution**: (See Figure 21)
  - *Complete binary trees*: (See Figure 21(a)) Order of performance: *Rings*, *Path*, *Separation*, *Level*. *Path* and *Rings* have not been considered in the analysis of angular resolution, because *Path* and *Rings* only draw binary trees using 90° and 180° angles. *Separation* provides good angular resolution, because of constant angles of 90°. *Level* begins very poorly and as n increases *Level* angular resolution becomes more severe.
  - *AVL trees*: (See Figure 21(b)) Order of performance: *Rings*, *Path*, *Separation*, *Level*. Similarly, the results resemble closely to complete binary trees. Once again, *Separation* reaches the optimal angular resolution value.
  - *Randomly-generated binary trees*: (See Figure 21(c)) Order of performance: *Rings*, *Path*, *Separation*, *Level*. *Separation* produces no angles less than 45°. Again, *Level* is increasingly prohibitive.
  - *Fibonacci trees*: (See Figure 21(d)) Order of performance: *Rings*, *Path*, *Separation*, *Level*. *Separation* has all values of 90°. The consistency of *Level* being the worst is confirmed.

- *Unbalanced-to-the-left binary trees*: (See Figure 21(e)) Order of performance: *Rings*, *Path*, *Separation*, *Level*. *Separation* has good angular resolution for lower values of n, but then becomes worse for larger tree sizes. *Level* again exhibits unsatisfactory performance.
- *Unbalanced-to-the-right binary trees*: (See Figure 21(f)) Order of performance: *Rings*, *Path*, *Separation*, *Level*. Results mimic that of its opposite tree-type unbalanced-to-the-left.
- *Molecular combinatory binary trees*: (See Figure 21(g)) Order of Performance: *Rings*, *Path*, *Separation*, *Level*. Similar to unbalanced binary trees, *Level* produces poor angular resolution. Also, for *Separation*, the angular resolution is good for all sizes of trees, with constant angular resolution of $45°$.

- **Closest Leaf**: (See Figure 22)
  - *Complete binary trees*: (See Figure 22(a)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. In general, all algorithms place at least one leaf close to the root. For example, for $n = 65,535$, the longest distance between the root and the closest leaf is 15.03 in the case of *Level*. Interestingly, *Rings* always places a leaf very close to the root at a constant distance of 1.41. *Separation*, *Path*, and *Level* place the closest leaf increasingly farther away from the root, growing at a very slow rate, with the distance between the root and the closest leaf for *Level* growing faster than the one for *Separation* and *Path*. *Separation* and *Path* exhibit almost identical behavior.
  - *AVL trees*: (See Figure 22(b)) Order of performance: *Rings*, *Path*, *Separation*, *Level*. The distance to the closest leaf for *Level* and *Path* slowly increases as n increases. The performance of *Separation* is very similar to *Path*. Also, *Level* and *Path* exhibit similar behaviors as in the case of complete and random binary trees. Again, *Rings* always places a leaf at distance 1.41 from the root all of the time.
  - *Randomly-generated binary trees*: (See Figure 22(c)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. The performances of *Path* and *Level* have similar rates of growth, with *Path* producing almost two times better results. *Level* and *Separation* provide almost identical results. In contrast, *Rings* performs poorly, becoming seven times larger than *Path* at $n = 30,000$.
  - *Fibonacci trees*: (See Figure 22(d)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. *Rings* places leaf nodes far from the root compared to the other algorithms, and the performance grows very quickly. Again, *Separation*, *Level*, and *Path* produce almost constant results, with *Separation* and *Level* exhibiting almost identical behavior.
  - *Unbalanced-to-the-left binary trees*: (See Figure 22(e)) The order of performance alternates for different tree sizes, but in general is: *Path*,

Separation, *Level*, *Rings*. All algorithms place the closest leaf at increasing distance from the root, with their performance non-linear. The performance of *Path* and *Separation* are almost identical, and that of *Level* significantly worse.

- *Unbalanced-to-the-right binary trees*: (See Figure 22(f)) Order of performance: *Path*, *Level*, *Separation*, *Rings*. The relationship between the performances of *Level*, *Path*, and *Separation* maintain the same behavior (good and bad) as n increases. For larger values of n, all algorithms experience a decrease in rate change.
- *Molecular combinatory binary trees*: (See Figure 22(g)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. The performance of *Level* becomes worse very fast. Very interestingly, *Path* always places a leaf at a distance of 2.24 from the root. Similarly, *Separation* always places a leaf at a distance of 3.61 from the root.

- **Farthest Leaf**: (See Figure 23)
  - *Complete binary trees*: (See Figure 23(a)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. While the performances of *Path*, *Rings*, and *Separation* are very good, with a very slow growth rate, the performance of *Level* is unsatisfactory.
  - *AVL trees*: (See Figure 23(b)) Order of performance: *Rings*, *Separation*, *Path*, *Level*. The performances of the algorithms on this measure are almost identical to their respective performances on complete trees, except *Path* is slightly worse.
  - *Randomly-generated binary trees*: (See Figure 23(c)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. Surprisingly, both *Separation* and *Rings* perform just slightly worse on randomly-generated binary trees compared to complete trees. Performances of *Path* and *Rings* are almost identical. Again, the distance to the farthest leaf grows much faster for *Level* than for *Path*, *Separation* and *Rings*.
  - *Fibonacci trees*: (See Figure 23(d)) Order of performance: *Separation*, *Path*, *Rings*, *Level*. For *Separation* and *Path* the same pattern remains. On the other hand, *Rings* still remains better than *Level*, but the rate of growth in *Rings* has increased. *Level* exhibits the same unsatisfactory behavior.
  - *Unbalanced-to-the-left binary trees*: (See Figure 23(e)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. Interestingly, the performances of the algorithms closely resemble that of the previous two tree-types.
  - *Unbalanced-to-the-right binary trees*: (See Figure 23(f)) Order of performance: *Path*, *Separation*, *Level*, *Rings*. A pattern has formed in the performances of the algorithms. The results in unbalanced-to-the-left trees closely mimic that of unbalanced-to-the-right trees.

– *Molecular combinatory binary trees*: (See Figure 23(g)) Order of performance: *Separation*, *Path*, *Level*, *Rings*. As the case in all categories of binary trees, *Level* rapidly becomes prohibitive. *Separation* produces satisfactory results and the behavior of *Path* is approximately three times worse than *Separation*. Very interestingly, the intuition that short maximum edge length and close farthest leaf go together is verified in the case of molecular combinatory binary trees, but not verified in the case of unbalanced trees.

4.2 Conclusions

After evaluating the performances of *Level*, *Path*, *Rings*, and *Separation*, we have reached the following conclusions:

- Quality measure **Area**:
  - *Separation* produces best results for randomly-generated and Fibonacci trees. The drawings that *Separation* constructs in these cases, also achieve optimal aspect ratios.
  - *Rings* produces excellent results for complete binary trees and AVL trees. Its performance degrades dramatically for other types to trees. For randomly-generated binary trees, its behavior is the worst of the four algorithms. For unbalanced binary trees, the area produced was so large that it could not be used for comparison.
  - *Path* produces excellent results for unbalanced and molecular combinatory binary trees. These results come at the expense of a very low aspect ratio. *Path* may produce drawings of these types of trees, with aspect ratio close to optimal, but in this case, its performance is comparable to that of *Separation*.
  - *Level* always produces results worse than *Path* and *Separation*.
- Quality measure **Aspect Ratio**:
  - Since the aspect ratio of the drawings produced by *Separation* is user-controlled, this algorithm always achieves an aspect ratio close to optimal. This may come at the expense of a slightly larger area.
  - Generally, *Path* produces aspect ratios close to optimal. This comes at the expense of a larger area. For unbalanced-to-the-right binary trees, AVL trees, Fibonacci trees, and molecular combinatory trees, the aspect ratios produced degrade quickly and become unsatisfactory.
  - For the cases in which can be plotted (complete, randomly-generated, AVL, and Fibonacci trees), *Rings* always produces aspect ratios over 0.5, and many times it achieves optimality.
  - *Level* always produces unsatisfactory results.

- Quality measure **Size**:
  - *Separation* performs well on all categories of trees and produces the best results on randomly-generated, unbalanced-to-the-left, AVL, Fibonacci, and molecular combinatory binary trees.
  - *Path* performs best on unbalanced-to-the-right, well on Fibonacci trees, and worst on randomly-generated and molecular combinatory binary trees.
  - *Rings* performs best on complete and unbalanced-to-the-left binary trees, and worst on randomly-generated and Fibonacci trees.
  - *Level* produces unsatisfactory results for all categories of trees.
- Quality measures **Total Edge Length** and **Average Edge Length**:
  - *Rings* produces best results for complete and AVL binary trees.
  - *Separation* produces best results for randomly-generated, Fibonacci, and unbalanced-to-the-right binary trees.
  - *Path* produces best results for unbalanced-to-the-left and molecular combinatory binary trees.
  - *Level* produces worst results in all categories except for unbalanced-to-the-right binary trees.
- Quality measure **Maximum Edge Length**:
  - *Separation* performs best on randomly-generated, Fibonacci, and molecular combinatory binary trees, and unbalanced-to-the-right binary trees.
  - *Path* performs worst on randomly-generated binary trees, best on unbalanced-to-the-left binary trees, and well on the other categories of trees.
  - *Level* produces very good results for unbalanced binary trees, and very bad results for all other categories of trees.
  - *Rings* performs best on complete and AVL binary trees, and well on the other types of trees (with the exception of unbalanced and molecular combinatory trees where we could not have a plot).
- Quality measures **Uniform Edge Length**:
  - *Rings* produces very good results for all categories of trees.
  - *Separation* produces good results for all categories of trees, with the best results on Fibonacci, unbalanced-to-the-right, and molecular combinatory binary trees.
  - *Path* performs best on unbalanced binary trees, well on random binary trees, and poorly on Complete and AVL binary trees.

  - *Level* performs best on unbalanced binary trees, and very poorly on all other categories of trees.

- Quality measures **Angular Resolution**:

  - *Separation* performs the best on complete, AVL, Fibonacci, and molecular combinatory binary trees, well on randomly-generated binary trees, and unsatisfactory on unbalanced binary trees.
  - *Rings* produces optimal results for all trees except unbalanced binary trees.
  - *Path* produces orthogonal drawings. Therefore, in all cases, the drawings produced by *Path* have at least a 90° separation between the edges connecting a parent-node to its children.
  - *Level* produces bad results for all categories of trees.

- Quality measure **Closest Leaf**:

  - *Path* produces excellent results on all types of trees, with its best performance on Fibonacci trees, and its worse performance on complete binary trees.
  - *Rings* produces excellent results on complete and AVL trees and worse, unsatisfactory results on Fibonacci and unbalanced binary trees.
  - *Separation* produces best results for molecular combinatory binary trees and very good results for all other types of trees, except for unbalanced-to-the-right binary trees, on which it produces its worse results.
  - *Level* performs best on unbalanced and Fibonacci trees and worst on AVL, complete, and randomly-generated binary trees.

