2002

Computer investigation on super edge-magic trees

Xinshan Qian
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

---

Recommended Citation

Qian, Xinshan, "Computer investigation on super edge-magic trees" (2002). *Master's Theses*. 2371.
DOI: <https://doi.org/10.31979/etd.9cxt-s2un>
https://scholarworks.sjsu.edu/etd_theses/2371

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, Mi 48106-1346 USA
800-521-0600**

UMI®

Computer Investigation On Super Edge-Magic Trees

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Xinshan Qian

December 2002

UMI Number: 1411624

UMI Microform 1411624

**Copyright 2003 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**

**ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**

©2002

Xinshan Qian

ALL RIGHT RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Sin-min Lee

Dr. Brad Jackson

Dr. Ho Kuen Ng

APPROVED FOR THE UNIVERSITY

ABSTRACT

COMPUTER INVESTIGATION ON SUPER EDGE-MAGIC TREES

by Xinshan Qian

A (p, q) -graph $G = (V, E)$ is total edge-magic if there exists a bijection $f: V \cup E \rightarrow \{1, 2, \dots, p + q\}$ such that $\forall e = (u, v) \in E, f(u) + f(e) + f(v) = \text{constant}$. A total edge-magic graph is a super edge-magic graph if $f(V(G)) = \{1, 2, \dots, p\}$. Kotzig and Rosa showed that all caterpillars are super edge-magic. Enomoto et al. conjectured that all trees are super edge-magic. Various aspects of super edge-magic graphs are studied in this writing project. Computers are used to search for super edge-magic labelings. A computer program that verifies that all trees with less than seventeen vertices are super edge-magic is presented. Some theorems for extending a caterpillar to a super edge-magic lobster are also described.

Contents

Part I Super Edge-magic Labelings

1. Introduction	1
2. Some Known Results on Edge-magic and Super Edge-magic Labelings	4
3. All Trees with Less than 17 Vertices Are Super Edge-magic by Computer	5
4. A Super Edge-magic Extension of Caterpillars	5
5. Construction of Super Edge-magic Lobsters	14
6. L-rotatable Caterpillars	21
7. Graphs that Are not Super Edge-magic	25
8. Full binary Trees	27
9. Mongolian Tents	31
10. Computer Search for Super Edge-magic Labelings	33
11. The Super Edge-magic Labelings of All 10-verticesTree	36

Part II Generation of All Trees of a Fixed Order

1. Tree Applications in Computer science	53
2. Definitions	53
3. Some Properties of Isomorphic Trees	55
4. Enumeration of Trees	57
5. Generation of the Full Set of n Order Trees	58
5.1. The Basic Ideas for Generation of the Full Set of n Order Trees	58
5.2. The Three Rules for Non-isomorphic Trees.	59
5.3. Example 1 for Using Rule 1	59
5.4. Example 2 for Using Rule 2	60
5.5. Example 3 for Using Rule 3	62

6. The Data Structures Used in the Generation of Trees	64
6.1. Edge	64
6.2. Length of Path	64
6.3. Tree	65
7. The Implementation for Generation of Trees	65
7.1. Calculate the Degree of a Vertex	65
7.2. Calculate the Length of a Path in the Tree	67
7.3. Generation of Trees	71
7.4. Improve the Running Time	72
7.4.1. Running on a Computer with a Small Memory	72
7.4.2. Running on a Computer with a Large Memory	73
7.4.3. Trade-off on Memory Usage and the CPU Usage	74
Appendix A: Generation of Trees Source Code	75
Appendix B: Search for Super Edge-Magic Labelings Source Code	91
References	95

List of Figures

Part I Super Edge-magic Labelings

1. The Corresponding Supermagic Labeling of $K_{3,3}$	2
2. Super Edge-magic Labeling of C_7	3
3. Arrange the Vertices of a Bipartite Graph.	6
4. The Lobster after Redrawn.	7
5. The Super Edge-magic Labeling when $a \leq c$	7
6. The Super Edge-magic Labeling when $a \geq c$	8
7. The Lobster when k is even after Redrawn.	8
8. The Super Edge-magic Labeling when $a \leq c$	8
9. The Super Edge-magic Labeling when $a \geq c$	9
10. Sample of the Spider.	9
11. An Example of Theorem 1.	10
12. An Example of Theorem 1.	11
13. An Example of Theorem 1.	11
14. An Example of Theorem 1.	12
15. An Example of Theorem 1.	12
16. The Spider $SP(n^1 3^1 3^1)$	13
17. The Super Edge-magic Labeling when $n=1$	13
18. The Super Edge-magic Labeling when $n>1$ and $n=2a$	13
19. The Super Edge-magic Labeling when $n>1$ and $n=2a+1(a \geq 1)$	14
20. The Super Edge-magic Labeling of $SP(10,3,3)$	14

21. The Super Edge-magic Labeling of $SP(11,3,3)$	14
22. The Super Edge-magic Labeling of Caterpillar $T(3,(1,2,1))$	15
23. The Super Edge-magic Labeling of Caterpillar $T(3,(2,1,1))$	15
24. The Super Edge-magic Labeling of Caterpillar $T(3,(2,1,2))$	16
25. The Super Edge-magic Labeling of Caterpillar $T(3,(2,2,1))$	16
26. The Super Edge-magic Labeling of Caterpillar $T(3,(3,1,1))$	17
27. The Super Edge-magic Labeling of Caterpillar $T(4,(1,0,1,1))$	17
28. The Super Edge-magic Labeling of Caterpillar $T(4,(1,1,1,1))$	18
29. The Super Edge-magic Labeling of Caterpillar $T(4,(1,0,2,1))$	18
30. The Super Edge-magic Labeling of Caterpillar $T(4,(2,0,0,2))$	19
31. The Super Edge-magic Labeling of Caterpillar $T(5,(1,0,0,1,1))$	19
32. The Super Edge-magic Labeling of Caterpillar $T(5,(1,0,1,0,1))$	20
33. The Super Edge-magic Labeling of Caterpillar $T(3,(1,4,1))$	20
34. The Super Edge-magic Labeling of Caterpillar $T(3,(1,3,2))$	21
35. The Super Edge-magic Labeling of Caterpillar $T(4,(2,0,0,2))$	21
36. The Super Edge-magic Labeling of $SP(1^1 2^3)$	22
37. The Super Edge-magic Labelings of the Caterpillar $T(4,(2,1,0,1))$	24
38. An Example of that a Caterpillar is not L-rotatable.	25
39. An Example of not Super Edge-magic Graph.	27
40. A Super Edge-magic Labeling of the Full Binary Tree with Depth 2	28
41. A Super Edge-magic Labeling of the Full Binary Tree with Depth 3.	28
42. A Super Edge-magic Labeling of the Full Binary Tree with Depth 4.	29

43. A Super Edge-magic Labeling of the Mongolian Tent with 5 Vertices. . .	32
44. A Super Edge-magic Labeling of the Mongolian Tent with 7 Vertices. . .	32
45. A Super Edge-magic Labeling of the Mongolian Tent with 9 Vertices. . .	32
46. The Mongolian Tent with 11 Vertices Has No Super Edge-magic Labeling	33
47. A Super Edge-magic Labeling of a 16-vertex Tree.	35

Part II Generation of All Free Trees With Less Than 19-Vertices

1. A Tree.	54
2. All Trees with 6 Vertices.	54
3. Two Same trees T1 and T2.	56
4. Two 10-vertex Trees which Have the Same Number of Leaves and the Same Sets of Lengths of Paths.	59
5. Two Trees Have the Same Degrees of Vertices.	60
6. Use Rule 3 to Differentiate Between two Trees.	62
7. Stack[1] and Stack[2]	70
8. Set up Three Stacks.	70
9. Push p1, p5, and p6 into Three Stacks Respectively.	71

List of Tables

Part I Super Edge-magic Labelings

1. A 3 x 3 Magic Square 1

Part II Generation of All Free Trees With Less Than 19-Vertices

1. The Number of Trees with p Vertices 58
2. The Degrees of Vertices of T1 and T2. 60
3. The Same Degrees of Vertices of T1 and T2. 60
4. The Degrees of Vertices of the Tree in Figure 1. 66

Part I Super Edge-magic Labelings

1. Introduction

All graphs in this paper are finite and simple (without loops and multiple edges).

A graph G has vertex set $V(G)$ and edge set $E(G)$.

