Quad general tree drawing algorithm and general trees characterization: towards an environment for the experimental study on general tree drawing algorithms

Chu Yao
Rowan University

Quad GENERAL TREE DRAWING ALGORITHM AND GENERAL TREES CHARACTERIZATION: TOWARDS AN ENVIRONMENT FOR THE EXPERIMENTAL STUDY ON GENERAL TREE DRAWING ALGORITHMS

by

Chu Yao

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical and Computer Engineering
Major: Engineering (Electrical Engineering)

Approved:	Members of the Committee
______________________	[illegible]
In Charge of Major Work	
[illegible]	[illegible]
For the Major Department	
[illegible]	______________________
For the College	

Rowan University
Glassboro, New Jersey
2008

ABSTRACT

Chu Yao
Quad GENERAL TREE DRAWING ALGORITHM AND GENERAL TREES CHARACTERIZATION: TOWARDS AN ENVIRONMENT FOR THE EXPERIMENTAL STUDY ON GENERAL TREE DRAWING ALGORITHMS
2008/05
Advisor: Dr. Adrian Rusu
Master of Science in Engineering

Information visualization produces (interactive) visual representations of abstract data to re-inforce human cognition and perception; thus enabling the viewer to gain knowledge about the internal structure of the data and causal relationships in it. The visualization of information hierarchies is concerned with the presentation of abstract hierarchical information about relationships between various entities. It has many applications in diverse domains such as software engineering, information systems, biology, and chemistry. Information hierarchies are typically modeled by an abstract tree, where vertices are entities and edges represent relationships between entities. The aim of visualizing tree drawings is to automatically produce drawings of trees which clearly reflect the relationships of the information hierarchy.

This thesis is primarily concerned with introducing the new general tree drawing algorithm *Quad* that produces good visually distinguishable angles, and a characterization of general trees which allows us to classify general trees into several types based on their characteristics. Both of these topics are part of building an experimental study environment for the evaluation of drawing

algorithms for general trees.

The main achievements of this thesis include:

1. A study on characterization of general trees that aims to classify them into several types.

2. A tree drawing algorithm that produces visually distinguishable angles for high degree general trees with user specified angular coefficient.

To my grandfather Yao, Yan Xiang and in the everlasting memory of my grandmother He, Jin Ying.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep appreciation to my major advisor, Dr. Adrian Rusu, for his clear guidance, encouragement, wise counsel, and unwavering support over the past few years. His influence has been invaluable in my development as a researcher. This thesis would not be possible without his continuous help, motivation, and insightful advice. I also want to express my respects for his incredible enthusiasm for research. Many thanks for supporting me as a Research Assistant. I will always be grateful for the opportunities he created for me. His hard work and dedication to me, as well as other students, will never be forgotten.

I would like to thank the members of Software Engineering, Graphics, and Visualization (SEGV) Research Group including Mr. Confesor Santiago and Mr. Andrew Crowell for their help and effort.

Special thanks to my parents, Jian Chi Yao and Zhi Ying Wang for their initial advice that led me to attempt graduate school, and my parents in-law, Toi Yee Lu and Wan Rong Lin for taking care of my son during my graduate school years.

Last but not least, super special thanks to my wife, Hui Bin Lu for her unconditional love and companionship, and my son, Yan Yao, watching his birth and growing up bring me so much happiness and motivation.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Tree Drawing

Tree drawing is concerned with the automatic generation of geometric representations of relational information, often for visualization purposes. The typical data structure for modeling hierarchical information is a tree whose vertices represent entities and whose edges correspond to relationships between entities. Visualizations of hierarchical structures are only useful to the degree that the associated diagrams effectively convey information to the people that use them. A good diagram helps the reader understand the system, but a poor diagram can be confusing [11].

The automatic generation of drawings of trees finds many applications, such as

- software engineering (program nesting trees, object-oriented class hierarchies - *see Figure 1.1*),
- information systems (organization charts - *see Figure 1.2*, course flow charts - *see Figure 1.3*),
- decision support systems (activity trees - *see Figure 1.4*),
- artificial intelligence (knowledge-representation hierarchies - *see Figure 1.5*),
- logic programming (SLD-trees - *see Figure 1.6*).

Further applications can be found in other science and engineering disciplines, such as

- biology (evolutionary trees),

- chemistry (molecular drawings).

The usefulness of a drawing of a tree depends on its *readability*, i.e, its capability of conveying the information contained in the tree quickly and clearly.

Tree drawing algorithms are methods that produce tree drawings which are easy to read. Algorithms for drawing trees are typically based on some graph-theoretic insight into the structure of the tree. The input to a tree drawing algorithm is a tree T that needs to be drawn. The output is a drawing Γ which maps each vertex of T to a distinct point in the 2D space and each edge (u,v) of T to a simple Jordan curve with endpoints u and v.

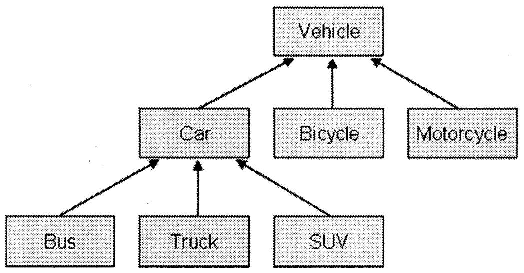


Figure 1.1 Object hierarchy drawing

1.2 Tree Drawing Conventions

In this thesis we consider planar straight-line grid drawings which exhibit the subtree separation property. Now we explain the properties of these drawings and the motivation behind using them.

1.2.1 Grid Drawings

A *grid drawing* is one in which each vertex is placed at integer coordinates (see Figure 1.7(a)). Grid drawings guarantee at least unit distance separation between nodes, and the integer coordinates

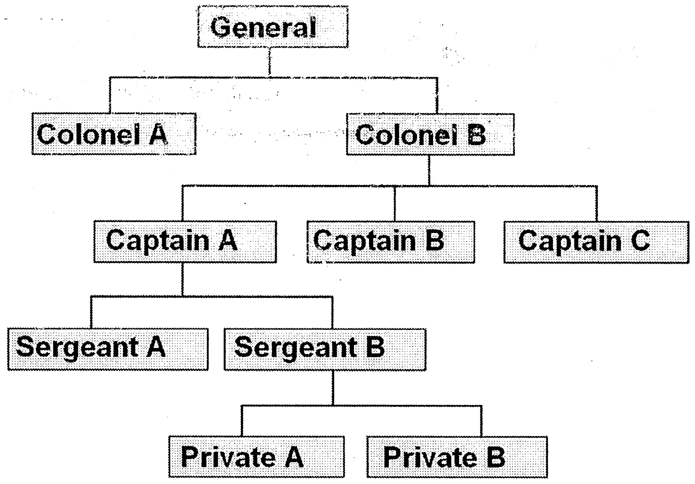


Figure 1.2 Organization chart.

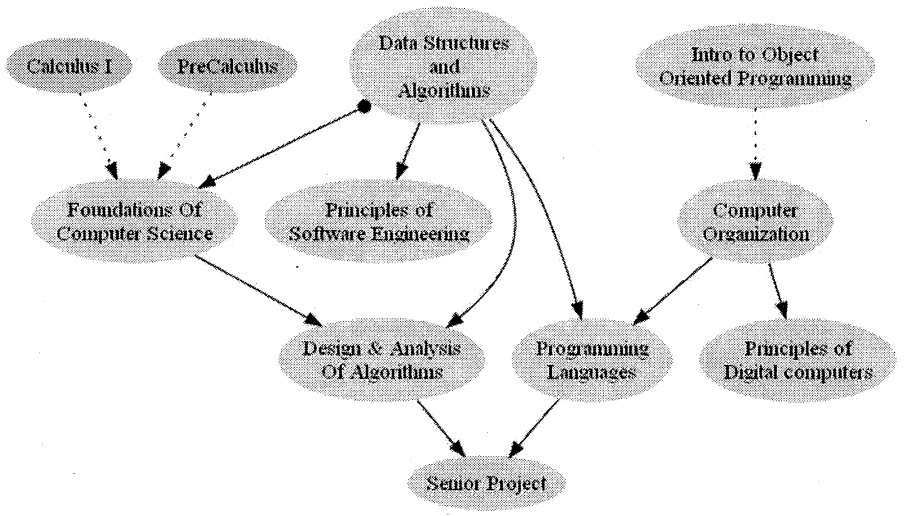

Calculus I
PreCalculus
Data Structures and Algorithms
Intro to Object Oriented Programming
Foundations Of Computer Science
Principles of Software Engineering
Computer Organization
Design & Analysis Of Algorithms
Programming Languages
Principles of Digital computers
Senior Project

Figure 1.3 Tree-like flow chart of Rowan University Computer Science Department's course offering

of nodes allow such drawings to be rendered on displays, such as a computer screen, without distortions due to truncation and round-off errors. We assume that the plane is covered by *horizontal* and *vertical channels*, with unit distance between two consecutive channels. The meeting point of a horizontal and a vertical channel is called a *grid-point.* The smallest rectangle with horizontal and

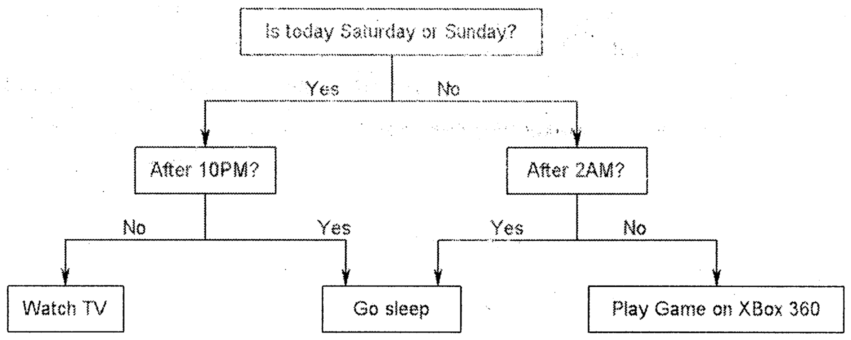


Figure 1.4 Decision supported activity tree-like graph.

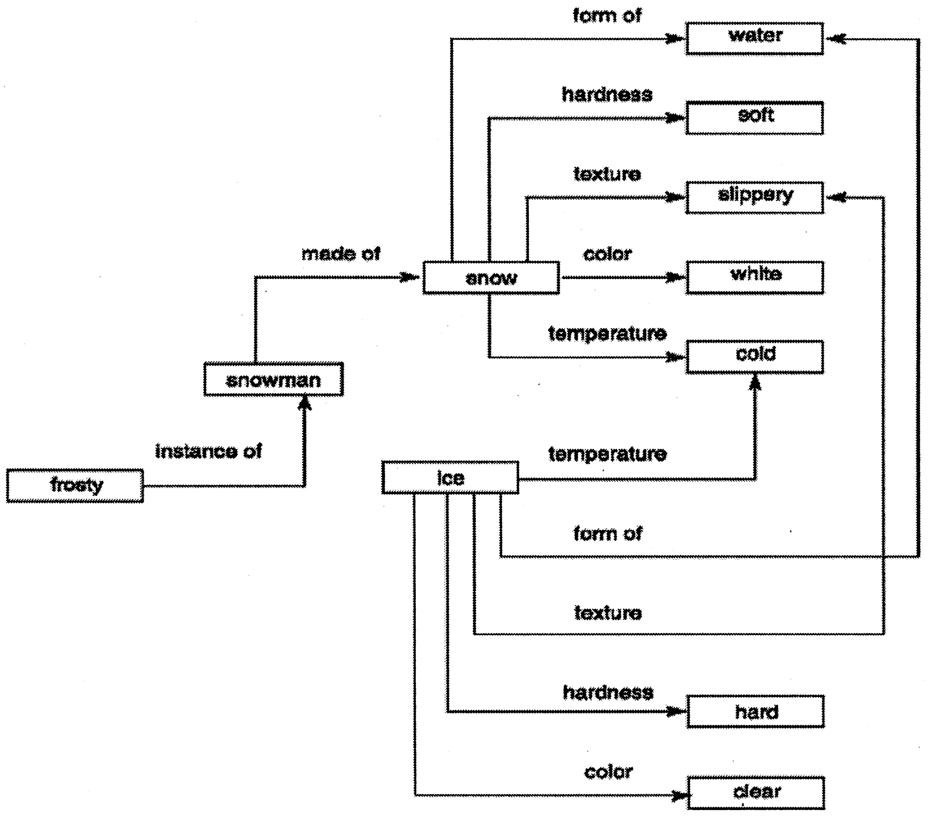


Figure 1.5 Knowledge representation graph: nodes are concepts, edges are relations between concepts

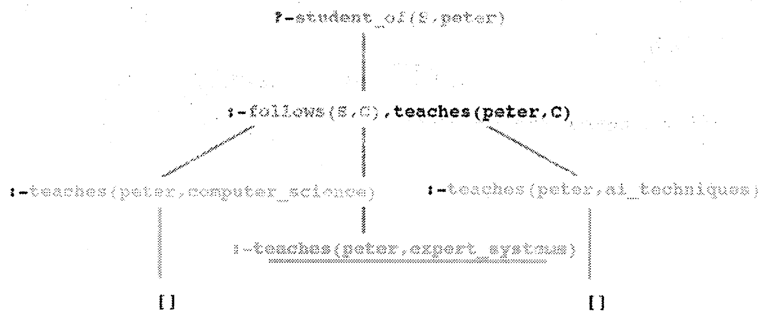

?-student_of(S,peter)
:-follows(S,C),teaches(peter,C)
:-teaches(peter,computer_science)
:-teaches(peter,expert_systems)
:-teaches(peter,ai_techniques)
[]
[]

Figure 1.6 SLD tree

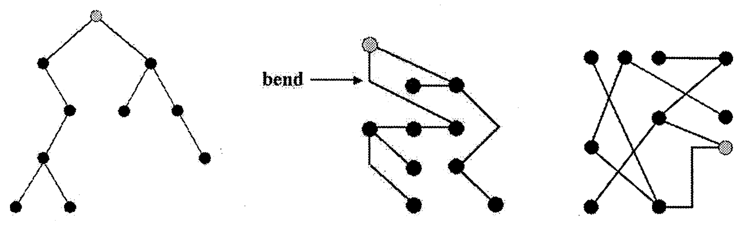


Figure 1.7 Grid drawings of the same tree: (a) straight-line; (b) polyline; (c) non-planar. The root of the tree is shown as a shaded circle, whereas other nodes are shown as black circles.

vertical sides parallel to the X and Y axis, that covers the entire grid drawing, is called the *enclosing rectangle*. The *area* of a grid drawing is defined as the number of grid points contained in its enclosing rectangle. Drawings with small area can be drawn with greater resolution on a fixed-size page. The *aspect ratio* of a grid drawing is defined as the ratio of the length of the longest side to the length of the shortest side of its enclosing rectangle. The optimal use of the screen space is achieved by minimizing the area of the drawing and by providing user-controlled aspect ratio.