- Quality measure **Farthest Leaf**:

  - *Separation* produces very good results on all categories of trees. Its best performance is on complete, AVL, and molecular combinatory binary trees, and its worse performance is on unbalanced-to-the-right binary trees.
  - *Path* produces good results for tree-types, with its best being for unbalanced binary trees.
  - *Rings* performs best on complete and AVL trees, and worst on unbalanced and Fibonacci trees.
  - *Level* performs worse than the other algorithms for all categories of trees. The performance of *Level* is not satisfactorily on any category of trees.

Overall, if the aspect ratio of the drawing is important, *Separation* is the algorithm which achieves best results in the majority of the aesthetics, with the worst results coming for unbalanced-to-the-right binary trees. This is because *Separation* always places the root in the top-left corner, thus making more difficult to place the rest of the nodes. If the aspect ratio is not important, then *Rings* should be the choice for AVL and complete binary trees, *Path* should be the choice for unbalanced binary trees, and *Separation* should be the choice for Fibonacci and molecular combinatory trees.

5 Adaptive Tree Drawing System

The lack of a single algorithm performing the best for all quality measures has lead us to design an adaptive system comprised of the four previously mentioned algorithms. Since it is necessary to know the type of a binary tree before being able to select an algorithm to draw it, our adaptive tree drawing system first analyzes the tree to classify it as a specific type and then selects an algorithm to draw it with respect to user-specified quality measures. The algorithm that is selected to draw a given tree is based upon the experimental comparison (See Section 4.1), which orders the performance of the algorithms for each quality measure. When the selected algorithm is *Path* or *Separation*, the system also uses a lookup table to find the best input parameters for the quality measures.

5.1 System Components

Our system is composed of approximately $5,000$ lines of code, the majority of which is C++ code. The user interface is implemented in Java. All of the underlying algorithms, including the tree-type determination algorithm, are implemented in C++. The Java and C++ portions of the system interact with each other through a series of system calls.

The four algorithms for producing planar straight-line grid drawings of binary trees used by our system are described in Section 2. Our binary trees database consists of the same trees included in Section 3.2, but any other tree is accepted as input. The quality measures defined in Section 3.3 can be selected within our system.

5.2 User Interface

Our system provides a simple interface for visualizing binary trees with respect to user-specified quality measures (See Figure 11).

The first step is to select a tree to visualize. The tree is stored in a file following the format described in Section 3.1. After selecting a tree, the user can then choose one, two, or three quality measures that the specified tree should be drawn with respect to.

The user then submits their selection and a drawing of the specified tree is displayed (See Figure 12).

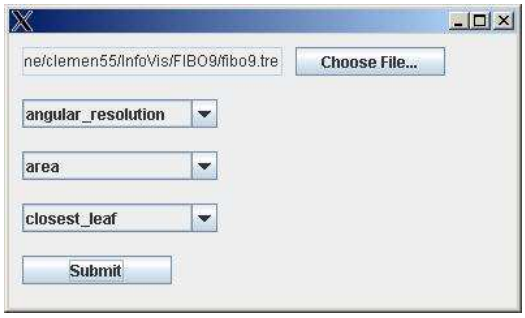


Figure 11: The selection screen with quality measures **angular resolution**, **area**, and **closest leaf** selected.

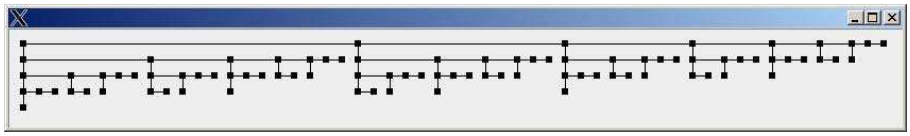


Figure 12: The drawing of the specified tree with respect to the specified quality measures.

5.3 Type Determination Algorithm

We denote by $n(v)$ the number of nodes in the tree rooted at node v, $h(v)$ the height of the tree rooted at node v, $lt(v)$ the subtree rooted at the left child of node v, $rt(v)$ the subtree rooted at the right child of node v, $l(v)$ the number of left children in the tree rooted at node v, and $r(v)$ the number of right children in the tree rooted at node v.

We give below pseudocode for the type determination algorithm. Our tree-type determination algorithm is correct since the type tests are based upon the definition of each of the tree types.

Algorithm DetermineType

Input: An input file containing a binary tree T with root node v stored in the format described in Section 3.1.
Output: One of the following strings describing the type of the binary tree: **"complete"**, **"unbalanced left"**, **"unbalanced right"**, **"Fibonacci"**, **"AVL"**, or **"random"**.

if $h(v) = log_2(n(v) + 1)$ **then** return **"complete"**.
else if T is AVL-balanced **then** {
  if $lt(v)$ is T_{n-1} and $rt(v)$ is T_{n-2} **then** return **"Fibonacci"**
  else return **"AVL"**

```
    }
    else if n(v)/log2(n(v)) < h(v) then {
        if l(v) < r(v) then return "unbalanced right"
        else return "unbalanced left"
    }
    else return "random";
end Algorithm.
```

Our system was given each tree in our test suite for identification. The tree type was correctly identified in every case. If the tree does not fall into any of the known categories, then the tree is considered random. A more relaxed criteria for classifying unknown types of trees, such as classifying trees based on structural similarities (instead of matchings) with known types, has the potential of producing better results.

Comparing the height of the tree with the value $log_2(n(v) + 1)$ takes $O(1)$ time. Therefore, determining if a given binary tree is complete takes $O(1)$ time. Determining if a tree is AVL-balanced by checking if the height of the left and right subtrees of a given binary tree differ by no more than one is $O(n(v))$ since every node is visited. The time complexities for the tests for determining if a given binary tree is a Fibonacci tree is $O(n(v))$. If a binary tree is determined to be AVL-balanced, but is not Fibonacci, it is considered an AVL tree. The time complexities for the tests for determining if a given binary tree is an unbalanced tree are as follows: Calculating $n(v)/log_2(n(v)) < h(v)$ is $O(1)$. Comparing $l(v)$ and $r(v)$ is $O(1)$. Overall, determining if a given binary tree is unbalanced-to-the-left or unbalanced-to-the-right takes $O(1)$ time. If a tree is random, then the complexity to come to this decision is $O(1)$. The time complexity of the entire tree-type determination algorithm is the sum of the complexities of the previous tests, $O(n(v))$.

5.4 Type Determination Results

Once the type of T is determined, one of the four algorithms is chosen to draw T. The type of the tree and the user-specified quality measures are used to determine which algorithm to use when creating the drawing of T. When multiple quality measures are specified, we keep track of a weight (initially 0) for each algorithm and add to the weight depending upon the order of performance for the quality measure. Our general weighting function can be defined as follows: given the type of binary tree and a set of n specified quality measures, the weight given to a specific algorithm is $\sum_{i=0}^{n} \frac{MAX\_WEIGHT}{2^{Rank(Algorithm, tree\_type, quality\_measures[i])}}$, where $Rank(Algorithm, tree\_type, quality\_measures[i])$ returns the position of the specified algorithm in the order of performance for the specified tree type and quality measure. In our case, our system uses four algorithms, so the possible ranks for a given algorithm are 0, 1, 2, and 3, with 0 being the best rank. MAX_WEIGHT is also determined by the number of algorithms in our system. In our case, MAX_WEIGHT is 8, which ensures a unique weight for every al-

gorithm for every specified quality measure. A more specific weight function may be chosen by the user if it pertains to their application. For example, *Rings* produces the best area for complete trees, followed by *Separation*, *Path*, and *Level*, in that order. If one of the specified quality measures was **Area**, then the weight associated with *Rings* would be increased by 8, the weight associated with *Separation* would be increased by 4, the weight associated with *Path* would be increased by 2, and the weight associated with *Level* would be increased by 1. Once all specified quality measures are processed, the algorithm with the largest weight is selected to draw the tree.

In the case of the molecular combinatory trees, each tree was identified as *Random*. We took one tree of each of the six types and specified each quality measure to determine which algorithm the system would choose. The algorithm with the optimal performance for the specified tree and quality measure was selected in every case. Also, when more than one quality measure was selected, the weights for each algorithm chose the algorithm best suited to draw the tree with respect to all specified quality measures. For example, a complete binary tree with selected quality measures **Aspect Ratio** and **Size** gives *Separation* a weight of 12, *Path* a weight of 6, *Level* a weight of 3, and *Rings* a weight of 9, which results in the tree being drawn using *Separation* (See Figure 13).

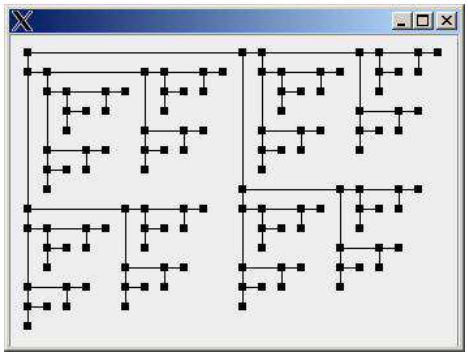


Figure 13: The drawing generated for a complete tree with 127 nodes with user-specified quality measures of **Aspect Ratio** and **Size**. The *Separation* algorithm was selected to create this drawing.

Specifying a third quality measure of **Closest Leaf** results in new weights of 16 for *Separation*, 8 for *Path*, 4 for *Level*, and 17 for *Rings*, making *Rings* the drawing algorithm of choice (See Figure 14).

6 Conclusion and Future Work

Many years of tree drawing research has produced a diverse setting. Experimental studies are important to determine the performance of tree drawing algorithms in real-life applications. In our study, we have experimented with

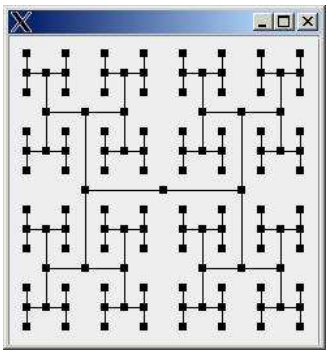


Figure 14: The drawing generated for a complete tree with 127 nodes with user-specified quality measures of **Aspect Ratio**, **Size**, and **Closest Leaf**. The *Rings* algorithm was selected to create this drawing.

four different algorithms: separation-based algorithm by Garg and Rusu [16], path-based algorithm by Chan et al. [8], level-based algorithm by Reingold and Tilford [25], and ringed circular layout algorithm by Teoh and Ma [33]. In general, the algorithms under evaluation exhibit various tradeoffs with respect to the quality measures analyzed, and, in general, none of them perform the best for all categories. The results of our experimental study allowed us to develop an adaptive tree drawing system, which outperforms any single drawing algorithm, by always choosing the best available method of drawing, under the settings of the experimental study. A single algorithm only performs well on specific tree types for specific quality measures. Our adaptive tree drawing system, however, has the potential to perform well on all tree types for a limitless number of quality measures with further enhancements and additional algorithms to select from.

We plan to generate other special types of binary trees, such as k-balanced trees, and extend our study to include algorithms specifically designed for them. We also plan to implement algorithms specifically designed for AVL and Fibonacci trees. For example, if the tree is Fibonacci, (AVL, complete), [10, 34] ([11, 10], respectively) give algorithms for constructing planar straight-line grid drawings with linear area. Also, if the tree belongs to a certain category of balanced trees that includes k-balanced trees, [32] gives an algorithm for constructing a planar straight-line drawing in $O(n \log \log n)$ area, where n is the number of nodes in the tree.