A labeling or valuation of a graph is an injection (one-to-one mapping) from graph elements to integers. Following (Wallis), if the domain is the set of all vertices and edges, the labelings are called total labelings. If the domain is the vertex set or edge set alone, the labelings are called vertex labelings and edge labelings respectively. There is an enormous amount of literature built up on these kinds of labelings over the past three decades. This writing project focused on investigating the magic property of graph labelings.

The study of magic property of graph labelings was motivated by the notion of magic squares in the number theory. Sedlacek (1963) calls a graph magic if it has an edge labeling such that the sum of labels around any vertex is constant, independent of the choice of the vertex. These labeling have been studied by Stewart (1996). He calls a magic labeling supermagic if the set of edge labels consists of consecutive integers, starting from 1. The classic concept of an $n \times n$ magic square in combinatorics theory corresponds to a supermagic labeling of $K_{n,n}$. Stewart proved that $K_{n,n}$ is magic for all $n \geq 3$. Table 1 shows how a 3×3 magic square corresponds to a supermagic labeling of $K_{3,3}$.

Table 1. A 3×3 Magic Square

8	1	6
3	5	7
4	9	2

Figure 1. The corresponding supermagic labeling of $K_{3,3}$.

The magic property studied by Sedlacek and Stewart is about vertices in an edge labeling. The idea can be naturally applied to edges in a total labeling. If the sum of all labels associated with an edge is the same for all edges, there will be some magic property. To distinguish between these two cases, a labeling is defined to be vertex-magic if the sum of all labels associated with a vertex equals a constant independent of the choice of the vertex, and edge-magic if the same property holds for edges. What Sedlacek and Stewart study is vertex-magic edge labelings.

The edge-magic graphs were first studied by Kotzig and Rosa in 1970. They defined a magic total labeling of a graph $G(V, E)$ as a bijection f from $V \cup E$ to $\{1, 2, \dots, |V| + |E|\}$ such that for all edges uv , $f(u) + f(v) + f(uv)$ is constant. Recently Enomoto et al have introduced the notion super edge-magic for magic labelings that all vertices receive labels from $\{1, 2, \dots, |V|\}$.

The topic of this paper is super edge-magic total labelings. For brevity a "labeling" will mean a total labeling in later sections.

Definition 1 An total edge-magic labeling of a graph G with p vertices and q edges is a one-to-one correspondence f from $V(G) \cup E(G)$ to $\{1, 2, \dots, p+q\}$ such that for all edges uv , $f(u) + f(v) + f(uv)$ is constant.

A graph G is called edge-magic if there exists an edge-magic labeling of G .

Definition 2 An super edge-magic labeling of a graph G with p vertices and q edges is an edge-magic labeling f of G with the property that $f(V(G)) = \{1, 2, \dots, p\}$.

A graph G is called super edge-magic if there exists a super edge-magic labeling of G .

For example, Figure 2 shows a super edge-magic labeling of C_7 .

Figure 2. Super edge-magic labeling of C_7 .

Definition 3 A tree is a connected, acyclic, undirected graph.

Let $G = (V, E)$ be an undirected graph. The following statements are equivalent.

- i G is a tree.
- ii Any two vertices in G are connected by a unique simple path.
- iii G is connected, but if any edge is removed from E , the resulting graph is

disconnected.

- iv G is connected, and $|E| = |V| - 1$.
- v G is acyclic, and $|E| = |V| - 1$.
- vi G is acyclic, but if any edge is added to E , the resulting graph contains a cycle.

2. Some Known Results on Edge-magic and Super Edge-magic Labelings

Kotzig and Rosa (1970) prove: $K_{m,n}$ has an edge-magic labeling for all m and n ; C_n is edge-magic for all $n \geq 3$; K_n is edge-magic if and only if $n = 1, 2, 3, 4, 5$, or 6 .

Wallis et al. prove that the following graphs are edge-magic: paths, crowns, complete bipartite graphs, and suns.

Enomoto, Llado, Nakamigana and Ringel (1998) prove that all complete bipartite graphs and all even cycles are edge-magic. They also prove that wheels W_n ($W_n = C_n + K_1$) are not edge-magic when $n \equiv 0 \pmod{4}$ and conjecture that all other wheels are edge-magic.

Ringel and Llado prove that a graph with p vertices and q edges is not edge-magic if q is even and $p + q \equiv 2 \pmod{4}$ and each vertex has odd degree. It follows from this that W_n is not edge-magic when $n \equiv 3 \pmod{4}$ thereby disproving the conjecture of Enomoto et al.

Enomoto et al. prove that C_n is super edge-magic if and only if n is odd, caterpillars (a caterpillar is a tree with the property that the removal of its endpoints leaves a path) are super edge-magic, $K_{m,n}$ is super edge-magic if and only if $m = 1$ or $n = 1$, and K_n is super edge-magic if and only if $n = 1, 2$ or 3 . They prove that if a (p, q)

graph is super edge-magic then $q \leq 2p - 3$. They conjecture that all trees are super edge-magic.

Z. Chen proves: the join of K_l with any star is super edge-magic; for any $p > 1$ and $q > 0$ there is a connected graph with p vertices and q edges if and only if $p - 1 \leq q \leq 2p - 3$; if a k -regular graph is super edge-magic then $k \leq 3$; there is a connected 3-regular super edge-magic graph with p vertices if and only if $p \equiv 2 \pmod{4}$; the join of two nontrivial graphs is super edge-magic if and only if at least one of them has exactly two vertices and the union of the two graphs has exactly one edge.

3. All Trees with Less Than 17 Vertices Are Super Edge-magic by Computer

Theorem 1 All trees with less than 17 vertices are super edge-magic.

There is a computer program that can generate at least one super edge-magic labeling for every tree with less than 17 vertices. Such a program is appended to this paper.

4. A Super Edge-magic Extension of Caterpillar

The typical caterpillar is a graph $T(n, (a_1, a_2, \dots, a_n))$ where $D(T(n, (a_1, a_2, \dots, a_n))) = P_n$ and $a_1 \geq 1, a_n \geq 1, a_i \geq 0$ for $i = 2, \dots, n-1$. We can view $T(n, (a_1, a_2, \dots, a_n))$ as $S_1 \cup S_2 \cup \dots \cup S_n$ where S_i is a star with center c_i and $a_i + 1$ edges, and in every case S_i shares an edge with S_{i+1} . The path with vertices $\{c_1, c_2, \dots, c_n\}$ will be called the **spine** of T . We will view the path P_n with $n \geq 4$ as caterpillar with spine P_{n-2} .

The tree has $n + a_1 + a_2 + \dots + a_n$ vertices and $n + a_1 + a_2 + \dots + a_n - 1$ edges. Assume $V(S_i) = \{c_i, x_{i,1}, x_{i,2}, \dots, x_{i,a_i}\}$ for each i . As T is a bipartite graph we can arrange the vertices of the tree as follows (Figure 3):

Figure 3. Arrange the vertices of a bipartite graph.

Let $X = \{x_{1,1}, x_{1,2}, x_{1,a1}, c_2, x_{3,1}, x_{3,2}, \dots, x_{3,a3}, c_4, \dots\}$ consists of all leaves of S_{2k+1} and center vertices of the form c_{2k} and $Y = \{c_1, x_{2,1}, x_{2,2}, \dots, x_{2,a2}, c_3, \dots\}$ consists of all leaves of S_{2k} and center vertices of the form c_{2k+1} . Now we label the vertices of Y from left to right by $4, \dots, \Sigma c_{2k} + \lceil n/2 \rceil + 3$ and X from left to right by $\Sigma c_{2k} + \lceil n/2 \rceil + 4, \dots, n+3 + \Sigma c_i$.

The labeling gives the consecutive set $\{4, 5, \dots, n+2 + \Sigma c_i\}$ on the edge labels.

We introduce here, a general construction of super edge-magic graphs from amalgamation of graphs. Recall that a graph G , with a fixed vertex u is amalgamated with another graph H with vertex v is the disjoint union of G and H and then identify u and v . We shall denote this new graph by $\text{Amal}((G, u), (H, v))$.

Let (P_m, u) denote the path P_m with the end vertex u . We can prove the following result.

Theorem 2 Every caterpillar T can be extended to a super edge-magic lobster by amalgamating (P_3, u) to any vertex of the spine of T which are not the end-vertices.

Proof. Assume the caterpillar $T = T(n, (a_1, a_2, \dots, a_n))$ where the spine $D(T(n, (a_1, a_2, \dots, a_n))) = P_n$ and

$a_1 \geq 1, a_n \geq 1, a_i \geq 0$ for $i=2, \dots, n-1$. We also assume $V(P_3) = \{v_1, v_2, u\}$. Assume P_3 is amalgamated with c_k . Consider two cases:

Case 1. If k is odd.