1.2.2 Planar Drawings

A *planar drawing* is a drawing in which no two edges cross (see Figure 1.7(a) and (b)). Planar drawings are normally easier to understand than non-planar drawings (see Figure 1.7(c)), i.e.

drawings with edge-crossings. Planarity is also an important tree theoretic concept, which has been widely studied. Extensive research has been done on various kinds of planar drawings. For example, [6–8, 10, 12–14, 19, 22, 27, 30, 31] provide important results.

1.2.3 Straight-line Drawings

It is natural to draw each edge of a tree as a straight line between its end-vertices. The so called *straight-line* tree drawings have each edge drawn as a straight line segment (see Figure 1.7(a)). Straight-line drawings are easier to understand than polyline drawings (see Figure 1.7(b) and (c)), i.e. drawings in which edges have bends (more than one line segment). The experimental study of the human perception of tree drawings has concluded that minimizing the number of edge crossings and minimizing the number of bends increases the understandability of drawings of trees [20,21,28]. Ideally, the drawings should have no edge crossings, i.e. they should be planar drawings, and should have no edge-bends, i.e. they should be straight-line drawings.

1.2.4 Subtree Separation

A drawing of a tree T has the *subtree separation* property [6] 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 [26] 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.

1.3 Types of General Trees

Currently, there are not many general tree types that have been defined rigorously. This is a consequence of most research in tree drawing being applied to binary trees. Another reason is general trees' unpredictability. A binary tree's size and width are restricted by it's height, but a general tree's size and width do not have a strong connection to its height. This is also the reason why only a small number of researchers have devoted time to general trees. One part of this thesis is the definition of a few types for general trees based on their characteristics.

1.4 Environment for Experimental Study of General Tree Drawing Algorithms

An experiment setting for comparing the practical performance of drawing algorithms for general trees consists of (i) A simple format for storing general trees in text files; (ii) Save/load routines for generating general trees to files and for uploading general trees from files, respectively; (iii) a large suite of various types of general trees of various sizes; (iv) Quality measures such as area, aspect ratio, total edge length, average edge length, maximum edge length, minimum angle size, average angle size, closest leaf, and farthest leaf. As part of my research work, I contributed to item (i), (ii), and the ability of classfying general trees in various types based on their characteristics.

1.5 Outline of This Thesis

First, we analyze the general trees in more detail with few new and existing characteristics. Next, we present *Quad*, a novel drawing algorithm for general trees that achieves better visually distinguishable edge angles compared to existing general tree drawing algorithms. These characteristics will be used to define types for general trees. These definitions are critical contributions towards building an experimental study environment for general tree drawing algorithms. We now outline the structure of this thesis and summarize the principal results obtained: (Note that each chapter is self-contained)

- In Chapter 1 (this Chapter), we give an overview of techniques in tree drawing and providing the motivation for defining general tree types.
- In Chapter 2, We define types for general trees.
- In Chapter 3, we present the grid planar straight-line grid drawing algorithm (*Quad*) that achieves better distinguishable edge angles than prior published algorithms.
- In Chapter 4, we summarize the main achievements of this thesis and identify future work.

2. CHARACTERIZATION AND CLASSIFICATION OF GENERAL TREES

2.1 Introduction

A binary tree is a tree data structure in which each node has at most two children. A general tree is a tree data structure in which each node can have any number of children. Binary trees have always been the favor or researchers because of their simplicity of their structure. Based on one or more structural properties, binary trees have been classified into many types: Full Binary Tree, Perfect Binary Tree, Complete Binary Tree, Balanced Binary Tree, AVL Binary Tree. One use of these classifications is to perform experimental studies to evaluate binary tree drawing algorithms (See for example, [2–5, 16–18, 32]).

However, general trees are more suitable to represent real-world data in many applications. Even though general trees are more applicable, surprisingly there has been very little study on defining types of general trees. Since, we are developing a general tree drawing algorithm and there are several existing general tree drawing algorithms, it is probable that one algorithm draws general trees better than another algorithm given certain quality measures of a general tree drawing. In the near future, researchers may wish to perform experimental studies on comparing general tree drawing algorithms. So, in this chapter, we classify general trees into several types, based on structural characteristics, just like the following binary tree types.

- *Full Binary Tree*: A full binary tree, or proper binary tree, is a tree in which every node has zero or two children.

- *Perfect Binary Tree*: A perfect binary tree is a full binary tree in which all leaves are at the same depth.

- *Complete Binary Tree*: A complete binary tree is a tree with $n+1$ levels, where for each level $d \leq n-1$, the number of existing nodes at level d is equal to 2^d. This means all possible nodes exist at these levels. An additional requirement for a complete binary tree is that for the nth level, while every node does not have to exist, the nodes that do exist must fill from left to right.

- *Balanced Binary Tree*: A balanced binary tree is where the depth of all the leaves differs by at most 1.

- *AVL Binary Tree*: An AVL binary tree's heights of the two child subtrees of any node differ by at most one.

2.2 Preliminaries

Throughout this chapter, all general trees or binary trees are rooted trees, which means each node has at most one parent and there is only one root in a tree (see Figure 2.1). Many widely referenced terminologies of tree data structures are used, defined as follows.

- *Node*: A node is an abstract basic unit used to build linked data structures such as trees, linked lists, and computer-based representations of graphs. Each node contains some data and possibly links to other nodes. Links between nodes are often implemented by pointers or references.

- *Parent of a node*: Parent node is a node that is the top node of other node(s).

- *Child of a node*: Child node is a node that is the sub node of another node.

- *Sibling of a node*: Siblings are nodes that share the same parent node.

- *Degree of a node*: Degree of a node is the number of edges from that node.

- *Direct edge*: A directed edge refers to the link from the parent node to the child node.

- *Root of a tree*: The root node of a tree is the node with no parent. There is at most one root node in a tree.

- *Leaf node*: A leaf node has no children.

- *Depth of a node*: The depth of a node n is the length of the path from the root to the node. The root node is at depth zero.

- *Level of a tree*: The set of nodes in a tree at a given depth. The root node is at level zero.

- *Size of a tree*: The size of a tree is the number of nodes it contains, including the root.

- *Height of a tree*: The height of a tree is the depth of its furthest leaf. A tree with only a root node has a height of zero. The height of a binary tree is closely related to its size.

All of these definitions are not only applicable to binary trees, but also to general trees. However, they were all created for binary trees, so, to assist in categorizing general trees, in addition to these definitions, we define several more, as follows.

- *Pivot node*: If a node R has an odd number of nodes, $2n - 1$, $n \geq 1$ nodes, R has children (C_1, C_2, ..., C_n, C_{n+1}, ..., C_{2n+1}), the child node at the center of all node C_n is called the pivot node, C_p. If a node R has an even number of node, $2n$, $n \geq 1$ nodes, R has children (C_1, C_2, ..., C_n, C_{n+1}, ..., C_{2n}), then there is no center node, then pivot node C_p is an imaginary node and $C_p = NULL$. If a node R is leaf node, then it does not have the pivot node (see Figure 2.6 and Figure 2.7).

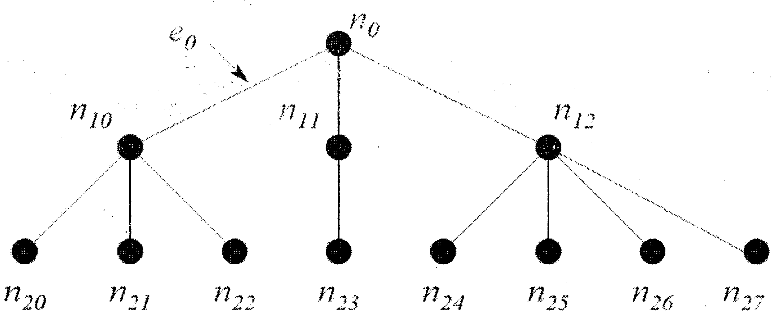


Figure 2.1 A general tree with size = 12. n_0 is the root. n_0 is the parent node of n_{10}, n_{11}, and n_{12}. n_{10}, n_{11}, n_{12} are children nodes of n_0. e_0 is the edge between n_0 and n_{10}. n_{20}'s depth is 2. n_{10}, n_{11}, and n_{12} are level 1 nodes. n_{10} and n_{11} are sibling nodes. n_{20}, n_{21} to n_{27} are leaf nodes.

- *Pivot tree*: For a tree T rooted at R and R is non-leaf node, pivot tree is the subtree rooted at the pivot node. If pivot node is an imaginary node, then pivot tree does not exist. (see Figure 2.2)
- *Left section*: For a tree T rooted at R and R is a non-leaf node, left section is a part of tree T containing all subtrees at the left of the pivot node and the root R. (see Figure 2.3)
- *Right section*: For a tree T rooted at R and R is a non-leaf node, right section is a part of tree T containing all subtrees at the right of the pivot node and the root R. (see Figure 2.4)
- *Load*: Load of tree is the size of the tree. Load of a node n is the size of the tree rooted at node n. Load of a section is the total number of nodes in the section.
- *Width of a tree*: Width of a tree is the number of leaves in a tree.

The purpose of creating the *load* definition is to emphasize the importance of the size (number of children) of a node. In a binary tree, the upper and the lower limit of the size of a node is closely related to, and restricted by, its height, but in a general tree, the height can only influence the lower

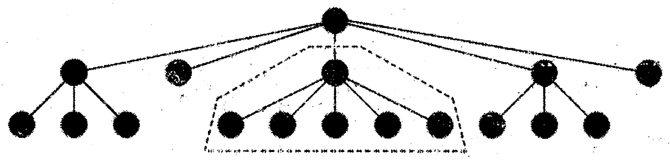


Figure 2.2 Pivot tree of a tree.

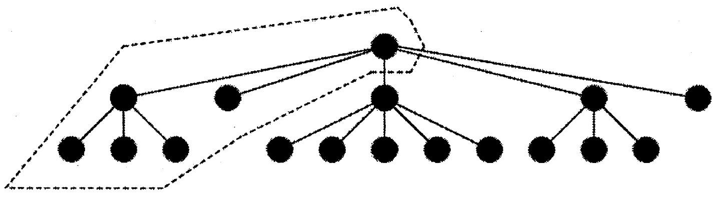


Figure 2.3 Left section of a tree.

limit of the size of a node. The size of a general tree node could have direct impacts on the size of its drawing or the number of operations needed to travel all nodes. Therefore, size should be considered as an important characteristic of a general tree.

The purpose of defining *pivot node* is to provide a ternary feeling for a general tree, left section, pivot section, and right section. Pivot node makes defining general tree types that relate to *balance* easier.

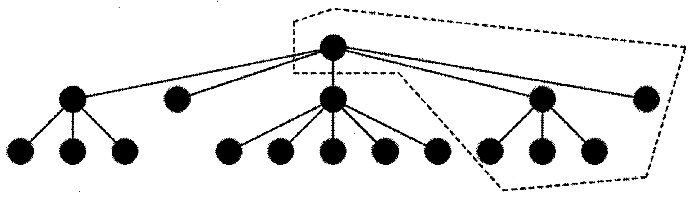
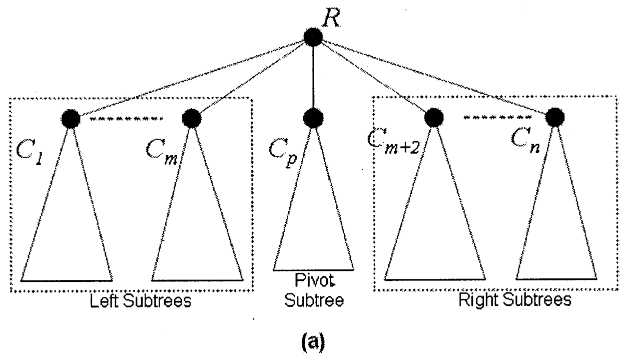


Figure 2.4 Right section of a tree.



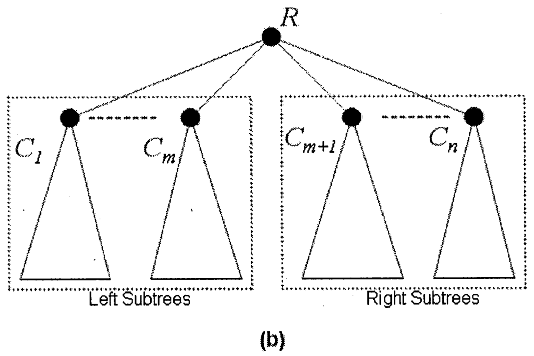


Figure 2.5 (a) Left subtrees, right subtrees, and pivot subtree, with general tree T rooted at R where R has odd number $(n = 2m+1)$ of children. (b) Left subtrees, right subtrees, with general tree T rooted at R where R has even number $(n = 2m)$ of children. There is no pivot subtree in (b).