All four algorithms used in our experimental study use a divide and conquer methodology in generating drawings for binary trees. Hence, each algorithm divides the entire tree into smaller subtrees and then draws them recursively. The algorithms create the drawing of the entire tree by connecting the drawings for the subtrees back together. The above considerations suggest the notion of a hybrid strategy that dynamically substitutes different algorithms when drawing separate subtrees of an entire tree. We plan to implement a hybrid tree drawing

algorithm and perform an experimental study, then compare its performance against the four tree drawing algorithms described in this paper.

We believe our approach has the potential of opening a new research area in graph drawing, in which smarter, and thus better, algorithms may be developed by drawing parts of a graph based upon what the system infers from the information stored in the database.

Acknowledgment

We would like to thank the anonymous referees for their very useful comments that have helped in improving the paper.

References

[1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, 4(5):235–282, 1994.

[2] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.

[3] G. Di Battista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. Drawing directed graphs: An experimental study. *International Journal of Computational Geometry and Applications (IJCGA)*, 10(6):623–648, 2000.

[4] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of three graph drawing algorithms (extended abstract). In *ACM Symposium on Computational Geometry*, pages 306–315, 1995.

[5] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry: Theory and Applications (CGTA)*, 7(5-6):303–325, 1997.

[6] F. J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Proceedings 3rd International Symposium on Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 1995.

[7] C. Buchheim, M. Jünger, and S. Leipert. Improving walker's algorithm to run in linear time. In *Proceedings 10th International Symposium on Graph Drawing*, volume 2528 of *Lecture Notes Comput. Sci.*, pages 344–353. Springer, 2002.

[8] T. Chan, M. Goodrich, S. Rao Kosaraju, and R. Tamassia. Optimizing area and aspect ratio in straight-line orthogonal tree drawings. *Comput. Geom. Theory Appl.*, 23:153–162, 2002.

[9] E. H. Chi and S. K. Card. Sensemaking of evolving web sites using visualization spreadsheets. In *Proceedings of the Symposium on Information Visualization*, pages 18–25. IEEE Press, 1999.

[10] P. Crescenzi, G. Di Battista, and A. Piperno. A note on optimal area algorithms for upward drawings of binary trees. *Comput. Geom. Theory Appl.*, 2(4):187–200, 1992.

[11] P. Crescenzi, P. Penna, and A. Piperno. Linear area upward drawings of AVL trees. *Comput. Geom. Theory Appl.*, 9(1-2):25–42, 1998.

[12] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume 1. North–Holland, Amsterdam, 1958.

[13] A. Garg and A. Rusu. Straight-line drawings of binary trees with linear area and arbitrary aspect ratio. In *Proceedings of* 10^{th} *International Symposium on Graph Drawing*, volume 2528 of *Lecture Notes Comput. Sci.*, pages 320–331. Springer, 2002.

[14] A. Garg and A. Rusu. Area-efficient order-preserving planar straight-line drawings of ordered trees. *International Journal of Computational Geometry and Applications (IJCGA)*, 13(6):487–505, 2003.

[15] A. Garg and A. Rusu. Straight-line drawings of general trees with linear area and arbitrary aspect ratio. In *Proceedings 2003 International Conference on Computational Science And Its Applications (ICCSA 2003)*, volume 2669 of *Lecture Notes Comput. Sci.*, pages 876–885. Springer, 2003.

[16] A. Garg and A. Rusu. Straight-line drawings of binary trees with linear area and arbitrary aspect ratio. *J. Graph Algorithms Appl.*, 8(2):135–160, 2004.

[17] M. Himsolt. Comparing and evaluating layout algorithms within graphed. *Journal of Visual Languages and Computing*, 6(3):255–273, 1995.

[18] S. Jones, P. Eades, A. Moran, N. Ward, G. Delott, and R. Tamassia. A note on planar graph drawing algorithms. Technical Report 216, Department of Computer Science, University of Queensland, 1991.

[19] M. Jünger and P. Mutzel. Exact and heuristic algorithms for 2-layer straight-line crossing minimization. In *Proceedings of* 3^{rd} *International Symposium on Graph Drawing*, volume 1027 of *Lecture Notes in Computer Science*, pages 337–348. Springer-Verlag, 1996.

[20] M. Jünger and P. Mutzel. *Graph Drawing Software*. Springer, 2003.

[21] B. MacLennan. Molecular combinatory computing for nanostructure synthesis and control. In *Proceedings* 3^{rd} *IEEE Conference on Nanotechnology*, volume 2, pages 179–182. IEEE Press, 2003.

[22] G. Melancon and I. Herman. Circular drawings of rooted trees. Technical Report INS-9817, Netherlands' National Research Institute for Mathematics and Computer Sciences, 1998.

[23] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 248–261. Springer-Verlag, 1997.

[24] H. C. Purchase, R. F. Cohen, and M. I. James. An experimental study of the basis for graph drawing algorithms. *ACM J. Experim. Algorithmics*, 2(4):4, 1997.

[25] E. Reingold and J. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[26] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194. ACM Press, 1991.

[27] A. Rusu, C. Clement, and R. Jianu. Performance analysis of a path-based algorithm for drawing binary trees. In *Proc. 5th International Conference on Artificial Intelligence and Digital Communications*, Research Notes in Computer Science, pages 84–102, 2005.

[28] A. Rusu, C. Clement, and R. Jianu. Adaptive binary trees visualization with respect to user-specified quality measures. In *IV '06: Proceedings of the International Conference on Information Visualisation*, pages 469–474. IEEE Computer Society, 2006.

[29] A. Rusu, R. Jianu, C. Santiago, and C. Clement. An experimental study on algorithms for drawing binary trees. In *APVis '06: Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation*, pages 85–88. Australian Computer Society, 2006.

[30] A. Rusu and C. Santiago. A practical algorithm for planar straight-line grid drawings of general trees with linear area and arbitrary aspect ratio. In *IV '07: Proceedings of the International Conference on Information Visualisation*, pages 743–750. IEEE Computer Society, 2007.

[31] M. Sarkar and M. H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–83, 1994.

[32] C. Shin, S. K. Kim, and K. Chwa. Area-efficient algorithms for straight-line tree drawings. *Comput. Geom. Theory Appl.*, 15(4):175–202, 2000.

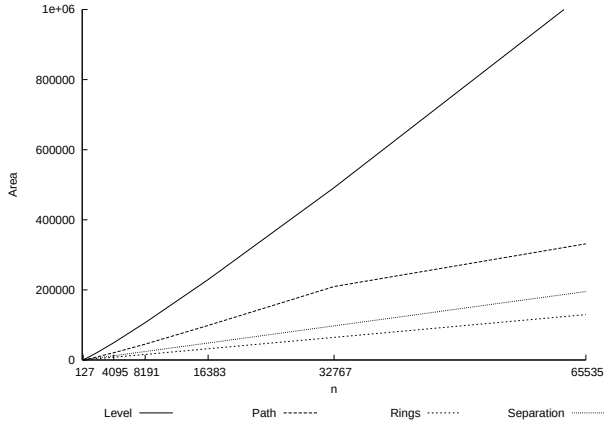
[33] S. T. Teoh and K. L. Ma. RINGS: A technique for visualizing large hierarchies. In *Proceedings 10th International Symposium on Graph Drawing*, volume 2528 of *Lecture Notes in Comput. Sci.*, pages 268–275. Springer, 2002.

[34] L. Trevisan. A note on minimum-area upward drawing of complete and fibonacci trees. *Information Processing Letters*, 57(5):231–236, 1996.

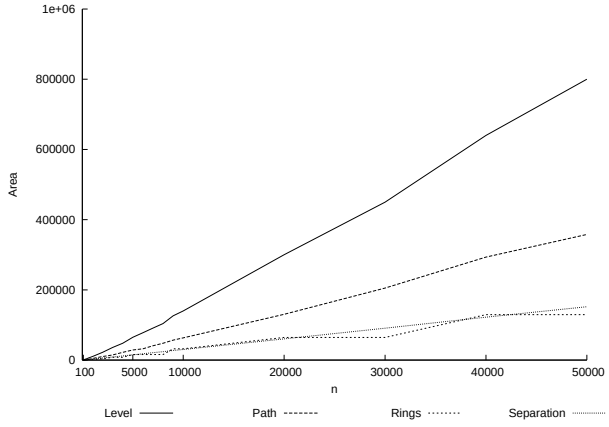
[35] L. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Comput.*, C-30(2):135–140, 1981.

[36] L. Vismara, G. Di Battista, A. Garg, G. Liotta, R. Tamassia, and F. Vargiu. Experimental studies on graph drawing algorithms. *Software Practice and Experience Journal*, 30(11):1235–1284, 2000.

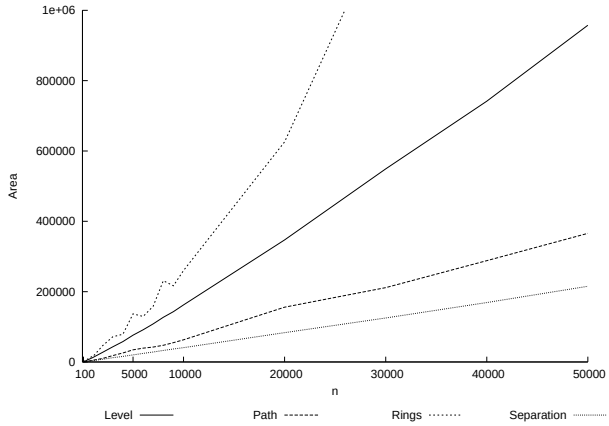
[37] J. Q. Walker. A node-positioning algorithm for general trees. *Software Practice and Experience Journal*, 20(7):685–705, 1990.