Count the number of the vertices of the lobster $G = \text{Amal}((P_{n+2}, c_k), (P_3, u))$ as follows (Figure 4).

Figure 4. The lobster after re-drawn.

If $a \leq c$, then we label the vertices of G as follows (Figure 5):

Figure 5. The super edge-magic labeling when $a \leq c$.

If $a \geq c$, then label the vertices of G as follows (Figure 6):

Figure 6. The super edge-magic labeling when $a \geq c$.

Case 2. If k is even.

Figure 7. The lobster when k is even after re-drawn.

If $a \leq c$, then we label the vertices of G as follows (Figure 8):

Figure 8. The super edge-magic labeling when $a \leq c$.

If $a \geq c$, then label the vertices of G as follows (Figure 9):

Figure 9. The super edge-magic labeling when $a \geq c$.

It induces a consecutive labeling on the edges. $\square$

A tree is called a **spider** if it has a center vertex c of degree $k > 1$ and all the other vertices are either a leaf or with degree 2. Thus a spider is an amalgamation of k paths with various lengths. If it has x_1 paths of length a_1 , x_2 paths of length a_2 , $\dots$, we shall denote the spider by $SP(a_1^{x_1} a_2^{x_2}, \dots, a_m^{x_m})$ where $a_1 < a_2 < \dots < a_m$ and $x_1 + x_2 + \dots + x_m = k$. (see Figure 35)

Figure 10. Sample of the spider.

Corollary 1 The spider $SP(2^l m^1 n^1)$ is super edge-magic for all $m, n > 1$.

Example 1 Let $T=P_{10}$, then we have the following super edge-magic lobsters.

(Figure 10). The first one is P_3 amalgamated with P_{10} on c_3 . The second one is P_3 amalgamated with P_{10} on c_4 .

Figure 11. An Example of Theorem 1.

Example 2 Consider $T(3,(6,2,5))$ amalgamated at c_2 . We list two different super edge-magic labelings (see Figure 12):

Figure 12. An example of Theorem 1.

Example 3 Consider $T(8, (6, 2, 2, 1, 3, 2, 2, 1))$ amalgamated at c_4 . We list two different super edge-magic labelings (see Figure 13):

Figure 13. An example of Theorem 1.

Example 4 Let the caterpillar be $T(7, (2, 2, 0, 1, 0, 2, 3))$ and a lobster G formed by amalgamating T with P_3 at c_3 . The super edge-magic labeling is shown as follows:

Figure 14. An example of Theorem 1.

Example 5 Let the caterpillar be $T(7, (6, 2, 2, 1, 3, 2, 3))$ and a lobster formed by amalgamating T with P_3 at c_4 . The super edge-magic labeling is shown as follows: (Figure 15)

Figure 15. The example of Theorem 1.

Theorem 3 The spider $SP(3^1 3^1 n^1)$ (Figure 16) is super edge-magic for all $n \geq 1$.

Figure 16. The spider $SP(n^1 3^1 3^1)$.

Proof.

If $n=1$, the labeling can be given as follows (Figure 17):

Figure 17. The super edge-magic labeling when $n=1$.

If $n>1$ and $n=2a$, labeling can be given as follows (Figure 18),

Figure 18. The super edge-magic labeling when $n>1$ and $n=2a$.

If $n > 1$ and $n = 2a + 1$, the labeling will be given as follows (Figure 19)

Figure 19. The super edge-magic labeling when $n > 1$ and $n = 2a + 1$ ($a \geq 1$).

Example 6

Figure 20. The super edge-magic labeling of $SP(10,3,3)$.

Example 7

Figure 21. The super edge-magic labeling of $SP(11,3,3)$.

5. Construction of Super Edge-magic Lobsters

Theorem 4 The caterpillar $T(3, (1,2,1))$ has an extension to infinitely many super edge-magic lobsters by amalgamation with stars at some vertex of the tree.

Proof. We give here two different labelings as follows (Figure 22):

Figure 22. The super edge-magic labeling of caterpillar $T(3, (1,2,1))$.

Theorem 5 The caterpillar $T(3, (2,1,1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here three different labelings as follows (Figure 23).

Figure 23. The super edge-magic labeling of caterpillar $T(3, (2,1,1))$.

Theorem 6 The caterpillar $T(3,(2,1,2))$ has an extension to infinitely many super edge-magic lobsters.

Proof.

Figure 24. The super edge-magic labeling of caterpillar $T(3,(2,1,2))$.

Theorem 7 The caterpillar $T(3,(2,2,1))$ can have an extension to infinitely many super edge-magic lobsters.

Proof.

Figure 25. The super edge-magic labeling of caterpillar $T(3,(2,2,1))$.

Theorem 8 The caterpillar $T(3,(3,1,1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here a labeling as follows (Figure 26):

Figure 26. The super edge-magic labeling of caterpillar $T(3, (3, 1, 1))$.

Theorem 9 The caterpillar $T(4, (1, 0, 1, 1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here two different labelings as follows (Figure 27)

Figure 27. The super edge-magic labeling of caterpillar $T(4, (1, 0, 1, 1))$.

Theorem 10 The caterpillar $T(4, (1, 1, 1, 1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here a labeling as follows (Figure 28):

Figure 28. The super edge-magic labeling of caterpillar $T(4, (1, 1, 1, 1))$.

Theorem 11 The caterpillar $T(4, (1, 0, 2, 1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here three different labelings as follows (Figure 29):

Figure 29. The super edge-magic labeling of caterpillar $T(4, (1, 0, 2, 1))$.

Theorem 12 The caterpillar $T(4, (2, 0, 0, 2))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here a labeling as follows (Figure 30)

Figure 30. The super edge-magic labeling of caterpillar $T(4, (2, 0, 0, 2))$.

Theorem 13 The caterpillar $T(5, (1, 0, 0, 1, 1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here three different labelings as follows (Figure 31):

Figure 31. The super edge-magic labeling of caterpillar $T(5, (1, 0, 0, 1, 1))$.

Theorem 14 The caterpillar $T(5, (1,0,1, 0, 1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here a super edge-magic labeling as follows (Figure 32):

Figure 32. The super edge-magic labeling of caterpillar $T(5, (1,0,1,0,1))$.

Theorem 15 The caterpillar $T(3, (1,4,1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here a labeling as follows (Figure 33)

Figure 33. The super edge-magic labeling of caterpillar $T(3, (1,4,1))$.

Theorem 16 The caterpillar $T(3, (1,3,2))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here a labeling as follows (Figure 34)

Figure 34. The super edge-magic labeling of caterpillar $T(3, (1, 3, 2))$.

Theorem 17 The caterpillar $T(3, (4, 1, 1))$ has an extension to infinitely many super edge-magic lobsters.

Proof. We give here a labeling as follows (Figure 35)

Figure 35. The super edge-magic labeling of caterpillar $T(4, (2, 0, 0, 2))$.

6. L-rotatable Caterpillars

Observation 1. The spider $SP(1^{x_1}k^1)$ is a caterpillar for all $x_1 > 0$ and $k > 1$.

Observation 2. The spider $SP(2^{x_1})$ $x_1 > 2$ and $SP(1^{x_1}2^{x_2})$ $x_2 > 2$ are lobsters.

Observation 3. The spider $SP(1^k2^2)$ for any $k > 0$ is a caterpillar.

Professor Lee give next the L-rotatable concept.

Definiton 3 A tree T is called L -rotatable if for any u in T , the amalgamation $\text{Amal}((T,u), (\text{St}(n),c))$ is a super-magic tree for any $n > 0$.

Theorem 18 The spider $\text{SP}(1^1 2^3)$ is L -rotatable.

Proof.

Figure 36. The super edge-magic labeling of $\text{SP}(1^1 2^3)$.