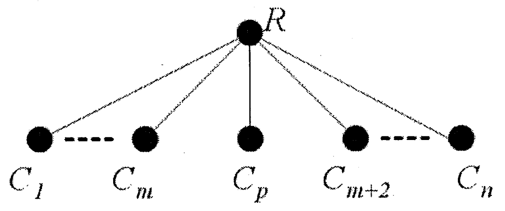


Figure 2.6 A node with odd number $(n = 2m+1)$ of children, C_p is the pivot node of R.

2.3 General tree types

- *Ordered General Tree*: A tree T rooted at R is an Ordered General Tree, if the order of children is important and is not allowed to be changed. (i.e family tree, Figure 2.11)

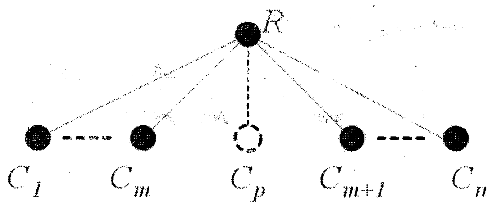


Figure 2.7 A node with even number ($n = 2m$) of children, C_p is an imaginary node and it is the pivot node of R.

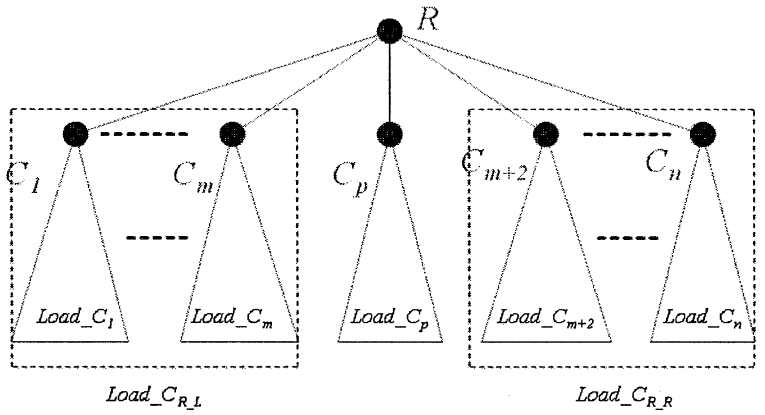


Figure 2.8 Different loads of subtrees rooted at node R. R has odd number (n = 2m+1) of children.

- *Unordered General Tree*: A tree T rooted at R is an Unordered General Tree, if the order of children is not important and can be changed by the purpose of usage (e.g. network node tree). Some unordered general trees can be considered ordered general trees with consideration of the order of children (e.g. networking tree, Figure 2.10). Since all children are not ordered, there is no way to identify the pivot node. Left subtree(s), right subtree(s), and pivot node cannot be defined for unordered general trees. All types that relate to left and right subtrees and pivot node, defined below are not applicable to unordered general trees.

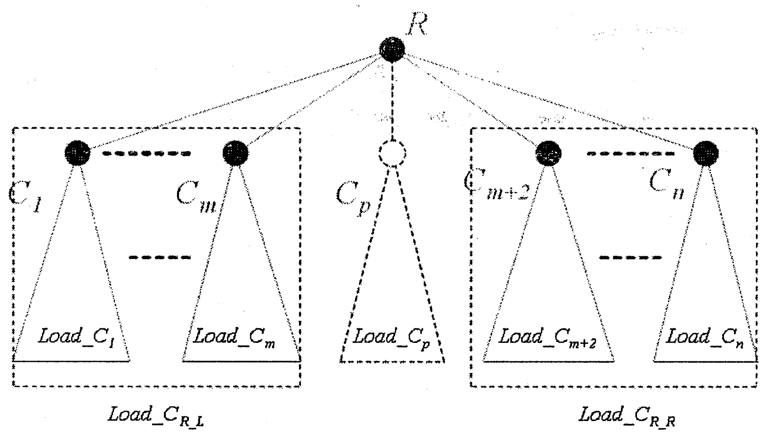


Figure 2.9 Different loads of subtrees rooted at node R. R has even number (n = 2m) of children. Since pivot node is an imaginary node, $Load\_C_p$ is 0.

- *Load Balanced General Tree*: A tree T rooted at R is a Load Balanced General Tree, if the load of T's left section is equal to the load of T's right section and T's pivot tree is a Load Balanced General Tree. T is also a Load Balanced General Tree, if R has no children. (see Figure 2.12)

- *Load Unbalanced General Tree*: A tree T rooted at R is a Load Unbalanced General Tree, if the load of T's left section is *not* equal to the load of T's right section or the subtree at T's pivot is a Load Unbalanced General Tree. (see Figure 2.13)

- *Load Complete Balanced General Tree*: A tree T rooted at R is a Load Complete Balanced General Tree, if all subtrees in T are Load Balanced General Trees. (see Figure 2.14)

- *Height Balanced General Tree*: A tree T rooted at R is a Height Balanced General Tree, if the height of T's left section differs from the height of T's right section by at most 1 and T's pivot tree is a Height Balanced General Tree. T is also a Height Balanced General Tree, if R has no children or is an imaginary node. (see Figure 2.15)

- *Height Unbalanced General Tree*: A tree T rooted at R is a Height Unbalanced General Tree, if the height of T's left section differs from the height of T's right section by more than 1 or the subtree at T's pivot is a Height Unbalanced General Tree. (see Figure 2.16)
- *Height Complete Balanced General Tree*: A tree T rooted at R is a Height Complete Balanced General Tree, if all subtrees in T are Height Balanced General Trees. (see Figure 2.17)
- *Complete Balanced General Tree*: A tree T rooted at R is a Complete Balanced General Tree, if T is a Height Complete Balanced General Tree as well as a Load Complete Balanced General Tree. (see Figure 2.18)
- *Perfect General Tree*: A tree T rooted at R is a Perfect General Tree, if all non-leaf nodes have the same number of children and all leaves are at the same level. (see Figure 2.19)
- *Even General Tree*: A tree T rooted at R is an Even General Tree, if all non-leaf nodes at the same level have the same number of children and all leaves are at same level. (see Figure 2.20)
- *Short General Tree*: A tree T rooted at R is a Short General Tree, if the height of T is less than half T's width. (see Figure 2.21)
- *Tall General Tree*: A tree T rooted at R is a Tall General Tree, if the height of T is greater than twice T's width. (see Figure 2.22)
- *Right Load General Tree*: A tree T rooted at R is a Right Load General Tree, if the load of T's left section plus the load of T's pivot tree is at most of half of the load of T's right section. (see Figure 2.24)

- *Left Load General Tree*: A tree T rooted at R is a Left Load General Tree, if the load of T's right section plus the load of T's pivot tree is less than half of the load of T's left section. (see Figure 2.23)

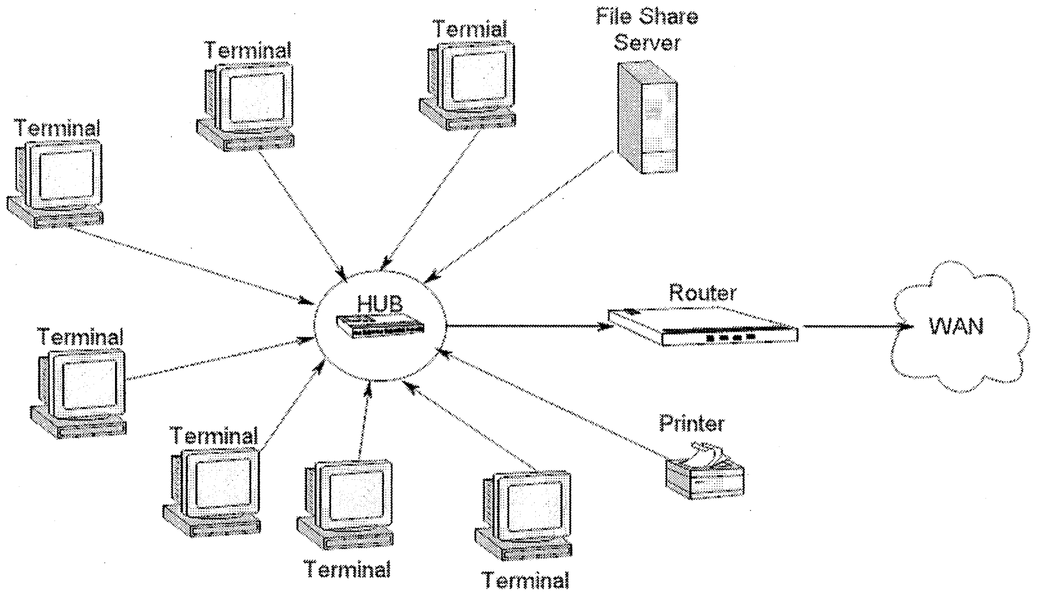


Figure 2.10 A general tree shows the content of a network. The order of each terminal (leaf) is not important for such a drawing.

Each of these types describes one or more structural characteristics of a general tree. Some general trees may qualify as one or more types. An ordered general tree could qualify for more types than an unordered general tree because of its ability to have left section, right section and a pivot tree. Sometimes, an ordered general tree may be viewed as an unordered general tree by user. For example a tree of clients in a network (see Figure 2.10) can also be viewed as an ordered general tree when clients are sorted by their IP address (see Figure 2.25). Drawing algorithms also treat trees differently. For example, general tree drawing algorithms H-V, Separation, and *Quad*

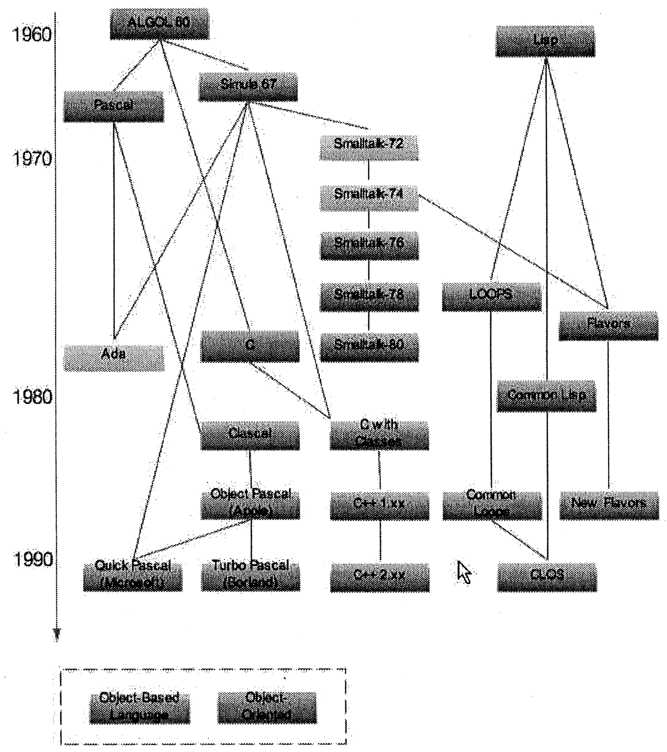


Figure 2.11 A partial programming language family tree-like graph. All nodes are ordered by time. *[Online image, available at http://www.edrawsoft.com]*

(described in next chapter) consider trees as unordered, because they all rearrange children to meet algorithm purposes. Level treats trees as ordered and keeps the order of children.

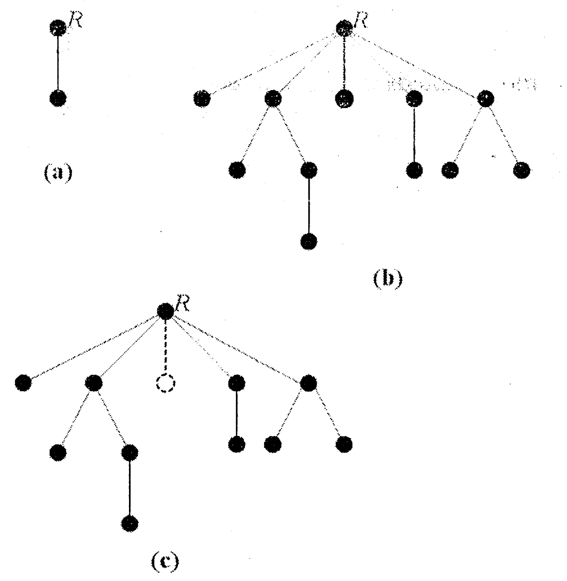


Figure 2.12 (a), (b), and (c) are three load balanced trees.

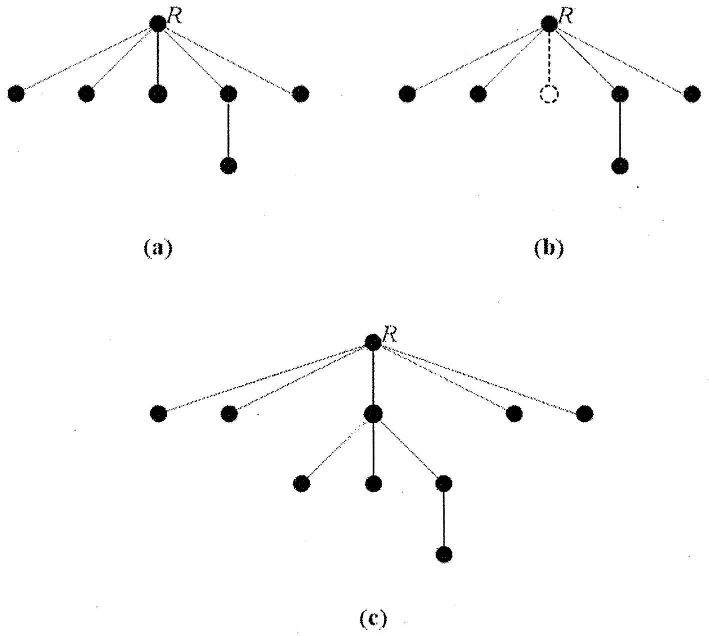


Figure 2.13 (a), (b), and (c) are three load unbalanced trees.