(a)



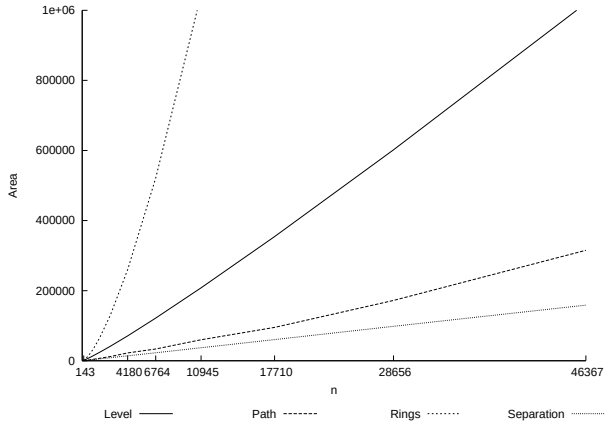
(b)



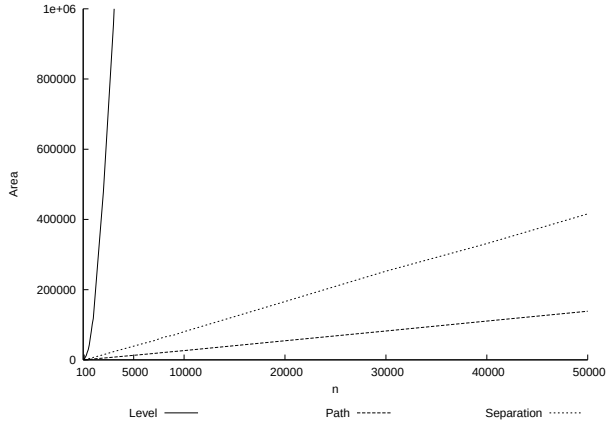
(c)

(Figure 15 continued on the next page)

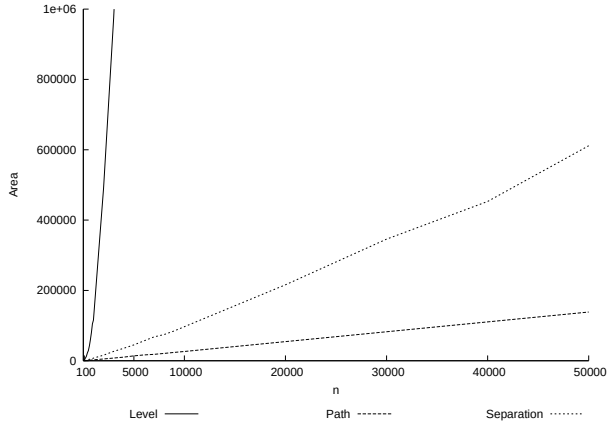
(figure continued from the previous page)



(d)



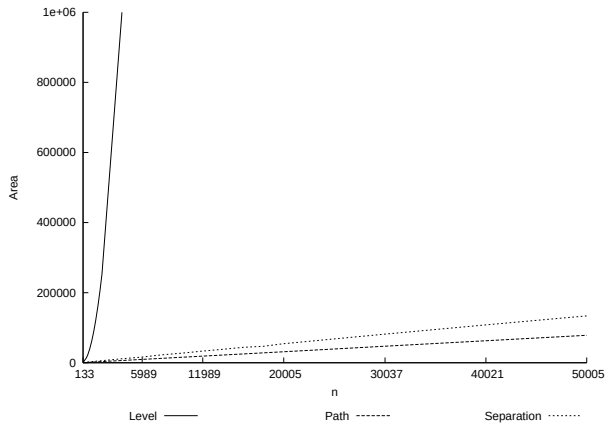
(e)



(f)

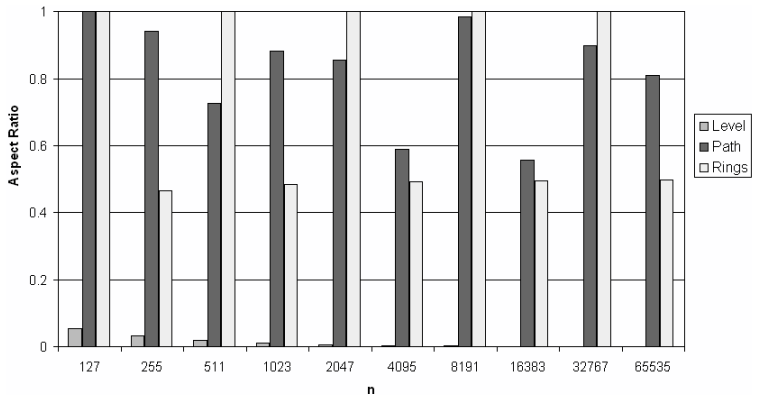
(Figure 15 continued on the next page)

(figure continued from the previous page)

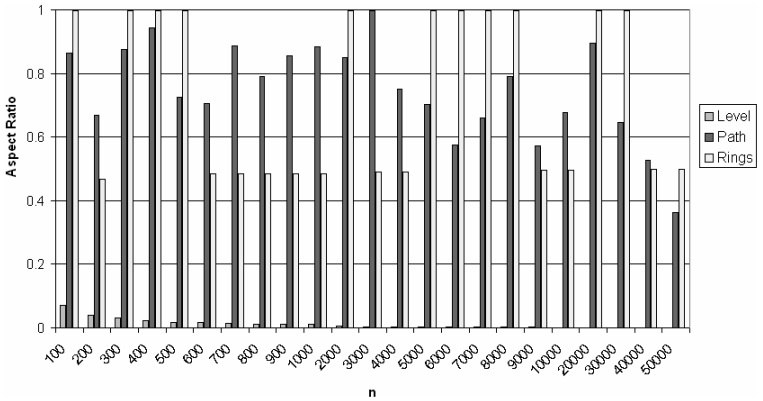


(g)

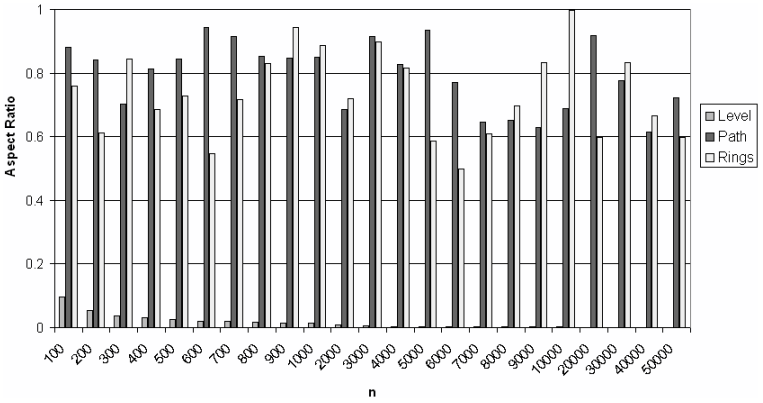
Figure 15: Comparison charts of the area for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary tree, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



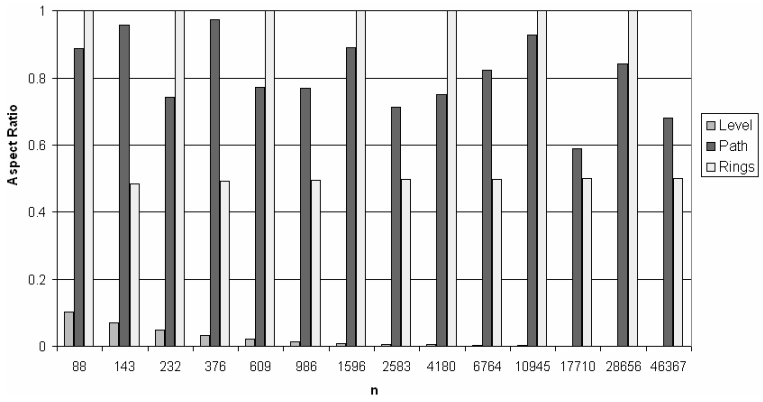
(b)



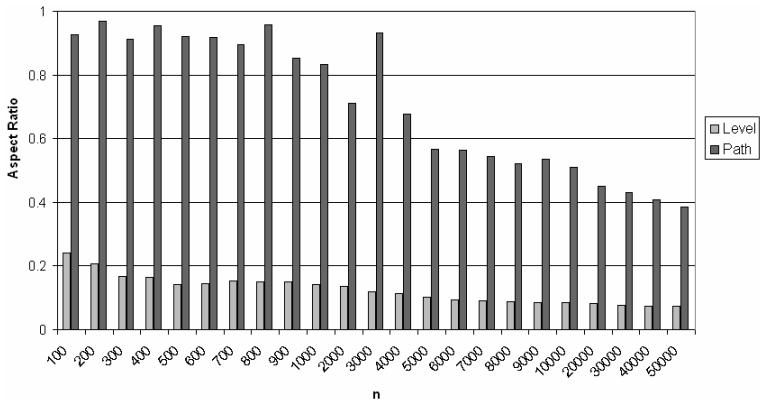
(c)

(Figure 16 continued on the next page)

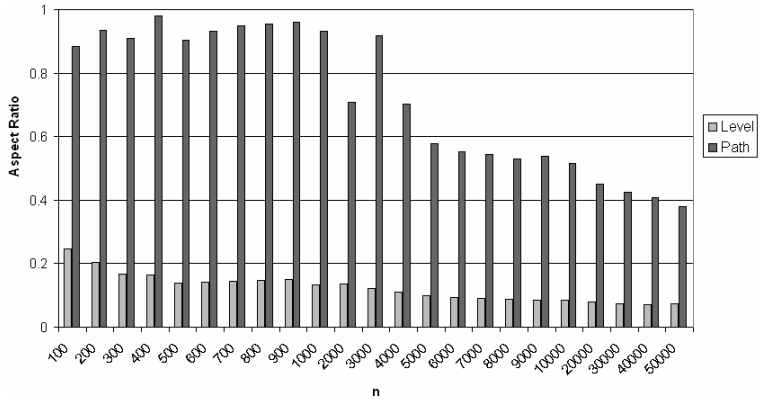
(figure continued from the previous page)



(d)



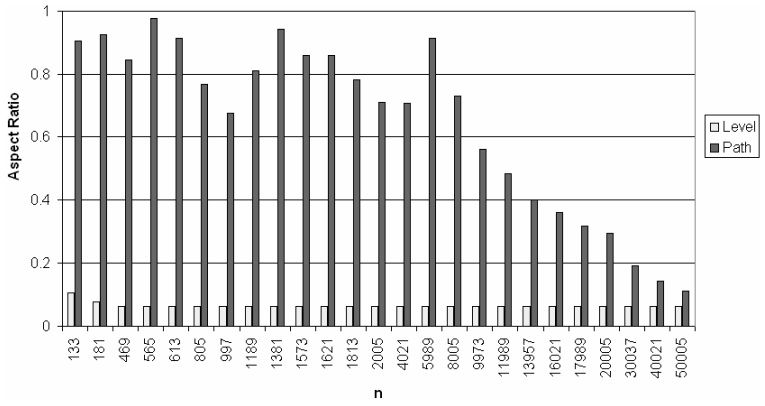
(e)



(f)

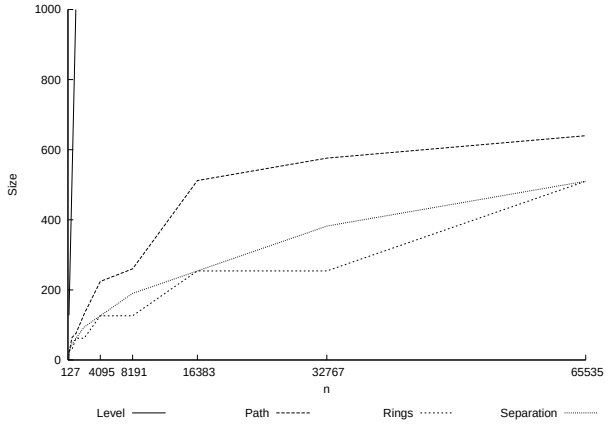
(Figure 16 continued on the next page)

(figure continued from the previous page)

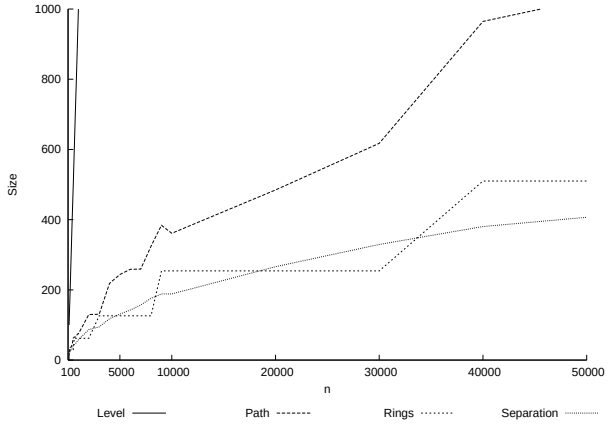


(g)