Remark Not all caterpillars are L -rotatable. Consider the caterpillar $T(4,(2,1,0,1))$. We list now all its possible super-magic labelings.

	p1	p2	p3	p4	p5	p6	p7	p8
1:	1	4	2	5	3	6	7	8
2:	1	4	2	5	3	8	6	7
3:	1	5	3	7	4	8	2	6
4:	1	6	2	4	7	5	3	8
5:	1	6	5	4	8	2	7	3
6:	1	6	7	5	3	8	4	2
7:	1	7	5	2	8	3	6	4
8:	1	7	5	6	4	3	2	8
9:	1	8	3	5	7	6	2	4
10:	1	8	5	2	6	4	3	7
11:	2	3	4	5	1	7	8	6
12:	2	4	1	7	3	8	5	6
13:	2	5	6	4	8	1	3	7
14:	2	5	7	4	6	3	8	1
15:	2	5	8	1	7	3	6	4
16:	2	8	5	7	1	6	3	4
17:	3	2	4	7	1	6	8	5

	p1	p2	p3	p4	p5	p6	p7	p8
18:	3	2	4	7	1	8	5	6
19:	3	4	1	8	2	6	7	5
20:	3	5	7	4	6	1	8	2
21:	3	6	1	4	2	8	5	7
22:	3	8	2	6	1	5	4	7
23:	4	3	8	5	7	1	6	2
24:	4	5	8	2	6	1	7	3
25:	4	7	5	2	8	1	6	3
26:	5	1	7	3	8	4	6	2
27:	5	3	2	4	6	1	8	7
28:	6	1	4	2	8	3	7	5
29:	6	1	4	7	3	5	8	2
30:	6	3	7	5	2	4	8	1
31:	6	4	3	5	1	8	7	2
32:	7	1	6	4	2	3	8	5
33:	7	2	4	3	5	6	8	1
34:	7	4	6	2	5	1	8	3

We see that $\text{Amal}((T(4, (2, 1, 0, 1), p_i), (\text{St}(n), c)))$ is super-magic for all $i=1, 2, \dots, 7$ and all n . (Figure 37)

Figure 37. The super edge-magic labelings of the caterpillar $T(4, (2, 1, 0, 1))$.

I next list all the super-edge-magic labeling while $p_9 = 9$. The list indicate that no labeling can be found, which when vertices $9+1, 9+2, \dots, 9+n$ are connected to vertex p_8 , the new labeling $p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, 9, 9+1, 9+2, \dots, 9+n$ is still super-edge-magic

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
1:	5	2	8	1	7	6	4	3	9
2:	4	2	8	1	7	6	5	3	9
3:	7	3	5	1	4	8	6	2	9
4:	6	3	5	1	4	8	7	2	9
5:	5	4	8	6	7	1	3	2	9
6:	3	4	8	6	7	1	5	2	9
7:	8	5	7	2	4	3	6	1	9
8:	6	5	7	2	4	3	8	1	9
9:	4	5	7	6	8	3	2	1	9
10:	2	5	7	6	8	3	4	1	9
11:	7	6	5	4	2	8	1	3	9
12:	1	6	5	4	2	8	7	3	9
13:	7	6	8	4	5	3	1	2	9
14:	1	6	8	4	5	3	7	2	9
15:	2	7	4	6	8	5	1	3	9
16:	1	7	4	6	8	5	2	3	9
17:	4	7	6	8	2	5	1	3	9
18:	1	7	6	8	2	5	4	3	9
19:	5	8	6	1	7	4	2	3	9
20:	2	8	6	1	7	4	5	3	9
21:	3	8	6	2	5	7	1	4	9
22:	1	8	6	2	5	7	3	4	9

Figure 38. An example of that a caterpillar is not L-rotatable.

7. Graphs that Are Not Super Edge-magic

Theorem 19 A graph G is not super edge-magic if $|E(G)| \equiv 2 \pmod{4}$, and each vertex of G has even degree.

Proof. Let $G = (V, E)$ be a super edge-magic graph with p vertices and q edges. Suppose f is a super edge-magic labeling of G and s is the constant number satisfying $s = f(u) + f(v) + f(uv)$ for any edge uv in G . Then

$$qs = \sum_{v \in V} d(v)f(v) + \sum_{e \in E} f(e) \quad (1)$$

where $d(v)$ is the degree of v .

Because f is a super edge-magic labeling, we have $f(V \cup E) = \{1, 2, \dots, p+q\}$ and $f(V) = \{1, 2, \dots, p\}$. Thus, $f(E) = f(V \cup E) - f(V) = \{1, 2, \dots, p+q\} - \{1, 2, \dots, p\} = \{p+1, p+2, \dots, p+q\}$. (2) can be gotten

$$\sum_{e \in E} f(e) = (p+1) + (p+2) + \dots + (p+q) = pq + q(q+1)/2 \quad (2)$$

Replace the sum of edge labels in (1) with (2), (3) can be gotten

$$qs = \sum_{v \in V} d(v)f(v) + pq + q(q+1)/2$$

$$\sum_{v \in V} d(v)f(v) = q(s-p) - q(q+1)/2 \quad (3)$$

Suppose $q = 4n + 2$ for some positive integer n and $d(v)$ is even for each v in V .

Then

$$q(s-p) - q(q+1)/2 = 2(2n+1)(s-p) - (2n+1)(4n+3)$$

is an odd number. On the other hand,

$$\sum_{v \in V} d(v)f(v)$$

is an even number. This is a contradiction. $\square$

Corollary 2 If a graph G is constructed by connecting all corresponding vertices of two identical trees T , then G is not super edge-magic when $|V(T)| \equiv 0 \pmod{4}$ and each vertex of T has odd degree.

Proof. Suppose T has $4n$ vertices for some positive integer n .

Then $|E(T)| = 4n - 1$.

Then $|E(G)| = 2|E(T)| + 4n = 2(4n - 1) + 4n = 4(3n - 1) + 2$. Because the degree of every vertex is increased by 1, every vertex has even degree. So, G is not super edge-magic by Theorem 16. $\square$

For example, the graph shown in Figure 39 has no super edge-magic labelings.

Figure 39. An example of not super edge-magic graph.

8. Full Binary Trees

A full binary tree is a binary tree in which each vertex has exactly zero or two children and all leaves are at the same level.

Theorem 20 If there is a super edge-magic labeling of a full binary tree, the label of the root is an odd number.

Proof. Let $T = (V, E)$ be a super edge-magic graph with p vertices and q edges. Suppose f is a super edge-magic labeling of T and s is the constant number satisfying $s = f(u) + f(v) + f(uv)$ for any edge uv in T . Use $d(v)$ to denote the degree of vertex v in T . From equation (3) in the proof of Theorem 16, next (1) can be gotten

$$\sum_{v \in V} d(v)f(v) = q(s-p) - q(q+1)/2 \quad (1)$$

Suppose the depth of T is d . Then $p = 2^d - 1$ and $q = p - 1 = 2^d - 2$.

$$q(s-p) - q(q+1)/2 = q(s-p) - (2^{d-1} - 1)(2^d - 1)$$

So the right-hand side of (1) is an odd number. Now let us check the parity of the left-hand side of (1).

Because $f(V) = \{1, 2, \dots, 2^d - 1\}$, 2^{d-1} vertices will receive odd labels. Note that every vertex except for the root has odd degree, if the label of the root is even, then all odd labels will be used to label vertices that have odd degrees. The left-hand side of (1) is an even number because of the even number of odd labels,. This is a contradiction. $\square$

Full binary trees with depths 2, 3, and 4 are super edge-magic with odd number root labeling, as follows (Figure 40, 41, and 42).

Figure 40. A super edge-magic labeling of the full binary tree with depth 2.

Figure 41. A super edge-magic labeling of the full binary tree with depth 3.

Figure 42. A super edge-magic labeling of the full binary tree with depth 4.

Lemma 1 (Chen) If a graph G has a super edge-magic labeling f then $\{f(u) + f(v) : uv \in E(G)\}$ is consecutive.

Lemma 2 (Wallis) If a graph G has a super edge-magic labeling f then $|V(G)| + 1 - f$ is a super edge-magic labeling of G .

If a full binary tree T is super edge-magic then there is a super edge-magic labeling of T such that the root of T receives a label that is less than $|V(T)|/2$. This can be easily proved using Lemma 2.

Theorem 21 If a full binary tree T is super edge-magic, and f is a super edge-magic labeling of T , and the label of root is less than $|V(T)|/2$, then

$$(2a^2 + a)/(2a - 1) \leq \min(\{f(u) + f(v) : uv \in E(G)\}) < (12a^2 - 5)/(4a - 2)$$

where $a = 2^{d-2}$, and d is the depth of T .

Proof. Use $\deg(v)$ to denote the degree of a vertex v . Define

$$R = \{f(v) : v \in V(T) \text{ and } \deg(v) = 2\},$$

$$M = \{f(v) : v \in V(T) \text{ and } \deg(v) = 3\},$$

$$L = \{f(v) : v \in V(T) \text{ and } \deg(v) = 1\},$$

$$Y = \{f(u) + f(v) : uv \in E(T)\}.$$

Clearly, $|R| = 1$, $|Y| = |E(T)| = |V(T)| - 1 = 2^d - 2$, $|L| = 2^{d-1}$, $|M| = 2^{d-1} - 2$.