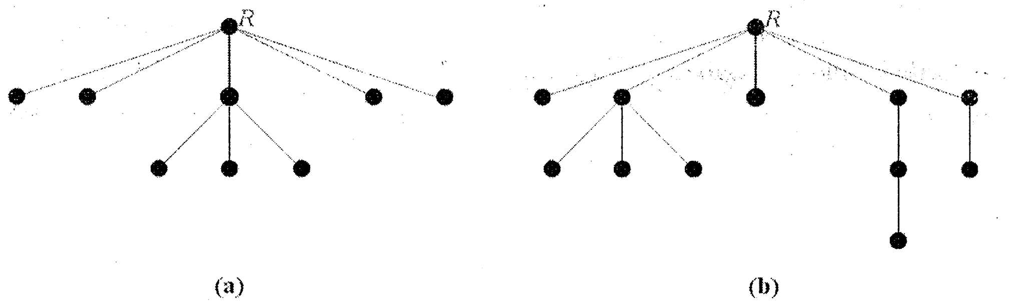


Figure 2.14 (a) and (b) are both load complete balanced trees.

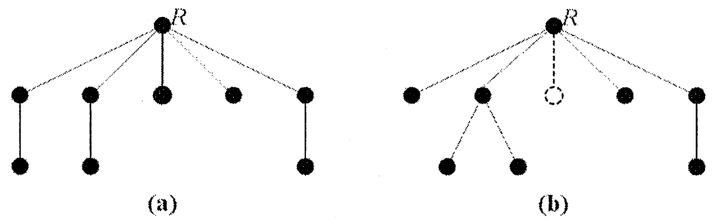


Figure 2.15 (a) and (b) are both height balanced trees.

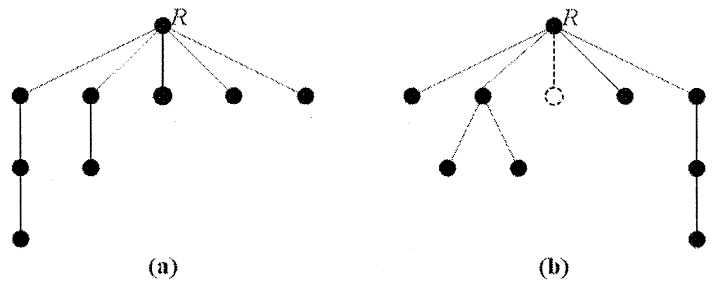


Figure 2.16 (a) and (b) are both height unbalanced trees.

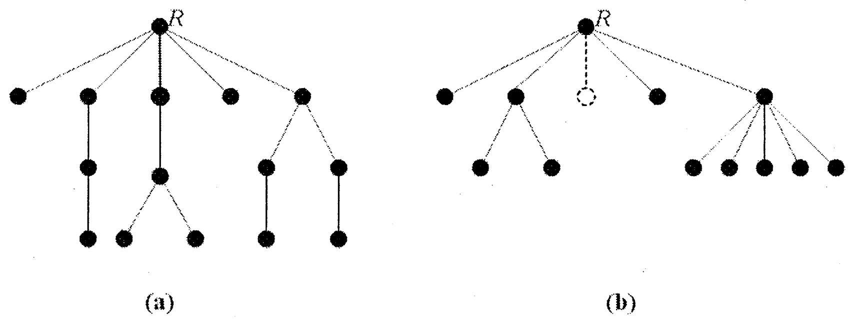


Figure 2.17 (a) and (b) are both height complete balanced trees.

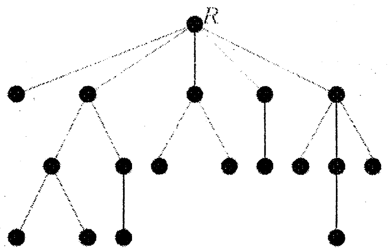


Figure 2.18 A drawing of a complete balanced general tree.

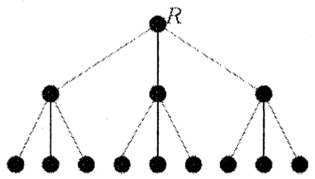


Figure 2.19 A drawing of a perfect general tree with all non-leaf nodes having 3 children.

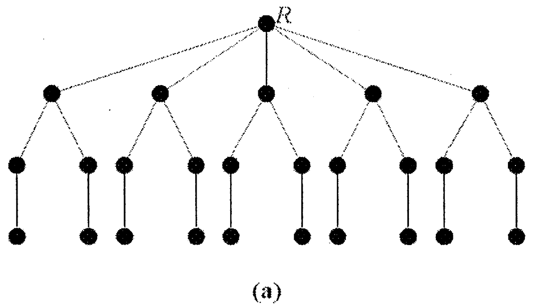


Figure 2.20 A drawing of an even general tree where each non-leaf node of level 1 has 2 children and all non-leaf nodes of level 2 have 1 child.

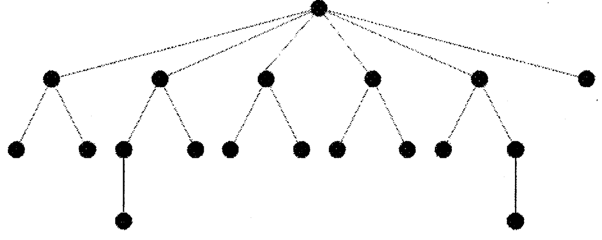


Figure 2.21 A drawing of a short general tree with height = 3 and width = 11.

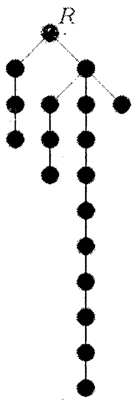


Figure 2.22 A drawing of a tall general tree with height = 10 and width = 4.

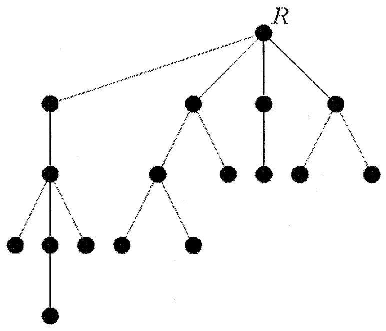


Figure 2.23 A drawing of a left load general tree.

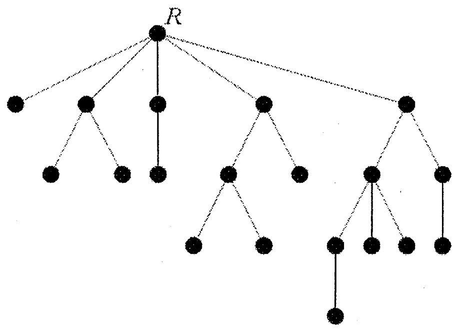


Figure 2.24 A drawing of a right load general tree.

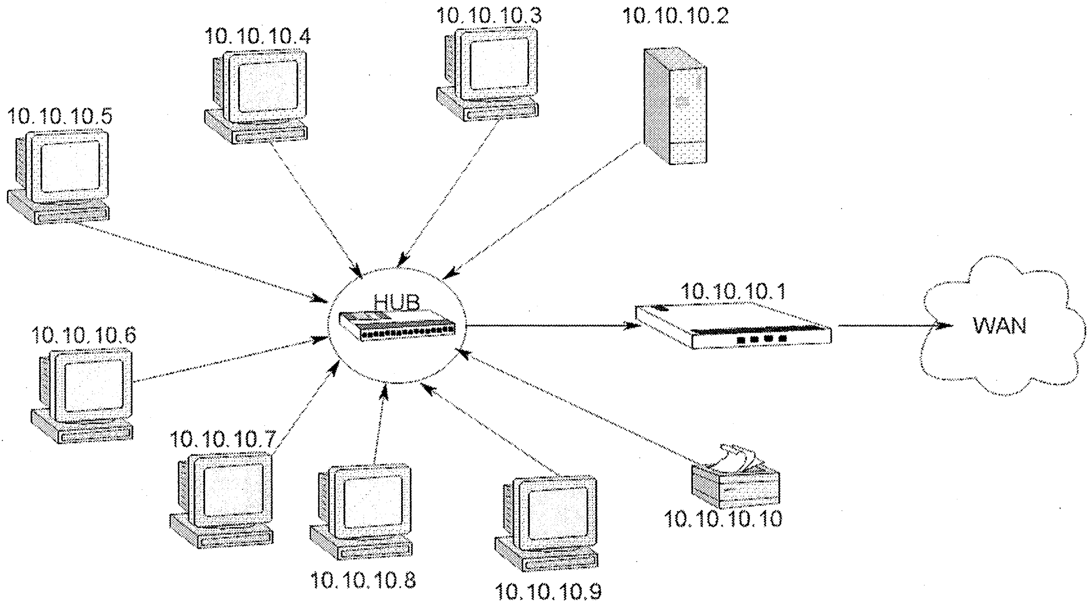


Figure 2.25 The same network in Figure 2.10, but now it is an ordered network tree where clients are ordered by IP address.

3. *QUAD* AN ALGORITHM FOR PLANAR STRAIGHT-LINE GRID DRAWINGS OF GENERAL TREES WITH USER SPECIFIED ANGULAR COEFFICIENT

3.1 Introduction

Visualizing a tree can enhance a user's ability in understanding its structure. Hence, much research has been devoted to tree visualization, which has produced a plethora of tree-drawing algorithms [6–8,10,12–15,19,22,23,27,29–31]. Because of their simpler structure, most of the research has been devoted to drawing binary trees [1]. However, general trees appear more commonly in practice. Currently, there are three main planar straight-line grid drawing algorithms available for general trees, Level [22], H-V [9] and Separation [24].

3.2 Background

The Level-based algorithm was developed for binary trees but can be extended for general trees, produces drawings with most trivial understanding of trees by people (see Figure 3.1). The problem of the drawings produced by Level algorithm is that they have poor area and poor aspect-ratio. In addition to poor area and poor aspect-ratio, Level generates many small angles among outer edges.

The general trees extension of the H-V algorithm generates drawings with much better area than Level ($O(n\ log\ n)$ versus $O(n^2)$), but it still produces drawings with poor aspect ratios and many small edge angles.

A more recent drawing algorithm, Separation [24], produces drawings with user-controlled aspect ratio and optimal area even for high degree trees. However, Separation may also generate drawings with many small angles (see Figure 3.2).

One of the reasons Level, H-V, and Separation generate so many small angles is that they only use at most two quads of the Cartesian plane, which limits the growing space of the drawing. In addition to limited drawing space, they all treat leaves and subtrees in the same way, placing leaf child nodes first and then placing subtree child nodes in a line. The problem of allocating children, leaves and subtrees, in a line is that it assigns the largest angle possible between the first two allocated nodes even if they are both leaves. The angles will then quickly get smaller and smaller as more and more leaves or subtrees are added.

We have developed *Quad* [25], a general tree drawing algorithm which produces plannar straight-line grid drawings. It mainly is intended to provide a drawing with good distinguishable angles when a node has high degree (see Figure 3.3). We also try to have a good aspect ratio and small area. Our algorithm is able to draw nodes in more than one quad (see Figure 3.4) and uses different procedures to place leaf nodes and subtree nodes. It first tries to place leaves in one quad, and then, if one quad is not enough to contain all leaves with angle above the specified angular coefficient, it will move into quad two and then quad four of a Cartesian plane (see Figure 3.5) as necessary. Our experiments show Quad produces drawings with better angles than all known existing general tree drawing algorithms on high degree trees.

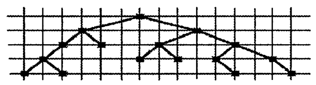


Figure 3.1 Drawing of a binary tree using the level algorithm with grid showing.

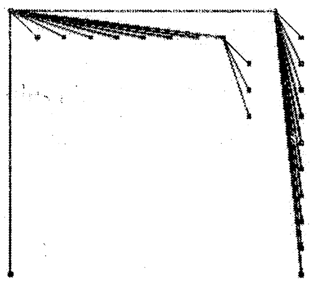


Figure 3.2 Drawing of a general tree using the Separation algorithm. This general tree has 8 leaves and 2 subtrees. The edge angle between the last leaf and the first tree is very small. It is very hard to visually distinguish the edges. The same situation arises with the last few leaves of second subtree of the root.

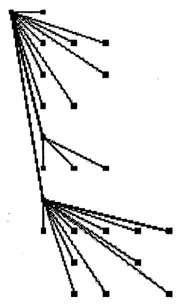


Figure 3.3 Drawing of a general tree using our Quad algorithm with angular coefficient of 1. This is the same general tree in Figure 3.2. All edges are much more distinguishable.

3.3 Preliminaries

In the remainder of this thesis, we use the term *drawing* to mean a planar straight-line grid drawing. We will assume that the plane is covered by an infinite rectangular grid. A *horizontal*

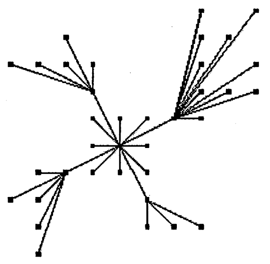


Figure 3.4 Drawing of a general tree using Quad. The drawing spreads to 4 quads.