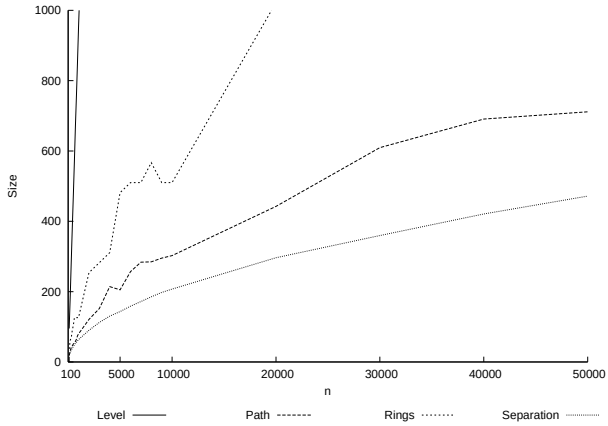
Figure 16: Comparison charts of the aspect ratio for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary tree, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



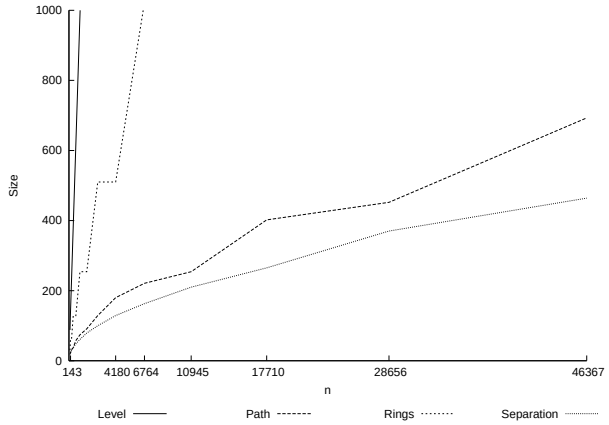
(b)



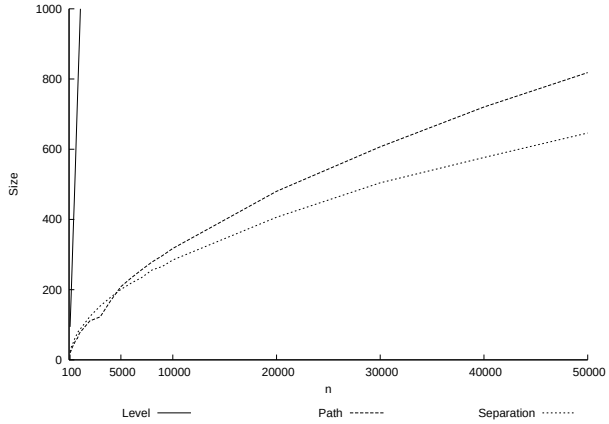
(c)

(Figure 17 continued on the next page)

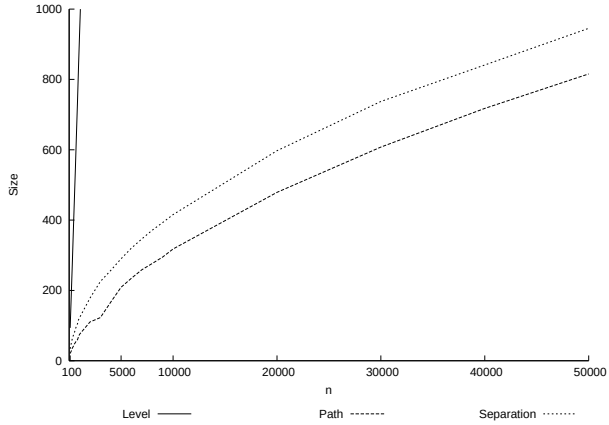
(figure continued from the previous page)



(d)



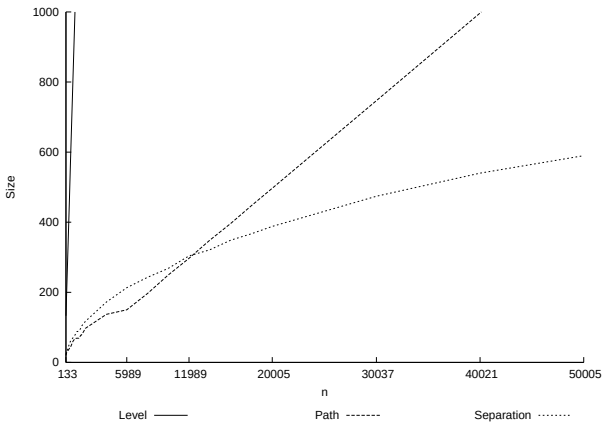
(e)



(f)

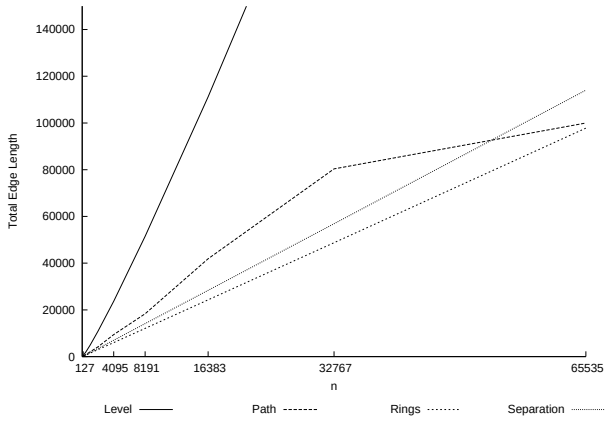
(Figure 17 continued on the next page)

(figure continued from the previous page)

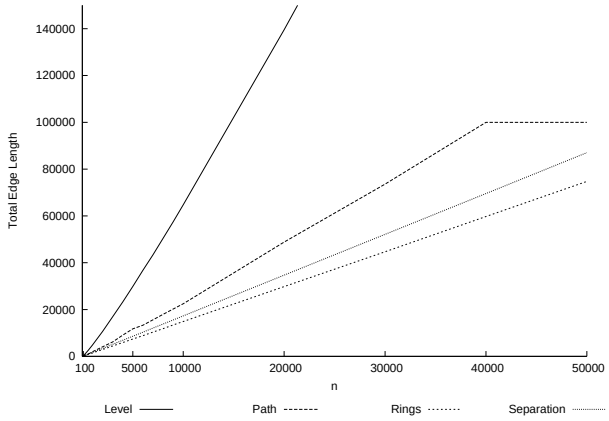


(g)

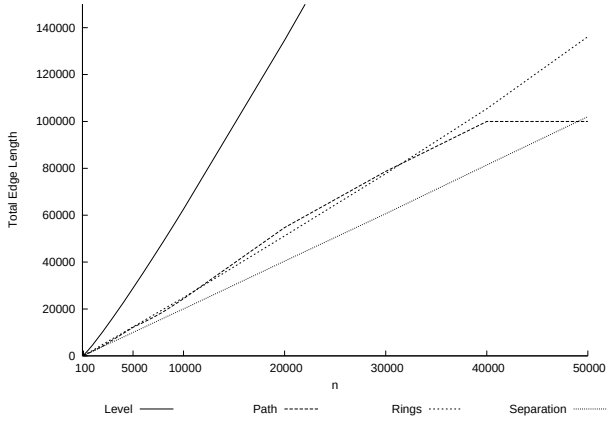
Figure 17: Comparison charts of the size for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary tree, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



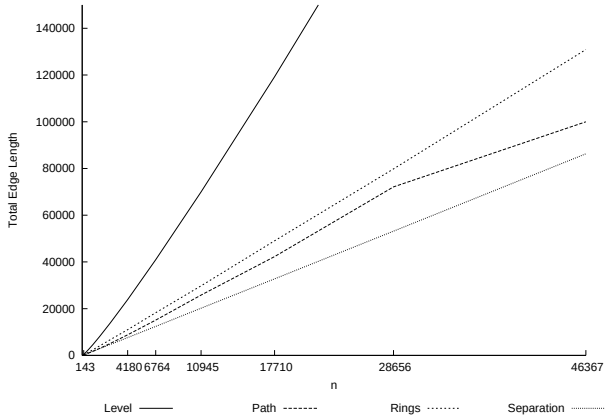
(b)



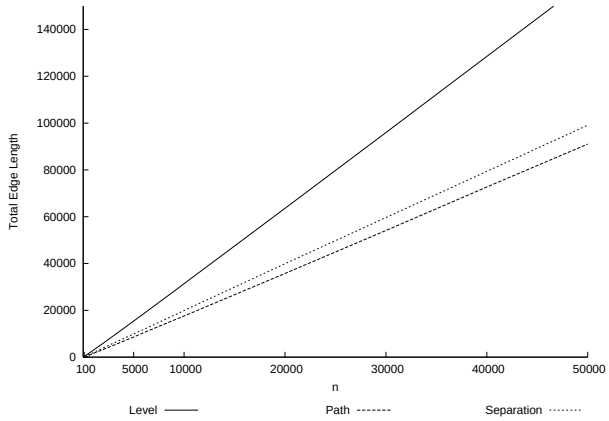
(c)

(Figure 18 continued on the next page)

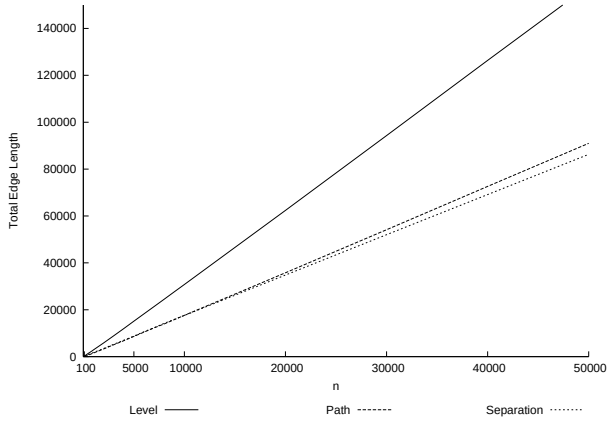
(figure continued from the previous page)



(d)



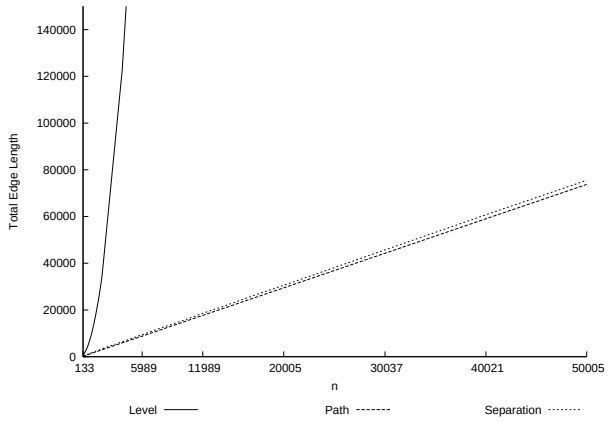
(e)



(f)