Define

$$r = \sum_{i \in R} I$$

$$m = \sum_{i \in M} i$$

$$l = \sum_{i \in L} i$$

$$y = \sum_{i \in Y} i$$

Clearly,

$$\begin{aligned} y &= 2r + 3m + l \\ &= r + 2m + (r + m + l) \\ &= r + 2m + (1 + 2 + \dots + (2^d - 1)) \\ &= r + 2m + 2^{d-1}(2^d - 1) \\ &= r + 2m + 8a^2 - 2a. \end{aligned} \tag{1}$$

On the other hand, Y is consecutive by Lemma 1, assume $t = \min(Y)$, then

$$\begin{aligned} y &= t + (t+1) + \dots + (t+2^d-2-1) \\ &= t(2^d-2) + (2^{d-1}-1)(2^d-3) \\ &= t(4a-2) + (2a-1)(4a-3) \end{aligned} \tag{2}$$

From (1) and (2),

$$t(4a-2) + (2a-1)(4a-3) = r + 2m + 8a^2 - 2a$$

$$t(4a - 2) = r + 2m + 8a - 3$$

$$t = (r + 2m + 8a - 3)/(4a - 2) \quad (3)$$

M can consist of the $|M|$ smallest labels, the $|M|$ largest labels, or somewhere in between, so

$$(2^{d-1} + 2) + \dots + (2^d - 1) \leq m \leq 1 + 2 + \dots + (2^{d-1} - 2)$$

$$2a^2 - 3a + 1 \leq m \leq 6a^2 - 5a - 1 \quad (4)$$

$$\text{Clearly, } 1 \leq r \leq |V(T)|/2 = (2^d - 1)/2 < 2a \quad (5)$$

Apply (4) and (5) to (3), next can be gotten

$$(2a^2 + a)/(2a - 1) \leq t < (12a^2 - 5)/(4a - 2) \quad \square$$

Lemma 3 A graph G is super edge-magic if there is a vertex labeling f such that $f(V(G)) = \{1, 2, \dots, |V(G)|\}$ and $\{f(u) + f(v) : uv \in E(G)\}$ consists of consecutive numbers.

Lemma 3 is very useful for testing if a graph is super edge-magic. Use the consecutive numbers starting from 1 to label all vertices, then add the labels of the two endpoints of each edge, if the sums are consecutive then the graph is super edge-magic. Sometimes it will help to design a labeling of the vertices if the smallest sum is known. Theorem 18 gives the range of the smallest sum of a full binary tree. For example, the smallest sum is between 6 and 13 when the depth of a full binary tree is 4.

9. Mongolian Tents

Remark 1 There is a Mongolian tent that is not super edge-magic.

The Mongolian tent with 11 vertices is not super edge-magic.

Mongolian tents with 5, 7, and 9 vertices are super edge-magic, as shown in Figure 43, 44, and 45. Figure 46 shows the Mongolian tent that is not super edge-magic.

Figure 43. A super edge-magic labeling of the mongolian tent with 5 vertices.

Figure 44. A super edge-magic labeling of the mongolian tent with 7 vertices.

Figure 45. A super edge-magic labeling of the mongolian tent with 9 vertices.

Figure 46. The mongolian tent with 11 vertices has no super edge-magic labeling.

10. Computer Search for Super Edge-magic Labelings

It has been conjectured that all trees are total edge-magic (Lee, 1999).

Following is the basic algorithm to try out all permutations of the labels to find all super edge-magic labelings of a graph.

```
f(in, cnt){
    if( cnt > 0 )
        for(i=0; i<cnt; i++){
            v[cnt] = in[i]; // label the vertex
            new_in = in - {in[i]}
            f(new_in, cnt-1);
        }
    // all vertices have been labeled
    else if( the labeling is super edge-magic )
        display the labeling
}
```

This algorithm will run days on a PC for a graph with just 15 vertices. However, the algorithm can be improved in four aspects.

i) Duality (Wallis)

By Lemma 2 every super edge-magic labeling of a graph has a dual labeling that is super edge-magic. The use of duality will cut the number of permutations in half.

When a label is assigned to the first vertex, it suffices to check either all cases with label $\leq |V(G)|$ or all cases with label $\geq |V(G)|$.

ii) Indexability (Acharya, 1991)

Definition 3 Let f be a vertex labeling of G , define $f^*(uv) = f(u) + f(v)$ for any edge uv of G . G is said to be *indexable* if there is a vertex labeling f such that $f(V(G)) = \{1, 2, \dots, |V(G)|\}$ and f^* is injective. f is called an *indexer* of G .

Clearly, any super edge-magic labeling must be an indexer. If the label of an edge is equal to the label of any previous edge then the permutation is not an indexer. Hence, there is no need to continue to complete the permutations.

iii) Symmetry

Many graphs are symmetric. Taking advantage of the symmetry of a graph can significantly reduce the number of permutations to be tested. For example, a full binary tree is symmetric, if set the label of the left child be always less than the label of the right child of any parent, the number of permutations to be tested will be divided by $2^{(|V(G)|/2)}$.

iv) Graph specific properties

Every graph has its own special properties. These properties might be used as constraints to reduce the number of permutations to be tested. For example, by Theorem 20, the root of a full binary tree is always odd in any super edge-magic labelings, so any even labels will not be assigned to the root.

After the four improvements the running time to test a full binary tree with 15 vertices is reduced from two days to less than one hour.

Even though the basic algorithm can be improved in one way or another, the complexity of super-edge-magic labelings is still there. Using the computer, it has been verified that all trees with vertices less than 17 are super edge-magic.

Figure 47 is a sample that a super edge-magic labeling of a 16-vertex tree.

Figure 47. A super edge-magic labeling of a 16-vertex tree.

11. The Super Edge-magic Labelings of All 10-vertex Trees

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

46

47

48

49

50

51

52

53

54

55

56

57

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

101

102

103

104

105

106

Part II Generation of All Free Trees With Less Than 19-Vertices

1. Tree Applications in Computer Science

Trees are a combinatorial structure that has many important applications in computer science. They are highly appreciated as elementary tools. They are used, for example, to sort and search data, and to represent mathematical expressions in a form that can be manipulated more easily.

Generally speaking, they are a good instrument to save structures that can be described recursively, or that appear when faster algorithms, using methods of the type "divide and conquer" or more heuristic methods (in-depth search, "min-max" algorithm), are employed.

Trees are also very useful for analyzing the behavior of a large number of algorithms.

They allow visualizing the calls of a recursive algorithm; they also appear as structures that are homeomorphic to other structures that are used in certain analyses.

2. Definitions

The **degree** of a vertex v is the number of edges incident with v . In Figure 1, the degree of vertex p_2 is 4 because there are 4 edges incident with p_2 (p_1 - p_2 , p_3 - p_2 , p_5 - p_2 , p_6 - p_2).

$D(T)$ is used to denote the set of degrees of vertices. In Figure 1, $D(T) = (1^4, 2^1, 4^1)$, which indicates that there are four points whose degrees are 1, one point whose degree is 2, and one point whose degree is 4.

Figure 1. A tree.

A **leaf** is a vertex with degree one. In Figure 1, p_1 , p_4 , p_5 , p_6 are leaves.

An **inner vertex** is a vertex with at least degree two. In Figure 1, p_2 and p_3 are inner vertices.

The **n -vertex tree** is a tree with n vertices. Figure 1 shows a 6-vertex tree.

The **full set of n -vertex trees** is all the free trees with n vertices. Figure 2 shows the full set of 6 size trees, there are six trees that are built with six vertices.

Figure 2. All trees with 6 vertices.

A **path** is a list $(n_1, n_2, \dots, n_p)$ such that there exists an edge which connects the vertices n_i and n_{i+1} for each i in $[1, p-1]$. In Figure 1, p_4, p_3, p_2, p_6 is a path that

connects vertices p_4 and p_6 . $n_1 - n_p$ is used to denote a path that connects vertices n_1 and n_p .

The **length of a path** is the number of edges in the path. In Figure 1, the length of path $p_4 - p_6$ is 3.