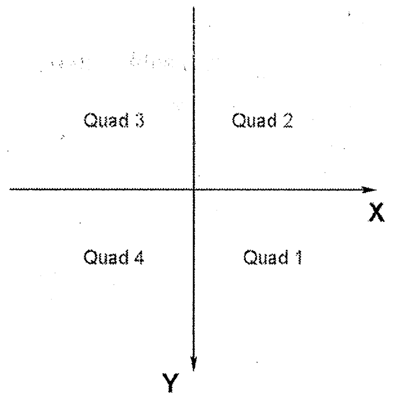


Figure 3.5 A Cartesian plane with numbering of each quad.

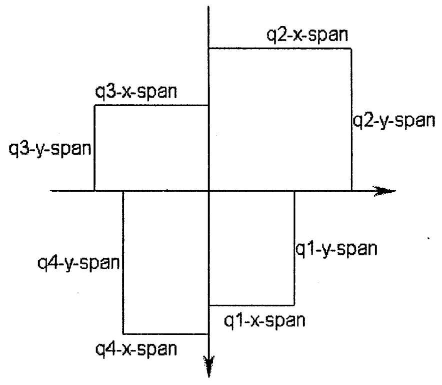


Figure 3.6 Illustration of all single quad-based directional spans.

channel (*vertical channel*) is an infinite line parallel to X- (Y-) axis, passing through the grid-points.

Let T be a degree-d tree and let Γ be a drawing of T.

Below is a list of terms with their definitions as they will be used in this paper.

- *Location of a tree drawing*: The X-Y location of the root of the tree.
- *Allocate a tree drawing*: The process of finding a good location for the tree in the X-Y plane.

- *Level*: The number of edges between a given node and the root. The Root by definition is at level 0.

- *Span*: The height and width of the tree drawing Γ. If the drawing Γ has height h and width w, then its span is denoted as (w, h). w and h both can be zero or positive integers.

- *Directional Span*: The directional span is used to describe the height and width of a drawing within a quad. The difference between Span and Directional Span is that the Directional Span's height or width are signed (+/−). The sign is relative to the subtree root node.

  - *q*n*-x-span/q*n*-y-span* (see Figure 3.6): This represents the directional span of quad n, $n \in \{1, 2, 3, 4\}$. If the sub-drawing of Γ contained in quad n has the height h and width w, then its *q*n*-x-span* is equal to w when $n \in \{1,2\}$, or $-w$ when $n \in \{3,4\}$; and *q*n*-y-span* is equal to h when $n \in \{1,4\}$, or $-h$ when $n \in \{2,3\}$.

  - *q12-x-span* (see Figure 3.7): This represents the X directional span of a tree drawing in quad-1 and quad-2. If the sub-drawing of Γ contained in quad 1 and quad 2 has the

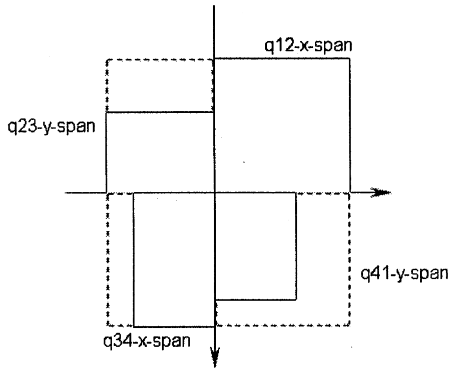


Figure 3.7 Illustration of all bi-quad based directional spans.

width w, then its *q12-x-span* is equal to w.

- *q23-y-span* (see Figure 3.7): This represents the Y directional span of a tree drawing in quad-2 and quad-3. If the sub-drawing of Γ contained in quad 2 and quad 3 has the height h, then its *q23-y-span* is equal to $-h$.
- *q34-x-span* (see Figure 3.7): This represents the X directional span of a tree drawing in quad-3 and quad-4. If the sub-drawing of Γ contained in quad 3 and quad 4 has the width w, then its *q34-x-span* is equal to $-w$.
- *q41-y-span* (see Figure 3.7): This represents the Y directional span of a tree drawing in quad-4 and quad-1. If the sub-drawing of Γ contained in quad 4 and quad 1 has the height h, then its *q41-y-span* is equal to h.

• *Angular Coefficient*: A user-specified coefficient used in determining the edge angle of a drawing. The coefficient is specified between 0 and 45 with a larger number increasing the majority of the angles between edges and thus the area of the drawing. *Quad* attempts to place the nodes at the locations that all edge angles are above the angular coefficient. For a subtree with a relatively small number of nodes, the angular coefficient will be the angular resolution of that particular subtree.

• *Expected Shape of the drawing*: *Quad* can produce a drawing of the tree spread to all four quads. The user can specify the desired shape of drawing they. The four choices are quad 1, quad 1-2, quad 1-2-3, and quad 1-2-3-4.

• *Direction of growth*: *Quad* puts the drawing of a nodes' leaves and subtrees along the X-axis or the Y-axis. The drawing along X-axis is referred to as X-direction-growth while along Y-axis is referred to as Y-direction-growth. Directional of growth is explained in detail in section 3.4

- *Open Channel*: The axis orthogonal to the direction of growth.

3.4 Direction of Growth

We introduce direction of growth as a way to have control over the overall aspect ratio of the drawing. Our algorithm mainly focuses on providing more distinguishable angles between edges, but, at the same time, we try to achieve a good aspect ratio.

For every tree node and the subtrees rooted at its children, if the algorithm wee to continue appending the subtrees' drawings along the positive Y-axis direction, then the output drawing will continue getting longer in the positive Y-axis direction, which we call Y-direction-growth (see Figure 3.8). This would lead to a drawing with a poor aspect ratio. *Quad* addresses this problem by alternating the direction of growth. If one node's subtrees are allocated in the positive Y-axis direction (Y-direction-growth), then its subtrees' subtrees will be allocated in the positive X-axis direction (X-direction-grow). In general, for all even level subtrees, Y-direction-growth is applied,

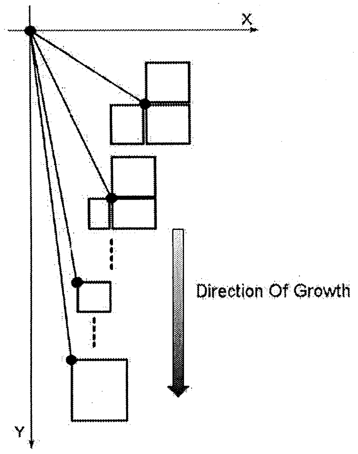


Figure 3.8 This is a quad-1 mark-up drawing showing the Y-direction-growth.

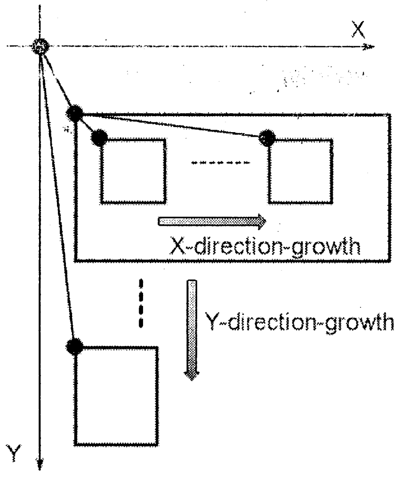


Figure 3.9 A quad-1 drawing which shows the direction of growth at each level is alternating from the previous level between Y-direction-growth and X-direction-growth.

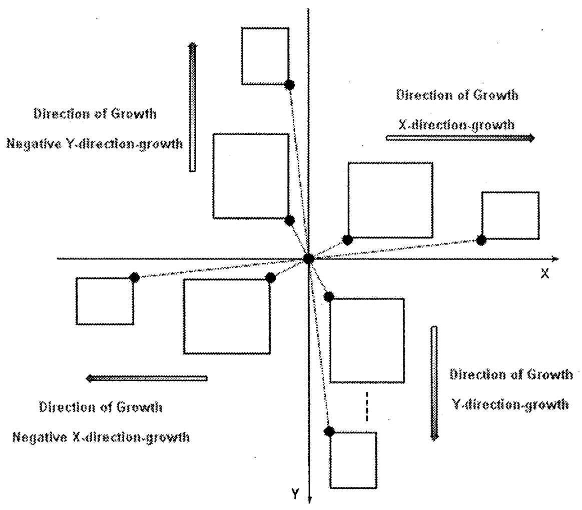


Figure 3.10 Each quad has different direction of growth from others, which keeps the overall drawing square. Quad 2, 3, 4's drawings are the result of rotating a quad 1's drawing counterclockwise by 90°, 180°, and 270° respectively.

and for all odd level subtrees, X-direction-growth is used (see Figure 3.9).

For simplicity in implementing Quad we use only positive X-direction-growth and positive Y-direction-growth, which are only within quad 1. If the drawing is intended to be drawn in quad 2, then it is "drawn' in quad 1 and then it is simply rotated 90° counterclockwise to relocate into the correct quad. What this means to direction of growth is that a node has positive Y-direction-growth in quad 1, then after rotating 90° counterclockwise it will become positive X-direction-growth in quad 2, or rotated 90° clockwise it will become negative X-direction-growth in quad 4, or rotated 180° clockwise it will become negative Y-direction-growth in quad 3. The intention of changing direction of growth in each quad is that the resulting drawing will have an good aspect ratio (see Figure 3.10).

3.5 General Algorithm Outline

Our algorithm uses the divide and conquer paradigm. It recursively splits the tree and applies the following sequence of steps at every subtree. For a tree with the root *R*.

- *Step 1*: Draw/allocate leaves. If *R* has leaf children then each leaf node is assigned to a location relative to the location of the *R*.

- *Step 2*: Draw subtrees. If *R* has subtrees, then recursively obtain drawings of all subtrees rooted at children of the *R*. After this step is completed, each subtree is represented by its root and a 1, 2, or 3 quad shaped drawing. This step will not be performed when the parent node does not have any subtrees.

- *Step 3*: Allocate subtrees. If *R* has subtrees, then each subtree's root will be assigned to a location relative to the location of the *R*.

During these three steps, all locations assigned to nodes are relative locations which assume parent node is at location $(0,0)$. When all nodes in a tree complete the three steps, a top-down flow

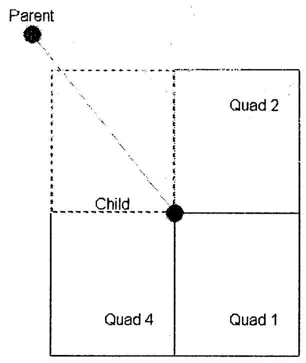


Figure 3.11 Quads 1, 2 and 4 are used for drawing of a subtree within quad 1 of the parent node. Quad 3 is used for the edge to the parent.

of process assigns a final location to each node in the tree which is relative to the root of the tree.

The algorithm also takes two parameters: the angular coefficient and the quads to be used at root level. The angular coefficient will be used throughout all steps to maintain a general uniformity of angles between edges within a single subtree. A larger angular coefficient will result in a larger angle between edges, in turn increasing the area of the drawing. For a subtree with a small number of nodes, relative to the specified angular coefficient, *Quad* is able to achieve a resulting angular resolution of the leaf nodes will be equal to or larger than the specified angular coefficient. At the root level, the drawing can be spread over one, two, three, or all quads as specified by the quads parameter given by the user.

The drawing of a subtree is designed to spread to a maximum of three quads: quads 1, 2, and 4. The reason we intentionally leave quad 3 empty, is that the subtree's parent will be located in quad 3, relative to the subtree's root as origin, and we need to have a way to connect this subtree's root to its parent without generating any crossings or overlappings (see Figure 3.11). We first draw all subtrees in quad 1, and then we rotate the drawing 90° clockwise or counterclockwise to produce a quad 2 or quad 4 drawing (see Figure 3.12).

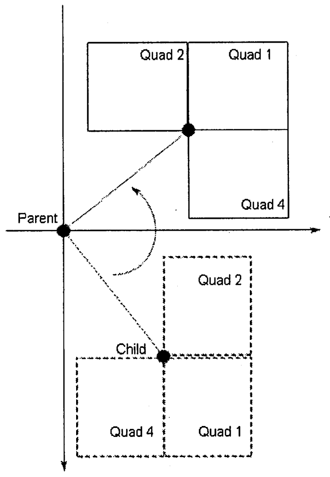


Figure 3.12 An illustration of a subtree drawing being allocated into quad 2 of the parent node. The child is first allocated in quad 1 and then the drawing is rotated 90° counterclockwise around the parent node into quad 2.

3.5.1 Leaf allocation

Leaf allocation is the first step of our algorithm. We try to assign a grid location for each leaf such that the area occupied by them is as small as possible and no two locations are collinear with the origin. Algorithms H-V and Separation align all leaves along the horizontal or vertical channels $X = 1$ or $Y = 1$, which makes the angular resolution decrease very quickly as more children are

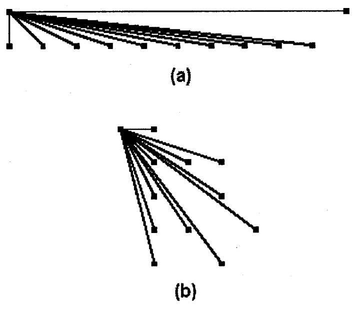


Figure 3.13 (a) Separation drawing of a leaf-only node with 11 leaves. The angular resolution gets very small: only approximately 0.5°. (b) Quad drawing of same tree. The angular resolution is approximately 3°.