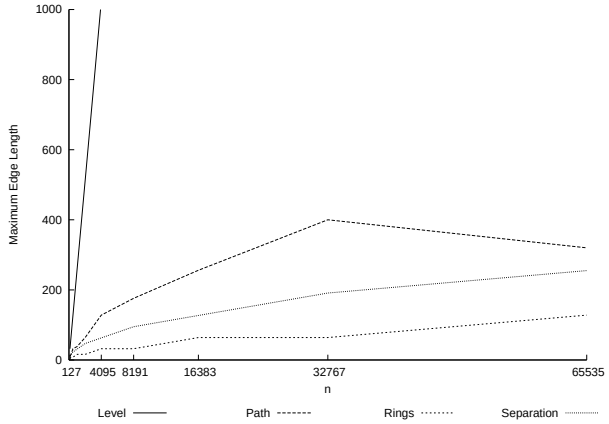
(Figure 18 continued on the next page)

(figure continued from the previous page)

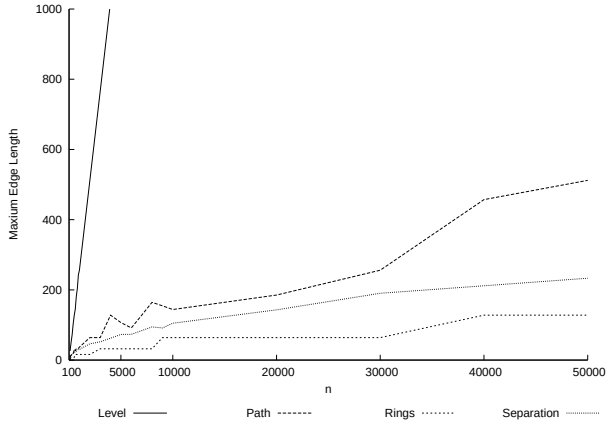


(g)

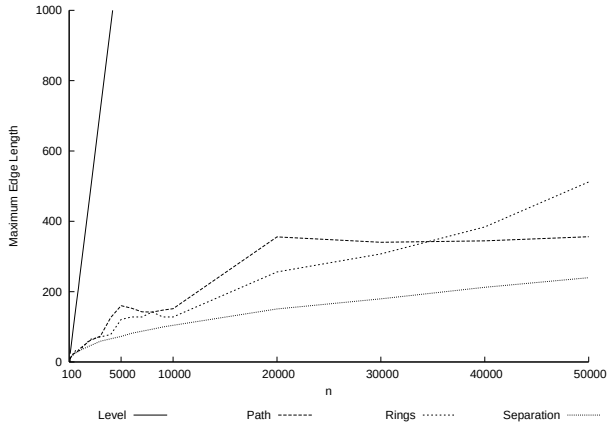
Figure 18: Comparison charts of the total edge length for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary tree, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



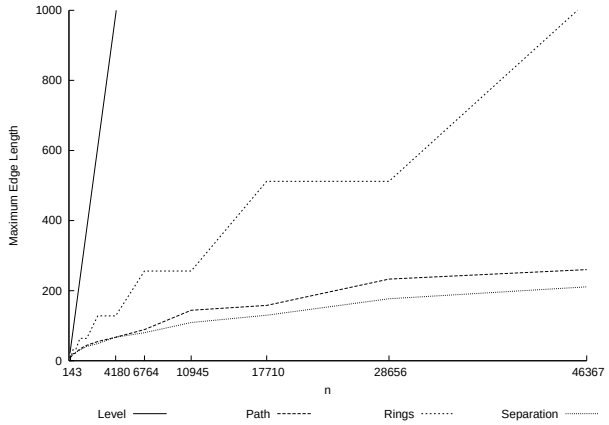
(b)



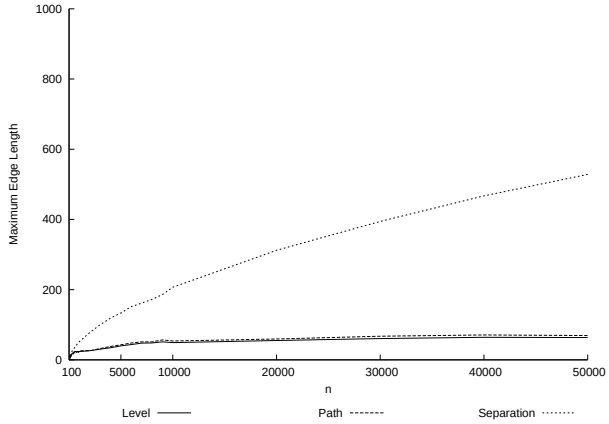
(c)

(Figure 19 continued on the next page)

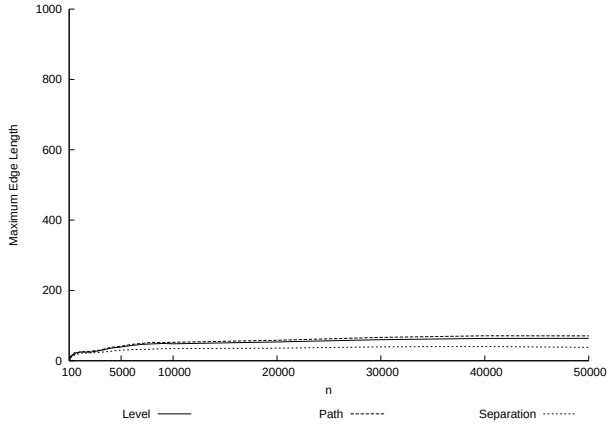
(figure continued from the previous page)



(d)



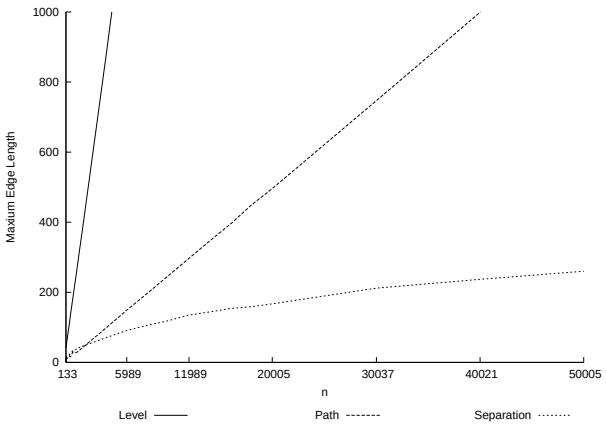
(e)



(f)

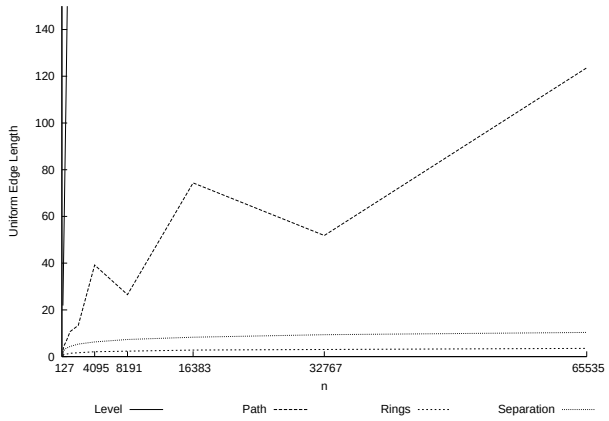
(Figure 19 continued on the next page)

(figure continued from the previous page)

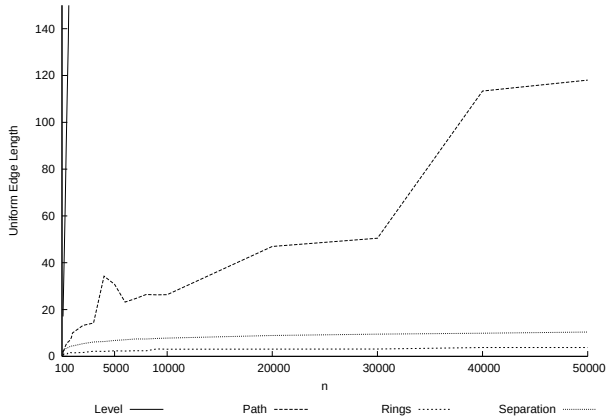


(g)

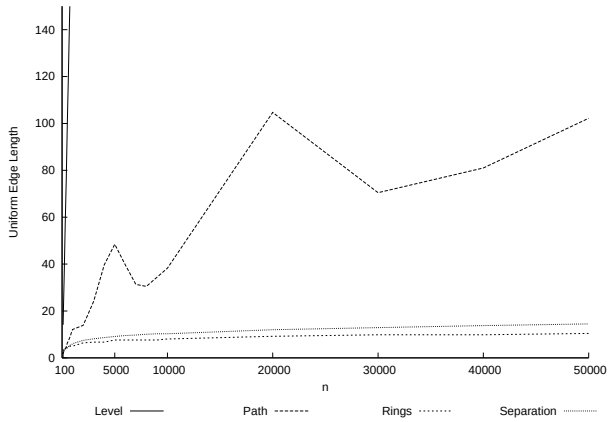
Figure 19: Comparison charts of the maximum edge length for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary tree, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



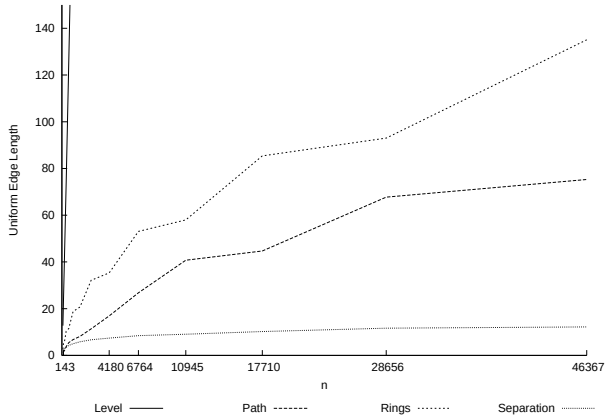
(b)



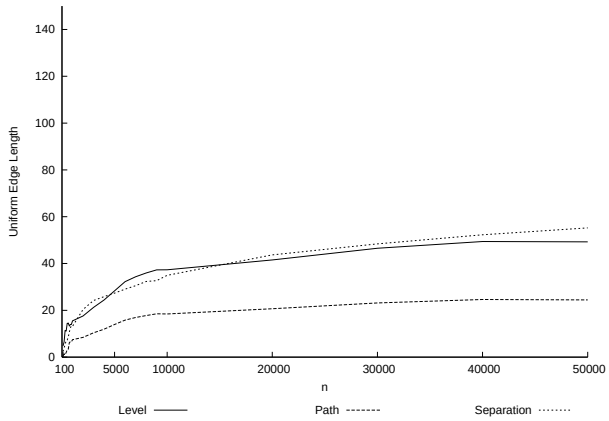
(c)

(Figure 20 continued on the next page)

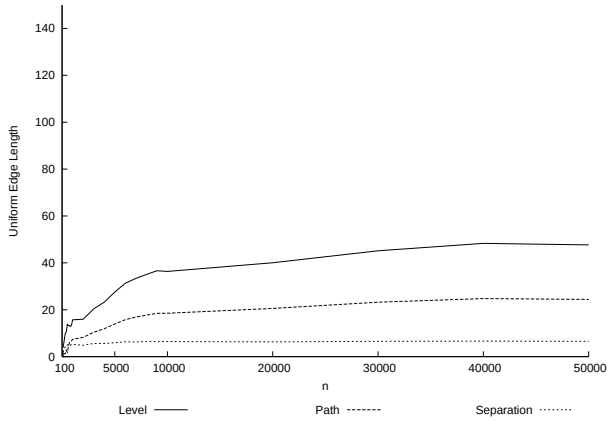
(figure continued from the previous page)