$L(T)$ is used to denote the set of all lengths of paths in tree T .

$$L(T) = \begin{matrix} & \begin{matrix} P1 & p2 & p3 & p4 & p5 & p6 \end{matrix} \\ \begin{matrix} P1 \\ P2 \\ P3 \\ P4 \\ P5 \\ P6 \end{matrix} & \begin{pmatrix} -- & 1 & 2 & 3 & 2 & 2 \\ 1 & -- & 1 & 2 & 1 & 1 \\ 2 & 1 & -- & 1 & 2 & 2 \\ 3 & 2 & 1 & -- & 3 & 3 \\ 2 & 1 & 2 & 3 & -- & 2 \\ 2 & 1 & 2 & 3 & 2 & -- \end{pmatrix} \end{matrix}$$

$SL(T)$ is used to denote a summary of the lengths of the paths in tree T .

$$SL(T) = \begin{pmatrix} (1^1, 2^3, 3^1)^3 \\ (1^4, 2^1)^1 \\ (1^2, 2^3)^1 \\ (1^1, 2^1, 3^3)^1 \end{pmatrix}$$

$(1^1, 2^3, 3^1)^3$ indicates that there are 3 sets $(1^1, 2^3, 3^1)$ and $(1^1, 2^3, 3^1)$ indicate that there are 1 path of which length is 1, 3 paths of which length is 2 and 1 path of which length is 3 from a vertex to other all vertices in the tree.

3. Some Properties of Isomorphic Trees

Two trees are isomorphic or identical if there exists a one-to-one correspondence between their vertex sets which preserves adjacency. If two trees are isomorphic, we have the following observations.

- Same set of degrees of vertices

For example, T1 and T2 in Figure 3 are identical. Degree set of T1 is $D(T1) = (1^7, 2^2, 4^1, 5^1)$. And degree set of T2 is also $D(T2) = (1^7, 2^2, 4^1, 5^1)$.

Figure 3. Two same trees T1 and T2.

- Same summary set of lengths

$$L(T1) = \begin{matrix} & P1 & p2 & p3 & p4 & p5 & p6 & p7 & p8 & p9 & p10 & p11 \\ \begin{matrix} P1 \\ P2 \\ P3 \\ P4 \\ P5 \\ P6 \\ P7 \\ P8 \\ P9 \\ P10 \\ P11 \end{matrix} & \begin{pmatrix} -- & 4 & 4 & 2 & 1 & 2 & 3 & 4 & 5 & 2 & 4 \\ 4 & -- & 2 & 4 & 3 & 2 & 1 & 2 & 3 & 4 & 2 \\ 4 & 2 & -- & 4 & 3 & 2 & 1 & 2 & 3 & 4 & 2 \\ 2 & 4 & 4 & -- & 1 & 2 & 3 & 4 & 5 & 2 & 4 \\ 1 & 3 & 3 & 1 & -- & 1 & 2 & 3 & 4 & 1 & 3 \\ 2 & 2 & 2 & 2 & 1 & -- & 1 & 2 & 3 & 2 & 2 \\ 3 & 1 & 1 & 3 & 2 & 1 & -- & 1 & 2 & 3 & 1 \\ 4 & 2 & 2 & 4 & 3 & 2 & 1 & -- & 1 & 4 & 2 \\ 5 & 3 & 3 & 5 & 4 & 3 & 2 & 1 & -- & 5 & 3 \\ 2 & 4 & 4 & 2 & 1 & 2 & 3 & 4 & 5 & -- & 4 \\ 4 & 2 & 2 & 4 & 3 & 2 & 1 & 2 & 3 & 4 & -- \end{pmatrix} \end{pmatrix}$$

$$SL(T1) = \begin{pmatrix} (1^1, 2^3, 3^1, 4^4, 5^1)^3 \\ (1^1, 2^1, 3^4, 4^1, 5^3)^1 \\ (1^1, 2^4, 3^2, 4^3)^3 \\ (1^4, 2^1, 3^4, 4^1)^1 \\ (1^2, 2^4, 3^1, 4^3)^1 \\ (1^2, 2^7, 3^1)^1 \\ (1^5, 2^2, 3^3)^1 \end{pmatrix}$$

$$L(T_2) = \begin{matrix} & p1 & p2 & p3 & p4 & p5 & p6 & p7 & p8 & p9 & p10 & p11 \\ \begin{matrix} P1 \\ P2 \\ P3 \\ P4 \\ P5 \\ P6 \\ P7 \\ P8 \\ P9 \\ P10 \\ P11 \end{matrix} & \begin{pmatrix} -- & 4 & 2 & 1 & 2 & 3 & 4 & 2 & 2 & 4 & 3 \\ 4 & -- & 4 & 3 & 2 & 1 & 2 & 4 & 4 & 2 & 5 \\ 2 & 4 & -- & 1 & 2 & 3 & 4 & 2 & 2 & 4 & 3 \\ 1 & 3 & 1 & -- & 1 & 2 & 3 & 1 & 1 & 3 & 2 \\ 2 & 2 & 2 & 1 & -- & 1 & 2 & 2 & 2 & 2 & 3 \\ 3 & 1 & 3 & 2 & 1 & -- & 1 & 3 & 3 & 1 & 4 \\ 4 & 2 & 4 & 3 & 2 & 1 & -- & 4 & 4 & 2 & 5 \\ 2 & 4 & 2 & 1 & 2 & 3 & 4 & -- & 2 & 4 & 1 \\ 2 & 4 & 2 & 1 & 2 & 3 & 4 & 2 & -- & 4 & 3 \\ 4 & 2 & 4 & 3 & 2 & 1 & 2 & 4 & 4 & -- & 5 \\ 3 & 5 & 3 & 2 & 3 & 4 & 5 & 1 & 3 & 5 & -- \end{pmatrix} \end{pmatrix}$$

$$SL(T_2) = \begin{pmatrix} (1^1, 2^3, 3^1, 4^4, 5^1)^3 \\ (1^1, 2^1, 3^4, 4^1, 5^3)^1 \\ (1^1, 2^4, 3^2, 4^3)^3 \\ (1^4, 2^1, 3^4, 4^1)^1 \\ (1^2, 2^4, 3^1, 4^3)^1 \\ (1^2, 2^7, 3^1)^1 \\ (1^5, 2^2, 3^3)^1 \end{pmatrix}$$

Although the $L(T_1) \neq L(T_2)$ because the trees are not ordered and rooted, $SL(T_1) = SL(T_2)$. They are the same.

4. Enumeration of Trees

Table 1 is copied from Harary's book "Graph Theory". It shows how many trees there are for certain vertex trees.

Table 1. The Number of Trees with p Vertices

p	number of trees	p	number of trees
1	1	13	1301
2	1	14	3159
3	1	15	7741
4	2	16	19320
5	3	17	48629
6	6	18	123867

7	11	19	317955
8	23	20	823065
9	47	21	2144065
10	106	22	5623756
11	235	23	14828074
12	551	24	39299897

5. Generation of the Full set of n -vertex Trees

5.1. The Basic Ideas for Generation of the Full Set of n Order Trees

The basic method is to generate trees by increasing the number of vertices. To generate the set of n -vertex trees, first trees with 2 vertices is generated, then trees with 3 vertices, trees with 4 vertices, . . . , and finally trees with n vertices. In general, generating all free trees with n vertices bases on free trees with $n-1$ vertices.

At the beginning, the full set of free trees with n vertices is $\emptyset$. Then for every tree in the full set of free trees with $i-1$ -vertices, add one vertex that connects the vertices from 1 to $i-1$ respectively to construct new free trees with $i-1$ vertices. And then check if any of the new trees is already in the set of free trees with n -vertices. If not, add this tree to the set of free trees with n -vertices.

5.2. The Three Rules for Non-isomorphic Trees

To generate the full set of free trees with n -vertices, it is important to ensure that no two trees in the full set of free trees with n -vertices are isomorphic. The algorithm for checking if two trees are isomorphic is essential. The following three rules is used to check if a new tree is not identical to any existing trees in the set.

Rule 1 Calculate the set of degrees of vertices for both trees. If their sets of degrees are not exactly the same, the two trees are different. Otherwise use rule 2 to continue.

Rule 2 Calculate the summary sets of path lengths of both trees and compare them. If the summary sets of path lengths are different, the two trees are different. Otherwise use rule 3 to continue.

Rule 3 Select a group of vertices $\{v_1, v_2, \dots, v_n\}$ whose degrees are the biggest in the two trees, compute the set of lengths from $v_1, v_2, \dots, v_n$ to every leaf in the two trees, and compare the lengths of two trees.

5.3. Example 1 for Using Rule 1

Figure 4. Two 10-vertex trees which have the same number of leaves and the same sets of lengths of paths.

The set of degrees of vertices of T1 is not the same as that of T2. See Table 2.