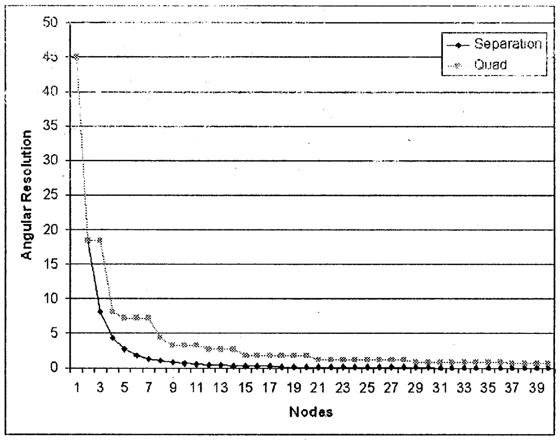


Figure 3.14 Graph shows the angular resolution drop of Separation and Quad, drawn in one quad, with increasing number of leaf nodes. The angular resolution of Separation is smaller than Quad, except for the first two nodes.

drawn (see Figure 3.13 (a)). We developed an algorithm called Valid Leaf Location (VLL), which finds valid grid locations for leaves within one quad. The angular resolution of a drawing from a leaf-only tree produced using VLL decreases at a slower pace than Separation, as more children are drawn (see Figure 3.13 (b) and Figure 3.14). Our tree drawing algorithm uses VLL with the angular coefficient to determine where to place leaves. *Quad* attempts to fit all leaves into quad 1 with an angle as close as possible to the angular coefficient. If the desired angular coefficient is too large for a one-quad drawing, leaves' placement will expand to other quads, again being placed with edge angles as close as possible to the angular coefficient. This results in very uniform angles between leaf node edges within a subtree.

Location Generator with VLL

We want to have a set of all grid locations $(x_0,y_0),(x_1,y_1),...,(x_n,y_n)$ connected with straight lines to the origin $(0,0)$, such that no two locations are collinear with the origin. For example, if we pick location $(0,1)$ as a valid location, then the location $(0,2)$ will not be a valid location, because

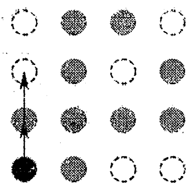


Figure 3.15 All solid dots are valid leaf locations, others are invalid leaf locations. Edges to a valid leaf location and a invalid leaf location will result in overlapping.

the lines between $(0,0)$ to $(0,1)$ and $(0,0)$ to $(0,2)$ will overlap each other (see Figure 3.15).

If two grid locations (a,b) and (c,d), $a < c$ and $b < d$, are collinear with origin $(0,0)$, then c is the result of a multiplied with some integer constant n $(n > 1)$ and d is the result of b multiplied with the same integer constant n (see Figure 3.16). So in order to have a grid location (p,q), which is not collinear with any other location (x,y), $x < p$, $y < q$ and origin $(0,0)$, p and q must not have a common integer divisor n, $n > 1$. This leads to the following lemma.

Lemma 3.1 *Grid location (x,y) is a valid leaf location, if and only if x and y are relatively prime.*

The algorithm Valid Leaf Location (VLL) uses the above lemma to generate valid grid locations (see Figure 3.17).

VLLGenerator is the procedure we use to produce a list of valid locations, using the VLL algorithm. The order in which these locations are generated is based on counterclockwise rounds, with each round being one grid unit further from the origin than the previous round (see Figure 3.18).

Angle check for leaves

The leaf location obtained using VLL is a valid potential location, but it might not be a location that meets the angular coefficient requirement provided by the user. *Quad* performs an angle check after each leaf node gets a potential location using VLL. *Quad* assumes there is an edge from the

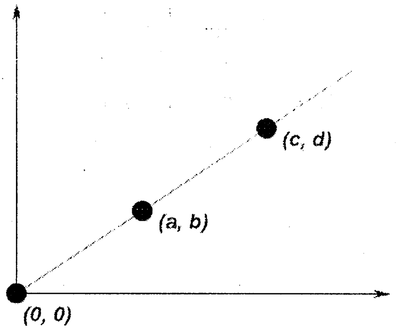


Figure 3.16 If grid locations $(0,0), (a,b)$ and (c,d) are collinear then there is an integer $n, n > 1$, such that $c = n * a, d = n * b$.

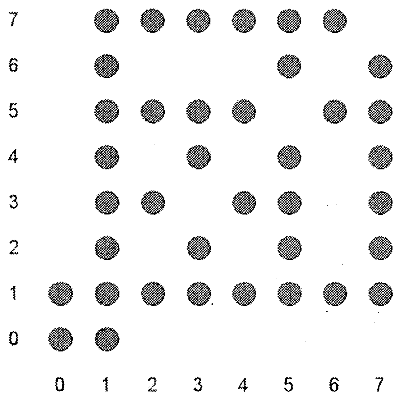


Figure 3.17 A map of all valid leaf locations within the square determined by points $(0,0)$ and $(7,7)$.

root to this potential location and checks the angle of this edge against the edges of all previously allocated leaves. If any angle is smaller than the angular coefficient, the potential location for this leaf might not be used if there is a neighboring quad available to expand to.

Spread leaves to other quads

After a potential location for a leaf is considered to violate the angular coefficient, the algorithm will try to place the leaf into another quad (see Figure 3.19). For example, when a node first tries

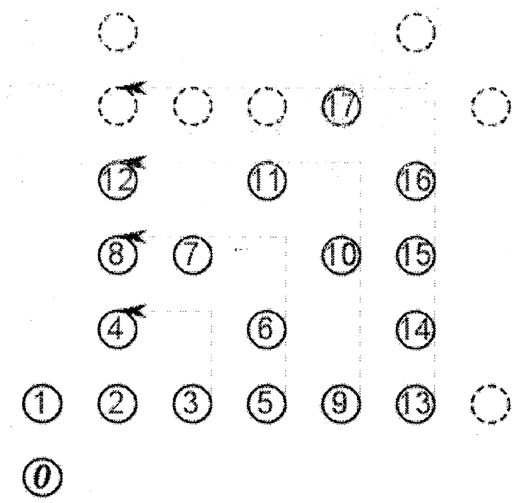


Figure 3.18 The order in which locations are provided by the VLL algorithm. Nodes 1 through 17 are children of node 0, labeled by the order in which they are generated. The arrows show that the VLL algorithm generates locations in a counterclockwise direction, with distance from the origin increasing with each pass.

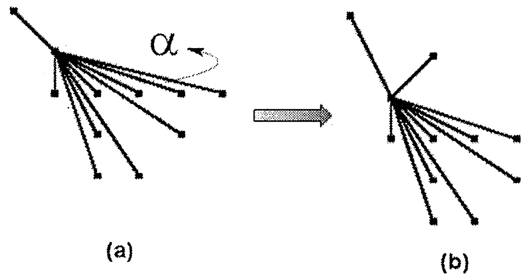


Figure 3.19 A drawing with angular coefficient of 5. In (b) the 9th leaf is allocated in quad 2, because if it is allocated in quad 1 in (a), the angle α between the 9th leaf with another leaf is approximately 4 degrees which is smaller than target angular coefficient 5.

to place all leaves into quad 1 and its $(n+1)th$ leaf gets a location which would create an angle smaller than the angular coefficient with one of the existing edges, our algorithm then will try to place the rest of the leaves into quad 2. If after another m leaves, there are no more valid locations with satisfiable angle in quad 2, then quad 4 will be used to place leaves. If the root has so many

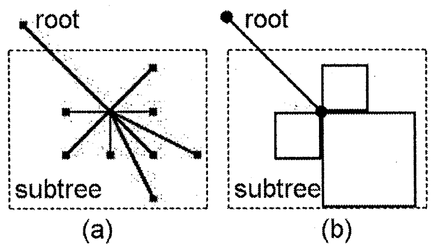


Figure 3.20 A three-quad drawing of a node with one leaf-only subtree. The *q1-x-span* $= 2$, *q1-y-span* $= 2$, *q2-x-span* $= 1$, *q2-y-span* $= -1$, *q4-x-span* $= -1$, *q4-y-span* $= 1$, *q12-x-span* $= 2$, *q23-y-span* $= -1$, and *q41-y-span* $= 2$. (a) A drawing of a tree which uses all three quads; (b) A notation representing the drawing in each quad with the rectangle corresponding to its size.

leaves that quad 4 still cannot contain the rest of the leaves without violating the angular coefficient, then the effort of inserting leaves with an angle equal to or greater than the angular coefficient will be stopped, and all remaining leaves will be evenly placed into quads 1, 2, and 4 in sequence. After all leaves are allocated, the algorithm will compute all directional span values based on the current stage of the drawing (see Figure 3.20) .

3.5.2 Subtree allocation

After assigning locations to the leaves, we can allocate the subtrees, by assigning a relative location to the root of each subtree. We define two cases for allocating subtrees. The two cases are defined below. Because subtree allocation does not initially take leaf nodes into consideration, after subtree allocation, there may be small angles or overlappings between leaves and subtrees. These situations are explained and handled in section 3.5.3.

- Case 1: Only one subtree is to be allocated into one Quad. If the current tree has only one subtree needing allocation, determining the position of its root is trivial, dependent only upon the shape and spans of the subtree. In the following, consider *S* to be the subtree needing allocation, and consider R to be the parent of S in the state it is prior to S being allocated. Since only one subtree needs allocation, R may only contains leaves. Each type of subtree is allocated as following:

- One-Quad Subtree (see Figure 3.21): $X = 1, Y = \textit{q1-y-span}(\text{drawing of leaves}) + (\text{offset between subtree drawings}, 1)$.
- Two-Quad Subtree (see Figure 3.22): $X = 1, Y = \textit{q1-y-span}(\text{drawing of leaves}) + (-\textit{q2-y-span})(\text{drawing of the subtree rooted at S}) + (\text{offset between subtree drawings}, 1)$.
- Three-Quad Subtree (see Figure 3.23): $X = \textit{q4-x-span}(\text{drawing of the subtree rooted at S}) + 1, Y = \textit{q1-y-span}(\text{drawing of leaves}) + (-\textit{q2-y-span})(\text{drawing of the subtree rooted at S}) + (\text{offset between subtree drawings}, 1)$.

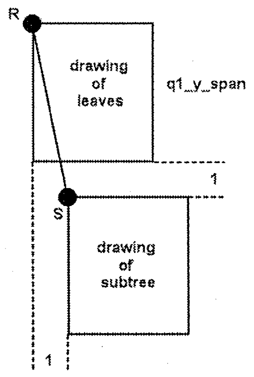


Figure 3.21 Representation of allocation of a one-quad subtree. The subtree is allocated one unit offset from its parent's leaves drawing, and one unit offset from the Y-axis.

- Case 2: Multiple subtrees to be allocated into one Quad. If the current tree has multiple subtrees that need to be allocated, the location of a subtree's root is determined by its shape and

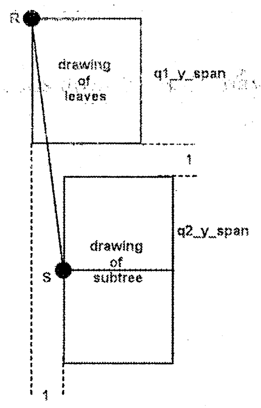


Figure 3.22 Representation of allocation of a two-quad subtree. The subtree is allocated one unit offset from its parent's leaves drawing plus the subtree's $-q2$-y-$span$, and one unit offset from the Y-axis.

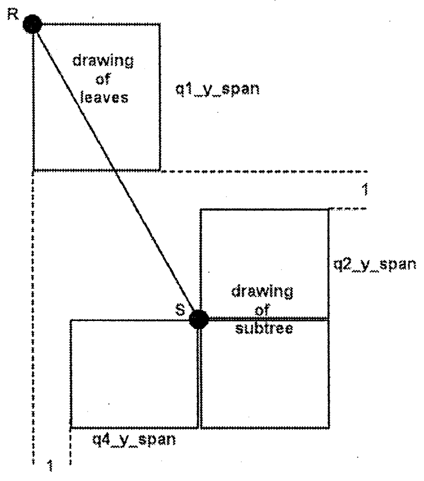


Figure 3.23 Representation of allocation of a three-quad subtree. The subtree is allocated one unit offset from its parent's leaves drawing plus the subtree's $-q2$-y-$span$, and one unit offset from the Y-axis plus the subtree's $-q4$-x-$span$.

spans, as well as the shapes and spans of all previously allocated subtrees. In the following, consider *S* to be the subtree needing allocation, consider *R* to be the parent of *S* in the state it is in prior to *S* being allocated. Since multiple subtrees need to be allocated, *R* may or may not contain subtrees. If there is any subtree, each type of subtree is allocated as following:

- One-Quad Subtree (see Figure 3.24): $X = 1$, $Y = \textit{q1-y-span}$(drawing of leaves and previous subtrees) + (offset between subtree drawings,1).
- Two-Quad Subtree (see Figure 3.25): $X = 1$, $Y = \textit{q1-y-span}$(drawing of leaves and previous subtrees) + $(-\textit{q2-y-span})$(drawing of the subtree rooted at S) + 1.
- Three-Quad Subtree (see Figure 3.26): $Y = \textit{q1-y-span}$(drawing of leaves and previous subtrees) + $(-\textit{q2-y-span})$(drawing of the subtree rooted at S) + (offset between subtree drawings,1). For a tree containing subtrees $\{t_1, t_2, ..., t_n\}$, X is a recursive function, defined as:
  * Recursive Case: $X(t_m) = \textit{q4-x-span}(t_m) + X(t_{m+1})$, where $m < n$
  * Base Case: $X(t_n) = \textit{q4-x-span}(t_n)$ + (offset between subtree drawings,1)

In this procedure, subtree drawings are closely allocated right next to each other. It requires to allocate three-quad subtrees, to avoid crossings (see Figure 3.27).