(d)



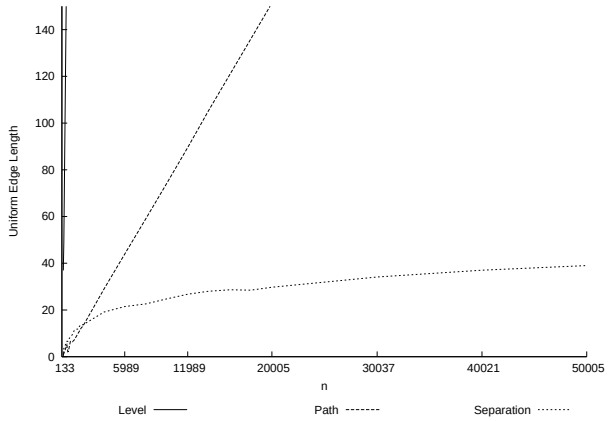
(e)



(f)

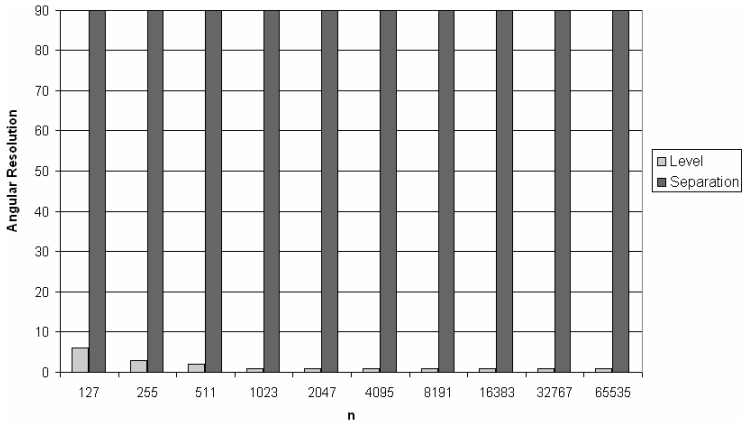
(Figure 20 continued on the next page)

(figure continued from the previous page)

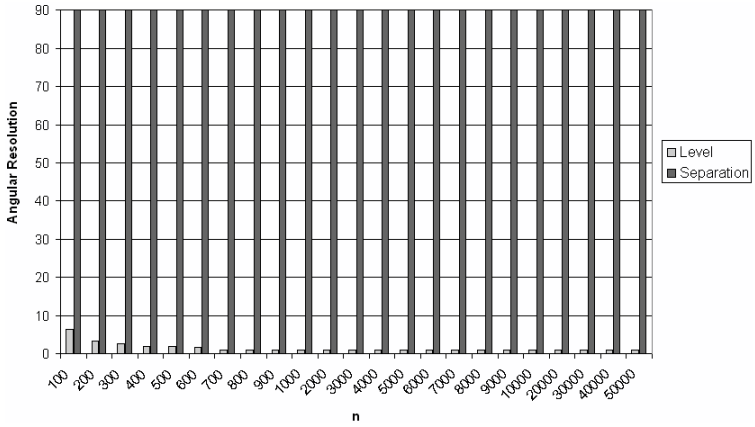


(g)

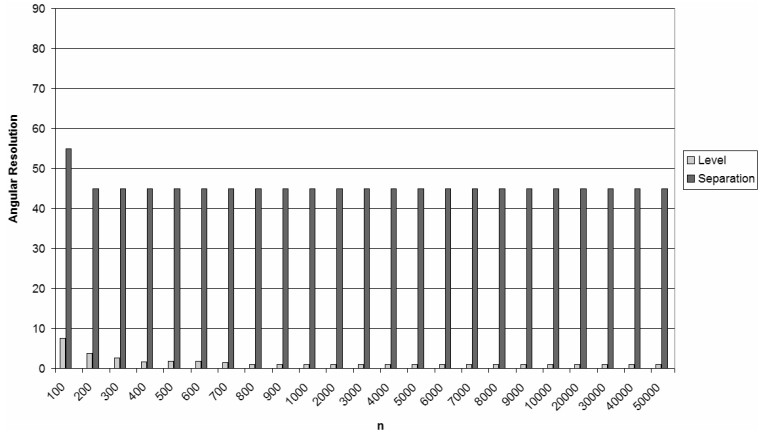
Figure 20: Comparison charts of the uniform edge length for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary tree, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



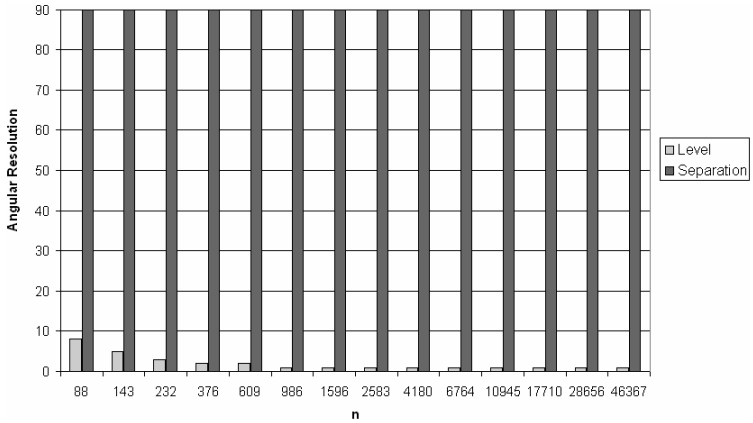
(b)



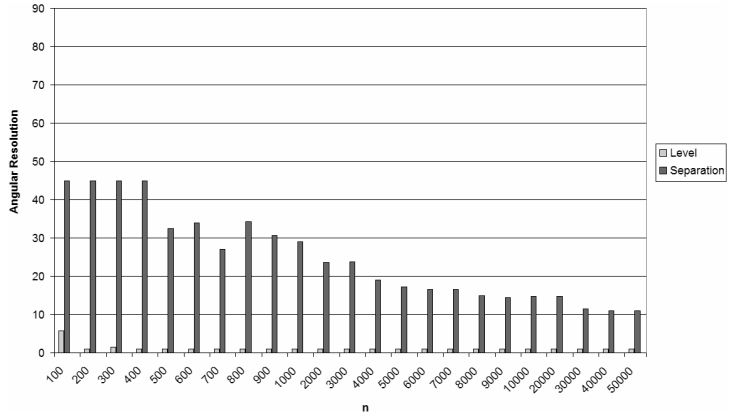
(c)

(Figure 21 continued on the next page)

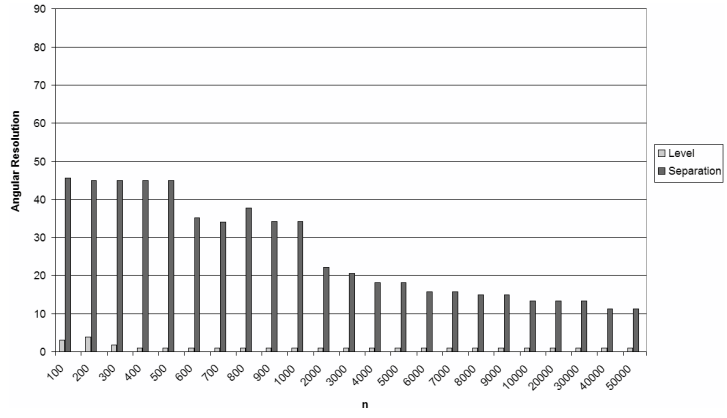
(figure continued from the previous page)



(d)



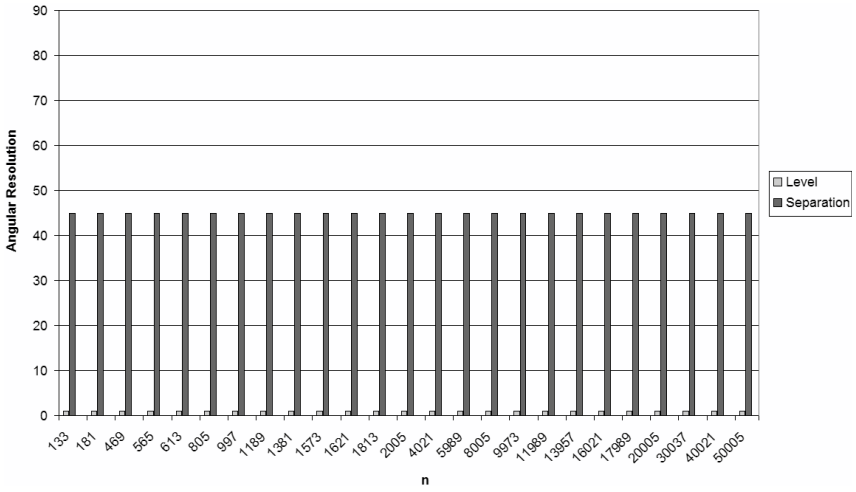
(e)



(f)

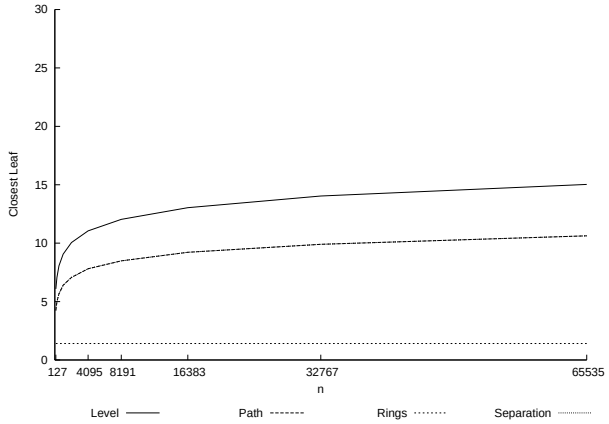
(Figure 21 continued on the next page)

(figure continued from the previous page)

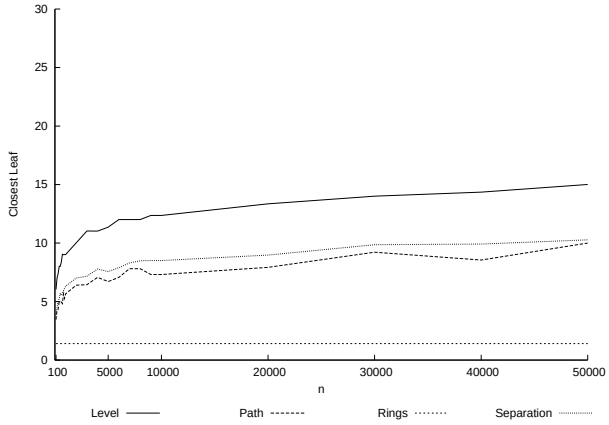


(g)