Table 2. The Degrees of Vertices of T1 and T2

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
T1	1	3	2	4	2	1	1	1	2	1
T2	1	2	3	3	3	1	2	1	1	1

The set of degree of T1 $\{1^5, 2^3, 3^1, 4^1\}$ and the set of degree of T2 $\{1^5, 2^2, 3^3\}$ are not same. So, they are not same.

5.4. Example for Using Rule 2

Figure 5. Two trees have the same degrees of vertices.

Table 3. The Same Degrees of Vertices of T1 and T2

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11
T1	1	1	3	4	3	2	1	1	1	2	1
T2	1	1	3	4	2	2	1	1	3	1	1

The set of degree of T1 $\{1^6, 2^2, 3^2, 4^1\}$ and the set of degree of T2 $\{1^6, 2^2, 3^2, 4^1\}$ are the same. But, they are not same trees.

$$D(T1) = (1^6, 2^2, 3^2, 4^1) \quad D(T2) = (1^6, 2^2, 3^2, 4^1)$$

$$\Rightarrow D(T1) = D(T2)$$

$$L(T1) = \begin{matrix} & \begin{matrix} p1 & p2 & p3 & p4 & p5 & p6 & p7 & p8 & p9 & p10 & p11 \end{matrix} \\ \begin{matrix} p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \\ p10 \\ p11 \end{matrix} & \begin{pmatrix} -- & 3 & 2 & 1 & 2 & 3 & 4 & 3 & 2 & 3 & 4 \\ 3 & -- & 1 & 2 & 3 & 4 & 5 & 2 & 3 & 4 & 5 \\ 2 & 1 & -- & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \\ 1 & 2 & 1 & -- & 1 & 2 & 3 & 2 & 1 & 2 & 3 \\ 2 & 3 & 2 & 1 & -- & 1 & 2 & 3 & 2 & 1 & 2 \\ 3 & 4 & 3 & 2 & 1 & -- & 1 & 4 & 3 & 2 & 3 \\ 4 & 5 & 4 & 3 & 2 & 1 & -- & 5 & 4 & 3 & 4 \\ 3 & 2 & 1 & 2 & 3 & 4 & 5 & -- & 3 & 4 & 5 \\ 2 & 3 & 2 & 1 & 2 & 3 & 4 & 3 & -- & 3 & 4 \\ 3 & 4 & 3 & 2 & 1 & 2 & 3 & 4 & 3 & -- & 1 \\ 4 & 5 & 4 & 3 & 2 & 3 & 4 & 5 & 4 & 1 & -- \end{pmatrix} \end{pmatrix}$$

$$SL(T1) = \begin{pmatrix} (1^1, 2^3, 3^4, 4^2)^2 \\ (1^1, 2^2, 3^3, 4^2, 5^2)^2 \\ (1^3, 2^3, 3^2, 4^2)^1 \\ (1^4, 2^4, 3^2)^1 \\ (1^3, 2^5, 3^2)^1 \\ (1^2, 2^2, 3^4, 4^2)^1 \\ (1^1, 2^1, 3^2, 4^4, 5^2)^2 \\ (1^2, 2^2, 3^3, 4^2)^1 \end{pmatrix}$$

$$L(T2) = \begin{matrix} & \begin{matrix} p1 & p2 & p3 & p4 & p5 & p6 & p7 & p8 & p9 & p10 & p11 \end{matrix} \\ \begin{matrix} p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \\ p10 \\ p11 \end{matrix} & \begin{pmatrix} -- & 3 & 2 & 1 & 2 & 3 & 4 & 3 & 2 & 3 & 4 \\ 3 & -- & 1 & 2 & 3 & 4 & 5 & 2 & 3 & 4 & 4 \\ 2 & 1 & -- & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 3 \\ 1 & 2 & 1 & -- & 1 & 2 & 3 & 2 & 1 & 2 & 2 \\ 2 & 3 & 2 & 1 & -- & 1 & 2 & 3 & 2 & 3 & 3 \\ 3 & 4 & 3 & 2 & 1 & -- & 1 & 4 & 3 & 4 & 4 \\ 4 & 5 & 4 & 3 & 2 & 1 & -- & 5 & 4 & 5 & 5 \\ 3 & 2 & 1 & 2 & 3 & 4 & 5 & -- & 3 & 4 & 4 \\ 2 & 3 & 2 & 1 & 2 & 3 & 4 & 3 & -- & 1 & 1 \\ 3 & 4 & 3 & 2 & 3 & 4 & 5 & 4 & 1 & -- & 2 \\ 4 & 4 & 3 & 2 & 3 & 4 & 5 & 4 & 1 & 2 & -- \end{pmatrix} \end{pmatrix}$$

$$SL(T2) = \begin{pmatrix} (1^1, 2^3, 3^5, 4^1)^1 \\ (1^1, 2^2, 3^3, 4^3, 5^1)^4 \\ (1^3, 2^3, 3^3, 4^1)^2 \\ (1^4, 2^5, 3^1)^1 \\ (1^2, 2^4, 3^4)^1 \\ (1^2, 2^1, 3^3, 4^4)^1 \\ (1^1, 2^1, 3^1, 4^3, 5^4)^1 \end{pmatrix}$$

$$SL(T1) \neq SL(T2),$$

So, use the rule2, T1 and T2 are not the same.

5.5. Example for Using Rule 3

Figure 6. Use rule 3 to differentiate between two trees.

$$L(T1) = \begin{matrix} & \begin{matrix} p1 & p2 & p3 & p4 & p5 & p6 & p7 & p8 & p9 & p10 & p11 & p12 & p13 & p14 & p15 & p16 & p17 & p18 \end{matrix} \\ \begin{matrix} p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \\ p10 \\ p11 \\ p12 \\ p13 \\ p14 \\ p15 \\ p16 \\ p17 \\ p18 \end{matrix} & \begin{pmatrix} -- & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 1 & -- & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 2 & 1 & -- & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 \\ 2 & 1 & 2 & -- & 2 & 2 & 3 & 3 & 3 & 3 & 1 & 1 & 3 & 3 & 4 & 4 & 4 & 4 \\ 2 & 1 & 2 & 2 & -- & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 1 & 3 & 4 & 4 & 4 & 4 \\ 2 & 1 & 2 & 2 & 2 & -- & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 1 & 4 & 4 & 4 & 4 \\ 3 & 2 & 1 & 3 & 3 & 3 & -- & 2 & 2 & 2 & 4 & 4 & 4 & 4 & 1 & 1 & 3 & 3 \\ 3 & 2 & 1 & 3 & 3 & 3 & 2 & -- & 2 & 2 & 4 & 4 & 4 & 4 & 3 & 3 & 1 & 3 \\ 3 & 2 & 1 & 3 & 3 & 3 & 2 & 2 & -- & 2 & 4 & 4 & 4 & 4 & 3 & 3 & 3 & 1 \\ 3 & 2 & 1 & 3 & 3 & 3 & 2 & 2 & 2 & -- & 4 & 4 & 4 & 4 & 3 & 3 & 3 & 3 \\ 3 & 2 & 3 & 1 & 3 & 3 & 4 & 4 & 4 & 4 & -- & 2 & 4 & 4 & 5 & 5 & 5 & 5 \\ 3 & 2 & 3 & 1 & 3 & 3 & 4 & 4 & 4 & 4 & 2 & -- & 4 & 4 & 5 & 5 & 5 & 5 \\ 3 & 2 & 3 & 3 & 1 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & -- & 4 & 5 & 5 & 5 & 5 \\ 3 & 2 & 3 & 3 & 3 & 1 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & -- & 5 & 5 & 5 & 5 \\ 4 & 3 & 2 & 4 & 4 & 4 & 1 & 3 & 3 & 3 & 5 & 5 & 5 & 5 & -- & 2 & 4 & 4 \\ 4 & 3 & 2 & 4 & 4 & 4 & 1 & 3 & 3 & 3 & 5 & 5 & 5 & 5 & 2 & -- & 4 & 4 \\ 4 & 3 & 2 & 4 & 4 & 4 & 3 & 1 & 3 & 3 & 5 & 5 & 5 & 5 & 4 & 4 & -- & 4 \\ 4 & 3 & 2 & 4 & 4 & 4 & 3 & 3 & 1 & 3 & 5 & 5 & 5 & 5 & 4 & 4 & 4 & -- \end{pmatrix} \end{matrix}$$