Spread subtrees to other quads

Similar to leaf allocation, our algorithm first tries to fit all subtrees into one quad. Because of the way we allocate the subtrees, the edge to the furthest two subtrees from the parent form the smallest angle. We can find this angle by computing the locations of the last two subtrees' roots (see Figure 3.28). This angle is then compared to the angular coefficient. If the angle is too small, then the algorithm will evenly distribute the subtrees among two quads. The angle will then be checked, again, for both quads, and if this angle is still too small, the subtrees will be evenly distributed among three quads (see Figure 3.29). This angle can be computed with the following formulas:

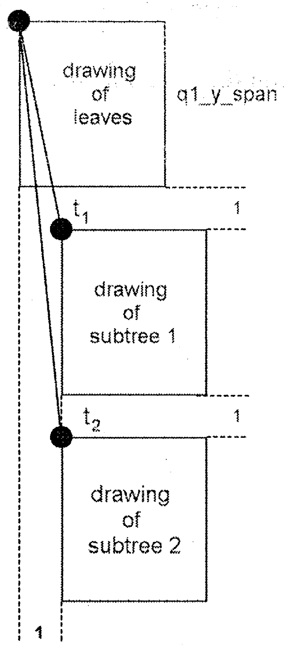


Figure 3.24 For a node containing one or more one-quad subtree, each one-quad subtree drawing is allocated to avoid crossings and overlappings with previous subtree drawings.

Let t_1, t_2, ..., t_n be the subtrees whose roots are the children of the tree under consideration. t_1, t_2, ..., t_n will be allocated in this order within one quad. The minimum angle can be computed by finding the locations of the root of t_n and t_{n-1}.

- The location (X_n, Y_n) of the root of t_n: $X_n = -$(q4-x-span) of $t_n + 1$, $Y_n = \sum_{i=1}^{n-1} -$(q2-y-span) $+$ (q41-y-span) of t_i.
- The location (X_{n-1}, Y_{n-1}) of the root of t_{n-1}: $X_{n-1} = -$(q4-x-span) of $t_n + -$(q4-x-span) of $t_{n-1} + 1$, $Y_{n-1} = \sum_{i=1}^{n-2} -$(q2-y-span) $+$ (q41-y-span) of t_i.

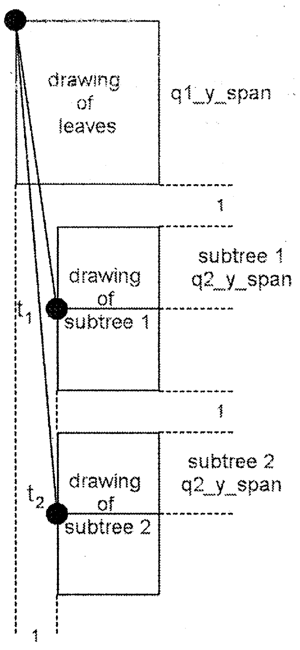


Figure 3.25 For a node containing one or more two-quad subtree, each two-quad subtree drawing is allocated to avoid crossings and overlappings with previous subtree drawings.

3.5.3 Small angle and overlap conflicts between leaves and subtrees

Because leaf nodes are allocated without consideration of subtree nodes, and vice versa, the algorithm may generate small angles, and even overlappings, between leaf edges and subtree edges. This is handled, within a quad, by reallocating any leaf node that conflicts with a subtree (see Figure 3.30).

There are two situations in which a conflict can occur. The first is when a node's leaf edge and a subtree edge overlap. The second situation is when a node's leaf edge and a subtree edge form an angle smaller than the angular coefficient. Both of these types of conflicts are handled as follows.

- If there is quad which has no leafs or subtrees allocated, the leaf node is reallocated into that open quad.

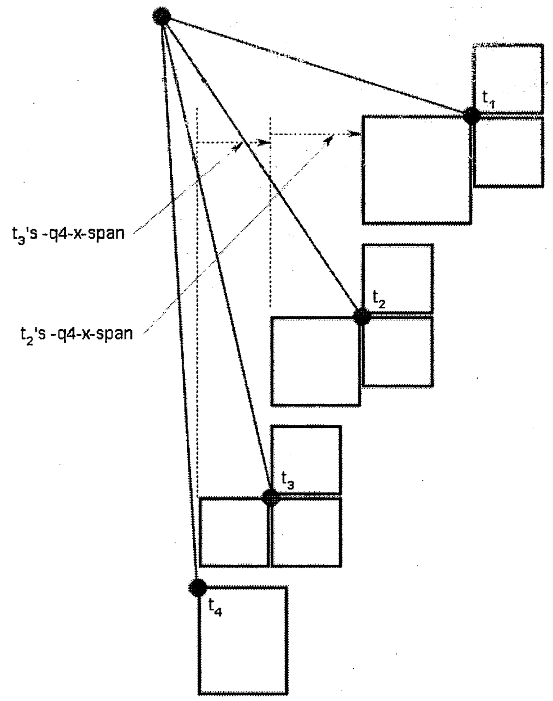


Figure 3.26 For a node containing more than one three-quad subtree, each three-quad drawing needs to be shifted to avoid crossings or overlappings with later three-quad subtrees.

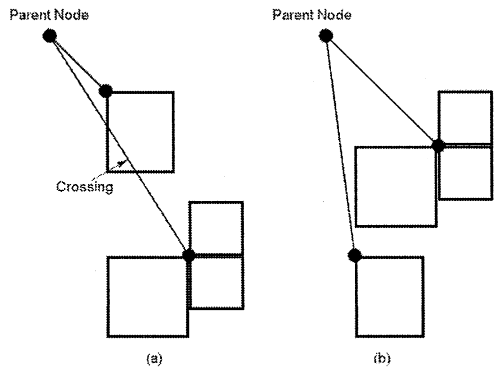


Figure 3.27 (a) A three-quad subtree drawing, where reallocation is needed for previously allocated subtrees to avoid crossings or overlappings. (b) If the three-quad tree is drawn first then no reallocation is needed.

- If there is no empty quad, then the leaf node is reallocated into the open channel, within the same quad. That is, if the subtrees were placed in the X direction, the conflicting leaves will be reallocated in the Y direction where $x = 1$.

By design, *Quad* ignores the small angle conflict when all three quads are already in use.

3.6 Experimental Study

For an experimental study, we implemented the algorithm in C++. The algorithm was evaluated on randomly generated general trees with high degree at the root, consisting of up to twenty thousand nodes. The experiment evaluates the area as related to the number of nodes and the angular coefficient.

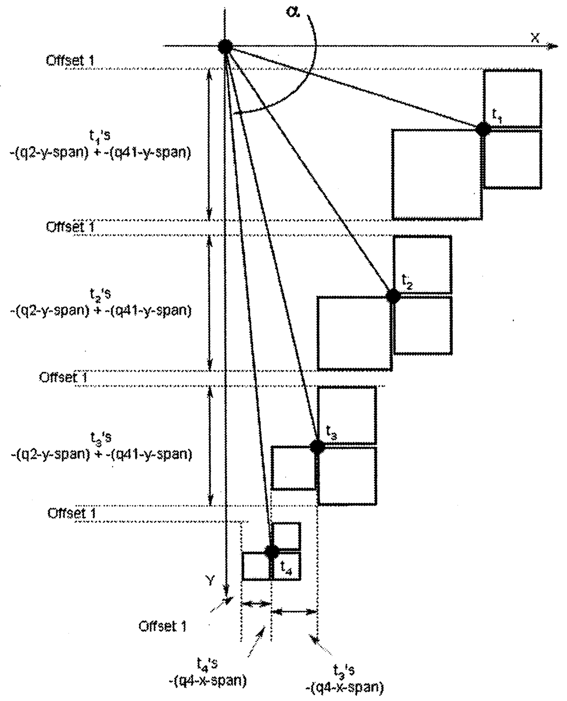


Figure 3.28 A drawing of a node with several three-quad drawing subtrees, all subtrees' directional span values are known, so the last two subtrees t_3 and t_4's location is easy to find. The angle formed by edges from the parent node to t_3 and t_4 is also easy to compute.

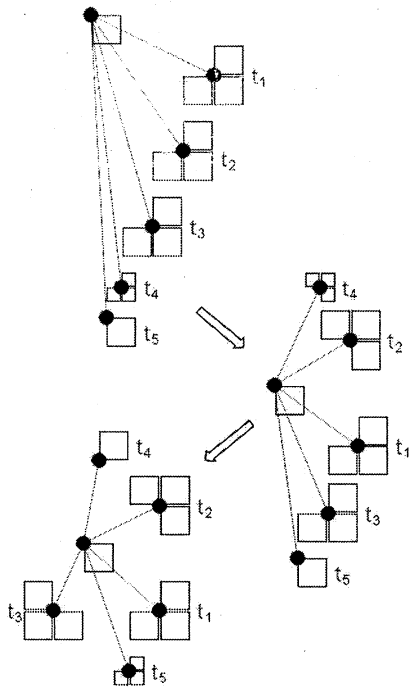


Figure 3.29 The process of distributing subtrees into three quads when the angular coefficient cannot be met by one quad or two quads.

3.6.1 Random general tree generation

We developed a simple program in C++ to generate randomly structured general trees. The program takes two parameters: the size of the tree which is expected to be at least 1 and the maximum degree allowed on each node which is also at least 1. First the program creates the root node of the tree, then add a random number of leaves to the root. Next, the program runs iterations to add nodes to the tree until the size limit is reached. In each iteration, a randomly selected leaf node is converted to a tree node with a random number of leaves added to this node. Number of leaves added to a node is restricted by the maximum degree allowed on a node and the remaining size of the tree.

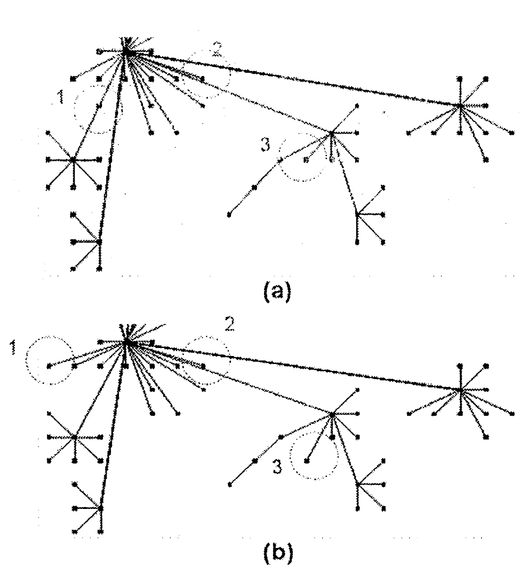


Figure 3.30 (a) Node 1 creates an overlap conflict; Node 2 creates a small angle conflict, but will not be honored since the angles between leaves are already below the angular coefficient; Node 3 creates a small angle conflict. (b) Node 1 has been reallocated into the open channel; Node 2 is not reallocated; Node 3 has been reallocated into the empty quad.

3.6.2 Evaluation process

To evaluate the algorithm, we varied n up to $20,000$. For each n, we have 10 different randomly generated trees, and use nine different angular coefficient values, from 5 to 45 in increments of 5.

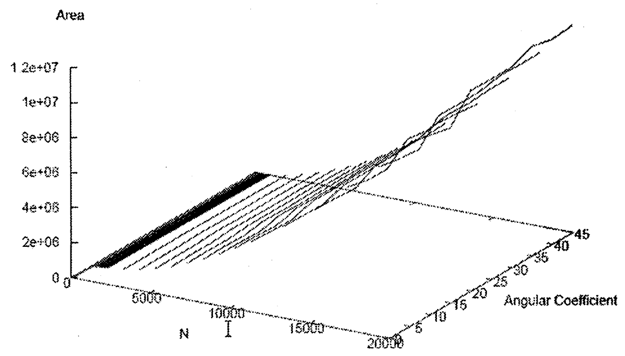


Figure 3.31 Tree performance of the algorithm on area versus angular coefficient for randomly generated general trees with different values of n (Number of nodes).

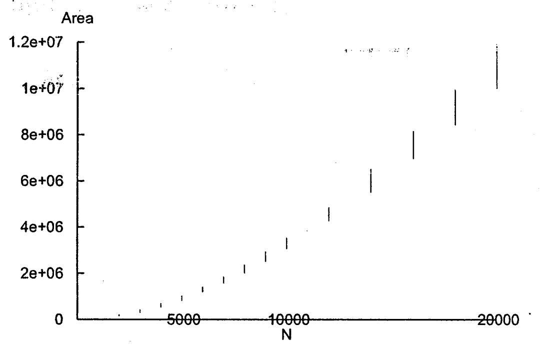


Figure 3.32 The projection of the X-Z plane of the graph shown in Figure 3.31. The area increases as expected as the value of *n* increases.

We then record the area of the drawing generated by *Quad* for each tree.

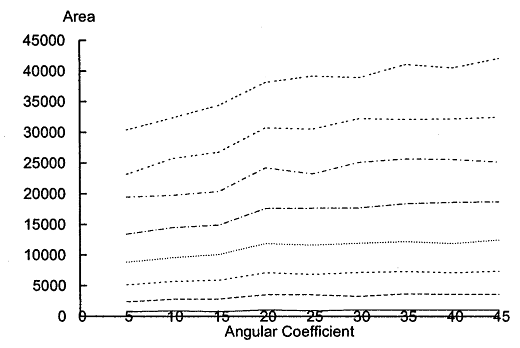


Figure 3.33 The projection of the X-Y plane of the graph shown in Figure 3.31. The rate at which the area increases, with respect to angular coefficient, increases as *n* increases.

3.6.3 Results

As a result of our experimental study (see Figure 3.31), we found that with a relatively small n (several hundred), increasing the angular coefficient has less of an impact on the area of the tree. As n increases, the area increases as expected (see Figure 3.32), and the rate at which the area increases, with respect to angular coefficient, increases (see Figure 3.33). When $n = 100$, increasing the angular coefficient from 5 to 45 has almost no impact on the area of the tree. However, when $n = 800$, increasing the angular coefficient from 5 to 45 increases the area from $30,000$ to over $40,000$.