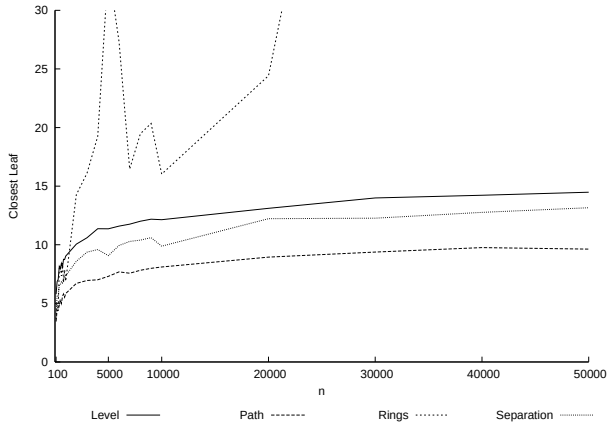
Figure 21: Comparison charts of the angular resolution for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary trees, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



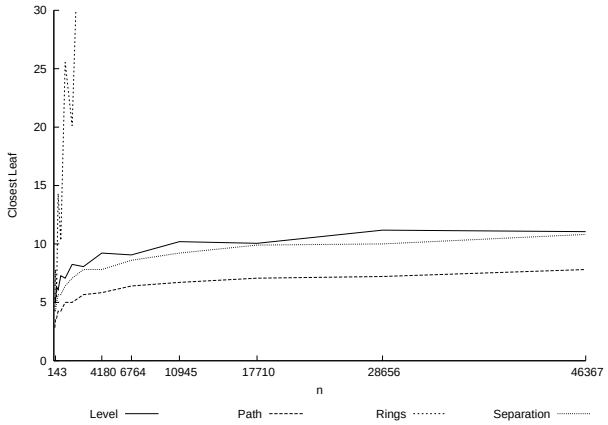
(b)



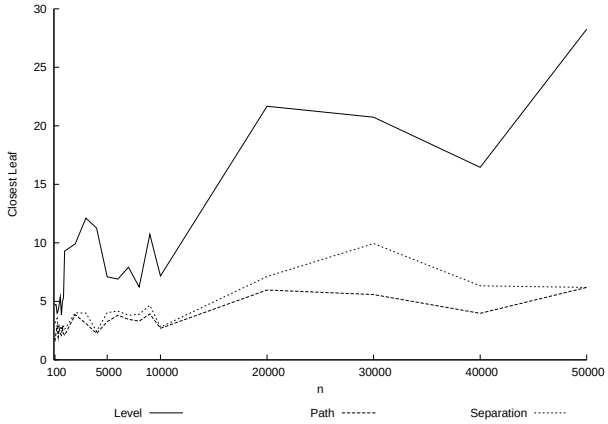
(c)

(Figure 22 continued on the next page)

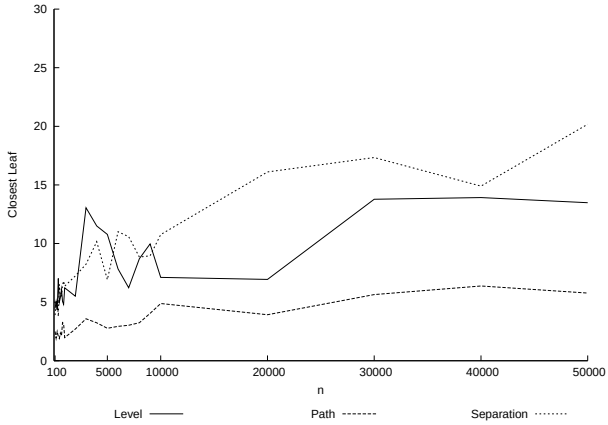
(figure continued from the previous page)



(d)



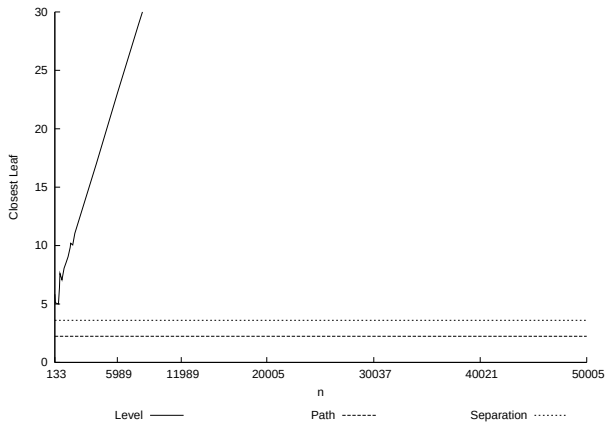
(e)



(f)

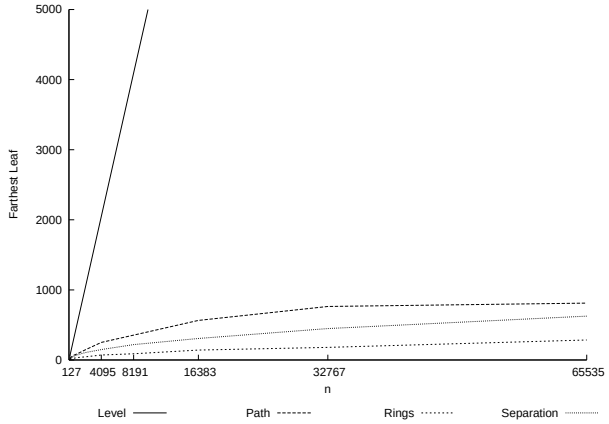
(Figure 22 continued on the next page)

(figure continued from the previous page)

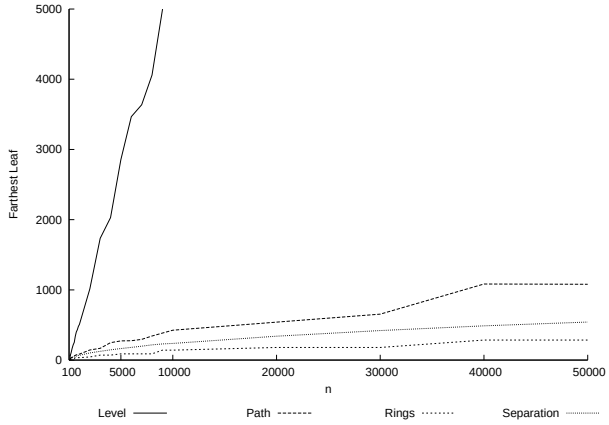


(g)

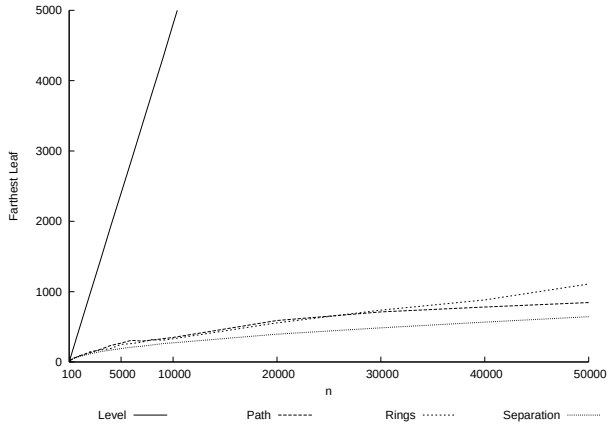
Figure 22: Comparison charts of the closest leaf for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary trees, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.



(a)



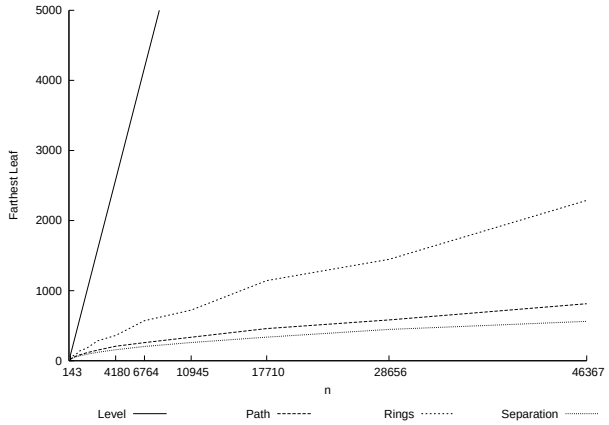
(b)



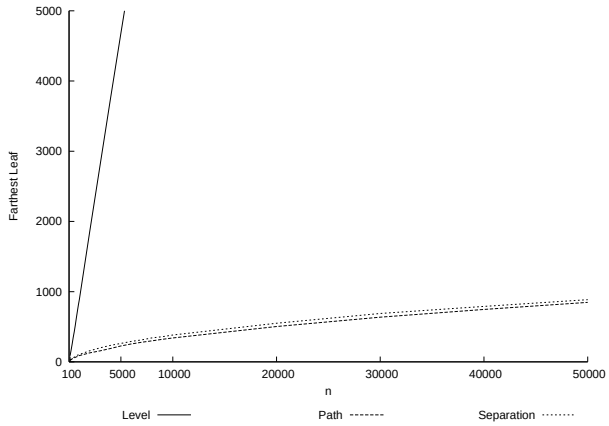
(c)

(Figure 23 continued on the next page)

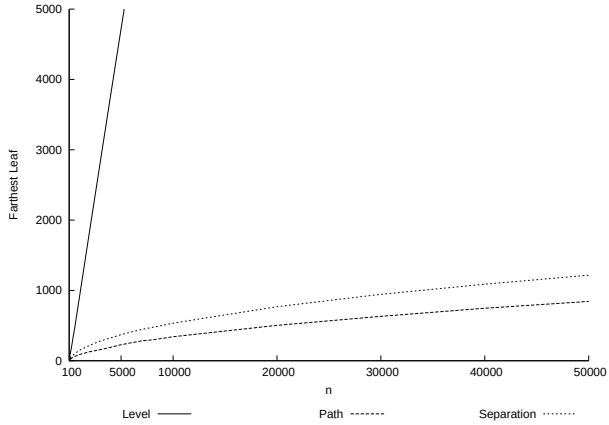
(figure continued from the previous page)



(d)



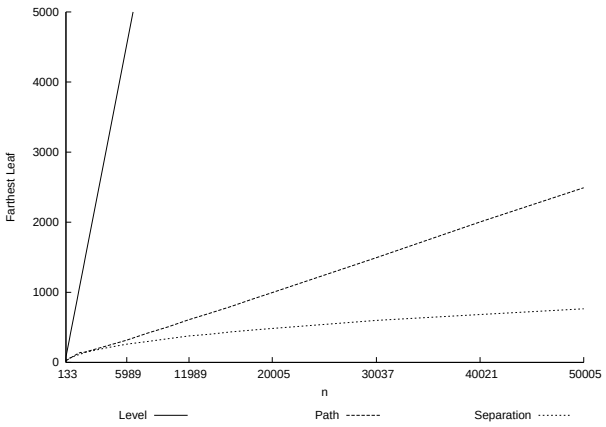
(e)



(f)

(Figure 23 continued on the next page)

(figure continued from the previous page)



(g)

Figure 23: Comparison charts of the farthest leaf for *Level*, *Path*, *Rings*, and *Separation*, for each tree-type: (a) Complete binary trees, (b) AVL trees, (c) Randomly-generated binary trees, (d) Fibonacci trees, (e) Unbalanced-to-the-left binary trees, (f) Unbalanced-to-the-right binary trees, (g) Molecular Combinatory binary trees.