3.7 Conclusion

The task of creating a general tree drawing with consistent and distinguishable angles is accomplished by Quad better than any other known existing algorithm (see Figure 3.34). *Quad* provides more room, maximum four quads, to allocate leaves and subtrees. *Quad* also uses angular coefficient to avoid small edges angles. *Quad*'s tendency to grow in four directions (see Figure 3.10) also naturally creates a good aspect ratio. Edge crossings and overlappings are nonexistent in quad, and due to its subtree separation property, related nodes are clustered. These two factors create an overall structure that is easy to read and understand (see Figure 3.35). Quad avoids computationally intensive operations, allowing it to perform its task quickly.

Finally, the angular coefficient provided by the user can be used to focus the algorithm on the user's preference of distinguishable angle or small area. A small angular coefficient will create a drawing with small area, but will be more crowded with small angles (see Figure 3.36), while a larger angular coefficient will result in larger area, but more separation between nodes (see Figure 3.37). *Quad* algorithm's main focus is not to achieve best area performance and it produces drawings

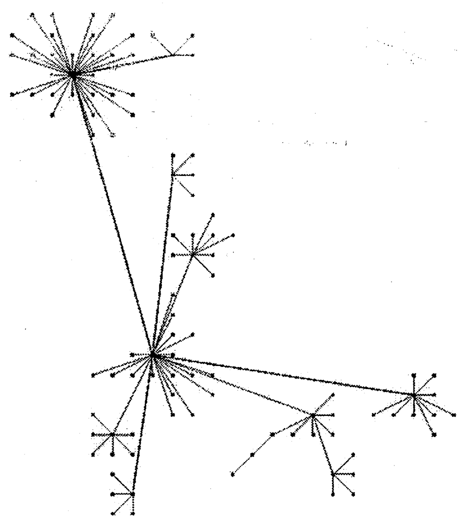


Figure 3.34 A 1-2-3-4-quad drawing of a general tree with several high degree nodes.

with area bigger than Separation algorithm. Figure 3.38, Figure 3.39, Figure 3.40, and Figure 3.41 are four drawings of same 100 nodes general tree with max degree equals to 7. Figure 3.38 is a drawing of the Separation algorithm and has the smallest area: 384. Figure 3.39, Figure 3.40, and Figure 3.41 are 3 drawings of *Quad* algorithm with different angular coefficients and shapes. All three of *Quad*'s drawings have larger area than Separation.

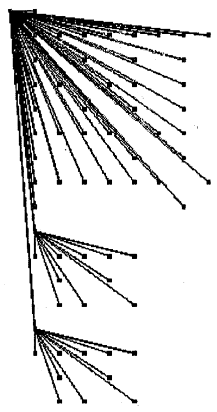


Figure 3.35 A tree with 40 leaves and two subtrees containing 10 leaves each. The angular resolution is small, around 0.5 degrees, but the tree is still understandable, because of subtree separation and the separation between leaves and subtrees.

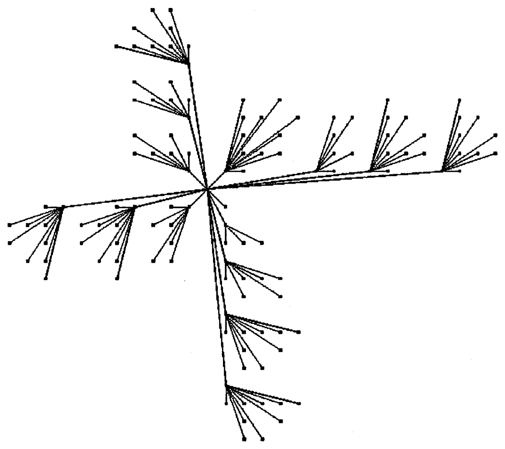


Figure 3.36 A 100 node general tree with angular coefficient of 1°. All edges are not very distinguishable, but the area of the drawing is 648.

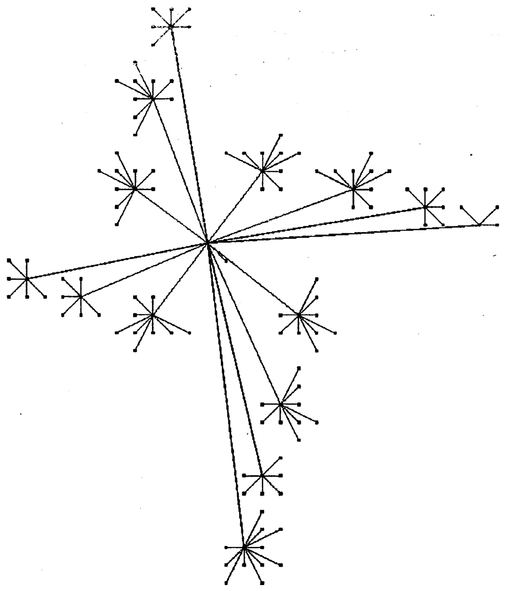


Figure 3.37 Same general tree as the one in Figure 3.36 with angular coefficient of 45°. This drawing has very distinguishable edges. However it has an area of 864.

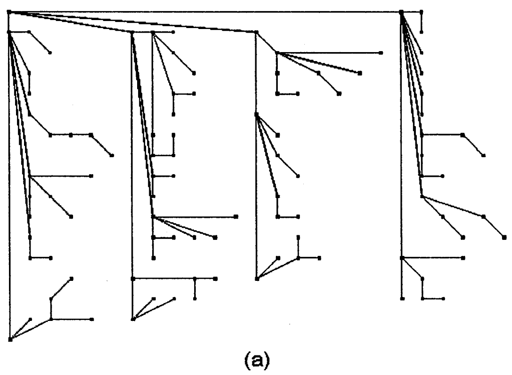


Figure 3.38 A low degree (max = 7) 100 nodes general tree drawing from Separation algorithm with aspect ratio expected to be 1. The area is 384.

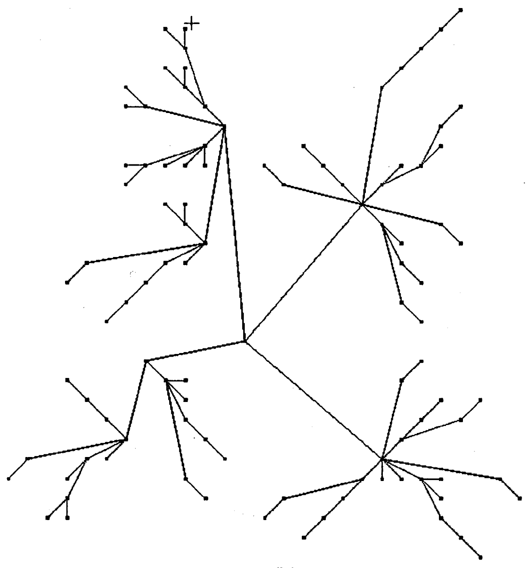
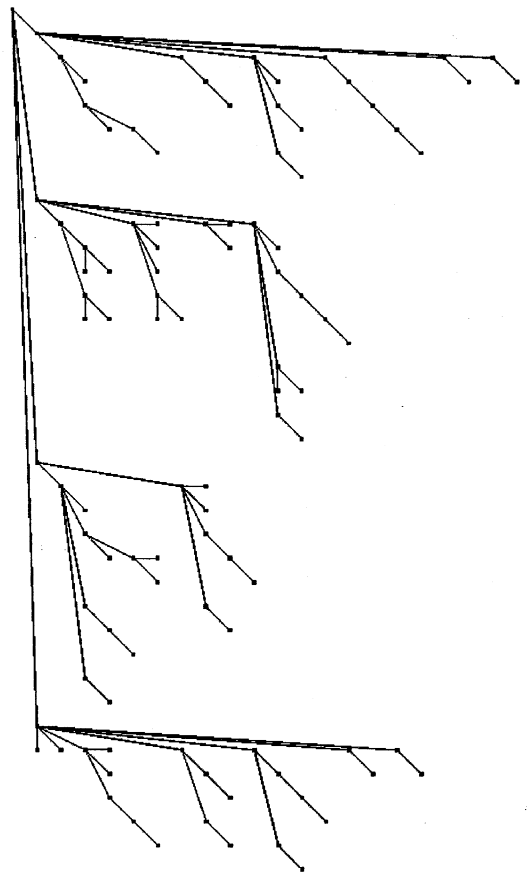
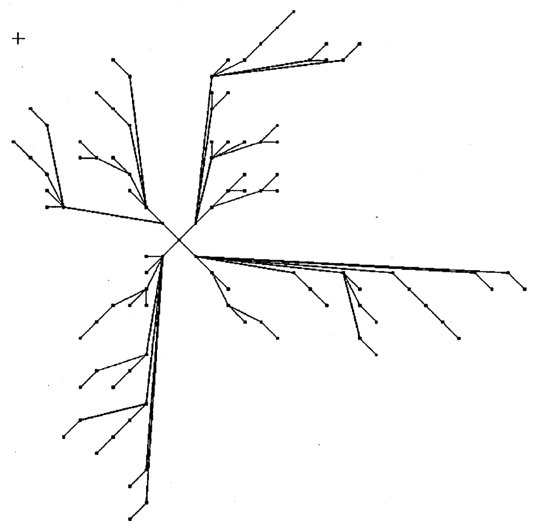

(b)

Figure 3.39 A low degree (max = 7) 100 nodes general tree drawing from *Quad* algorithm with angular coefficient equal to 5 and uses all 4 quads for root. The area of this drawing is 754.



(c)

Figure 3.40 A low degree (max = 7) 100 nodes general tree drawing from *Quad* algorithm with angular coefficient equal to 0 and only uses quad 1 for root. The area of this drawing is 735.



(d)

Figure 3.41 A low degree (max = 7) 100 nodes general tree drawing from *Quad* algorithm with angular coefficient equal to 0 and uses all 4 quads for root. The area of this drawing is 672.

4. CONCLUSION AND FUTURE WORK

This thesis has investigated problems related to the automatic generation of planar straight-line grid drawings of trees with visual distinguishable edge angles. Trees are very common data-structures. They are used to model information in a variety of applications, such as software engineering, biology, business administration, and web-site design. The area of a drawing is perhaps the most commonly used metric of the aesthetic quality of that drawing. However without visual distinguishable edge angles, the drawings are difficult to understand. We developed *Quad*, a new algorithm that enables the user to input an angular coefficient value adjust the edge angle. For future work, we are going to develop an application that uses *Quad* to produce drawings for COI (Community Of Interest) graphs (see Figure 4.1) which is used by AT&T to flag or throttle the traffic in telephony network. *Quad*'s efficiency in regard to area could also be improved, while making sure the running time of the algorithm does not increase.

This thesis also defined several types of general trees. These general tree types are based on many existing general tree characteristic definitions and few new characteristic definitions introduced in this thesis.

Both topics discussed in this thesis are part of building the environment for an experimental study on general tree drawing algorithms. *Quad* will be one of the drawing algorithms to be tested, along with H-V and Separation. Classifying general trees allows future studies to perform on specific type of trees. It is possible to find the best algorithm to draw one or more types of general

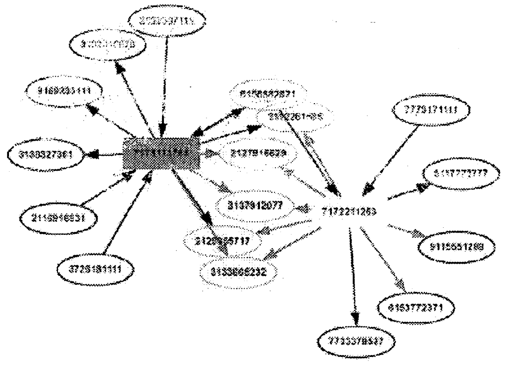


Figure 4.1 A drawing of a COI (Community of Interest) graph [*Image from: Adrian Rusu, Yehuda Koren, presentation at AT&T labs 2006 University Collaboration Symposium.]*.

trees, one or more particular aesthetics, or a combination of both. So for future work, we need to complete building the experimental study environment and perform the experimental study.

REFERENCES

[1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1999.

[2] 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 *Symposium on Computational Geometry*, pages 306–315, 1995.

[3] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *CGTA: Computational Geometry: Theory and Applications*, 7, 1997.

[4] G. Di Battista, A. Garg, G. Liottaa, 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.

[5] F. Brandenburg, M. Himsholt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Graph Drawing*, pages 76–87, 1995.

[6] 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.

[7] E. H. Chi and S. K. Card. Sensemaking of evolving web sites using visualization spreadsheets. In *INFOVIS*, page 18, 1999.

[8] 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.

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

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

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

[12] 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, pages 320–331. Springer-Verlag, 2002.

[13] 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.

[14] 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, pages 876–885. Springer, 2003.

[15] 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.

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

[17] 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.

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

[19] G. Melanon and I. Herman. Circular drawings of rooted trees, 1998.

[20] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 248–261. Springer-Verlag, 1997.

[21] 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), 1997.

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

[23] 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, New York, NY, USA, 1991. ACM Press.

[24] 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 Conference on Information Visualization*, Zurich, Switzerland, 2007. IEEE Computer Society.

[25] A. Rusu, C. Yao, and A. Crowell. A planar straight-line grid drawing algorithm for high degree general trees with user-controlled angular resolution. In *IV '08: Proceedings of the Conference on Information Visualization*, London, United Kingdom, 2008. IEEE Computer Society.

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

[27] 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.

[28] R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, SMC-18(1):61–79, 1988.

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

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

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

[32] 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.