$$SL(T1) = \begin{pmatrix} (1^1, 2^4, 3^8, 4^4)^2 \\ (1^3, 2^4, 3^6, 4^4)^2 \\ (1^5, 2^8, 3^4)^2 \\ (1^2, 2^4, 3^7, 4^4)^4 \\ (1^1, 2^2, 3^4, 4^6, 5^4)^4 \\ (1^1, 2^1, 3^4, 4^7, 5^4)^4 \end{pmatrix}$$

$$L(T2) = \begin{matrix} & p1 & p2 & p3 & p4 & p5 & p6 & p7 & p8 & p9 & p10 & p11 & p12 & p13 & p14 & p15 & p16 & p17 & p18 \\ \begin{matrix} p1 \\ p2 \\ p3 \\ p4 \\ p5 \\ p6 \\ p7 \\ p8 \\ p9 \\ p10 \\ p11 \\ p12 \\ p13 \\ p14 \\ p15 \\ p16 \\ p17 \\ p18 \end{matrix} & \begin{pmatrix} - & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 1 & - & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 2 & 1 & - & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 \\ 2 & 1 & 2 & - & 2 & 2 & 3 & 3 & 3 & 3 & 1 & 1 & 3 & 3 & 4 & 4 & 4 & 4 \\ 2 & 1 & 2 & 2 & - & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 1 & 1 & 4 & 4 & 4 & 4 \\ 2 & 1 & 2 & 2 & 2 & - & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 3 & 2 & 1 & 3 & 3 & 3 & - & 2 & 2 & 2 & 4 & 4 & 4 & 4 & 1 & 3 & 3 & 3 \\ 3 & 2 & 1 & 3 & 3 & 3 & 2 & - & 2 & 2 & 4 & 4 & 4 & 4 & 3 & 1 & 3 & 3 \\ 3 & 2 & 1 & 3 & 3 & 3 & 2 & 2 & - & 2 & 4 & 4 & 4 & 4 & 3 & 3 & 1 & 3 \\ 3 & 2 & 3 & 1 & 3 & 3 & 4 & 4 & 4 & - & 4 & 4 & 4 & 4 & 5 & 5 & 5 & 5 \\ 3 & 2 & 3 & 1 & 3 & 3 & 4 & 4 & 4 & 4 & 2 & - & 4 & 4 & 5 & 5 & 5 & 5 \\ 3 & 2 & 3 & 3 & 1 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & - & 2 & 5 & 5 & 5 & 5 \\ 3 & 2 & 3 & 3 & 1 & 3 & 4 & 4 & 4 & 4 & 4 & 4 & 2 & - & 5 & 5 & 5 & 5 \\ 4 & 3 & 2 & 4 & 4 & 4 & 1 & 3 & 3 & 3 & 5 & 5 & 5 & 5 & - & 4 & 4 & 4 \\ 4 & 3 & 2 & 4 & 4 & 4 & 3 & 1 & 3 & 3 & 5 & 5 & 5 & 5 & 4 & - & 4 & 4 \\ 4 & 3 & 2 & 4 & 4 & 4 & 3 & 3 & 1 & 3 & 5 & 5 & 5 & 5 & 4 & 4 & - & 4 \\ 4 & 3 & 2 & 4 & 4 & 4 & 3 & 3 & 3 & 1 & 5 & 5 & 5 & 5 & 4 & 4 & 4 & - \end{pmatrix} \end{pmatrix}$$

$$SL(T2) = \begin{pmatrix} (1^1, 2^4, 3^8, 4^4)^2 \\ (1^3, 2^4, 3^6, 4^4)^2 \\ (1^5, 2^8, 3^4)^2 \\ (1^2, 2^4, 3^7, 4^4)^4 \\ (1^3, 2^2, 3^4, 4^6, 5^4)^4 \\ (1^1, 2^1, 3^4, 4^7, 5^4)^4 \end{pmatrix}$$

$$SL(T1) = SL(T2),$$

The group of vertices with the largest degrees are (p2, p3) in T1 and same in T2.

Next shows that the lengths from p2 and p3 to every leaf.

$$\begin{matrix} p2 \\ p3 \end{matrix} \begin{pmatrix} p1 & p10 & p11 & p12 & p13 & p14 & p15 & p16 & p17 & p18 \\ 1 & 2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 2 & 1 & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 \end{pmatrix} \quad T1$$

	p1	p6	p11	p12	p13	p14	p15	p16	p17	p18	
p2	1	1	2	2	2	2	3	3	3	3	T2
p3	2	2	3	3	3	3	2	2	2	2	

They are not same. Use rule 3, T1 are not same as T2.

6. The Data Structures Used in the Generation of Trees

6.1. Edge

```
class Edge
{
private:
    int p1;
    int p2;
    int used; //0 not yet checked; 1 already checked;
public:
    void setPoint(int v1,int v2);
    void setUsedFlag(int f);
    int getP1();
    int getP2();
    int getUsedFlag();
};
```

An edge connects two vertices. So in the data structure, use two points p1 and p2 to represent an edge. The "used" is a flag indicating if the edge has been checked.

6.2. Length of Path

```
class setOfLength
{
private:
    int length; //the length of the path that connects two vertieces
    int numberL; //the number of the length
public:
    int getLength();
    int getNumberL;
    void setLength(int L);
    void setNumberL(int n);
};
```

This structure stores the length of path that connect two vertieces. *length* is for the length of the path which connects two vertices. *int numberL* is for the number of the paths which have same length.

6.3. Tree

```

class Tree
{
private:
    Edge e[20];           //defines a tree
    int leafV[20];        // vertices which are leaves in the tree
    int degree[20];       // vertex degree
    int numberOfLeaves;    //the number of leaves in the tree
    int numberOfLines;     // the number of edges in the tree
    int columOfThisRow[20];
    int rowUsedFlag[20];
    int numberOfSetV;      //used for checking if the two trees are
    setOfLength setL[20];  //the same
    setOfLength setV[20];
    setOfLength setLTable[20][20];

public:
    Tree() { numberOfSetL=0;}
    void setLengthTable(int a, int leaf);
    void operator +=(Tree t1);
    void degreeOfVertieces();
    void cLengthTable();
    bool ifLeaf(int p);
    bool ifSameTree(Tree t);
};

```

7. The Implementation for Generation of Trees

7.1 Calculate the Degree of a Vertex

To calculate the degree of a vertex, just check every edge in the tree and find the number of edges that connect that vertex. For example, in Figure 1, the tree has six vertices and five edges. The tree is as following:

```

e1:    p1, p2
e2:    p2, p3

```

e3: p3, p4
e4: p2, p5
e5: p2, p6

From above, the degrees of vertices are in the Table 4:

Table 4. The Degrees of Vertices of the Tree in Figure 1

vertex	p1	p2	p3	p4	p5	p6
degree	1	4	2	1	1	1

From the Table 4, p1, p4, p5, p6 are leaf vertices and p2 and p3 are inner vertices.

/* count every degree of vertex and then count leave number */

void Tree::degreeOfVerticees()

```
{
    int cSetV=0;
    int leaveC=0;
    for (int i=1; i<=numberOfLines+1; i++) { // the number of vertices are numberOfLines+1
        int c=0;
        for (int j=0; j<numberOfLines; j++) {
            if(e[j].p1==i || e[j].p2==i)
                c++;
        }
        degree[i]=c;
        if (c==1) {
            leaveC++;
            leafV[leaveC]=i;
        }
        int flag=0;
        int k=1;
        while(k<=cSetV && flag==0) {
            if(setV[k].valueL==c) {
                flag=1;
                setV[k].numberL++;
            }
            k++;
        }
        if (flag==0) {
            cSetV++;
            setV[cSetV].valueL=c;
            setV[cSetV].numberL=1;
        }
    }
    numberOfLeaves=leaveC;
    numberOfSetV=cSetV;
}
```

} //For vertex I, check all the edges in the
//tree and find the edges include the vertex i.

} //If the vertex has degree one, it is a leaf.
//Save this vertex in the leaf[] array variable.

} //Statistics that how many vertices have the
//same degree, keep the degree in the
// degreeL variable and the number of that
// degree in the numberL.

} //If there has been not yet that degree before,
// the degree of numberL is one.

7.2. Calculate the Length of a Path in the Tree

The program for calculating the length of the path.

```

void Tree::setLengthTable(int a, int vertex)
{
    if(columnOfThisRow[vertex]==0) {
        columnOfThisRow[vertex]=1;
        setLTable[vertex][columnOfThisRow[vertex]].valueL=a;
        setLTable[vertex][columnOfThisRow[vertex]].numberL=1;
    } else {
        int i=1;
        int flag=0;
        while (i<=columnOfThisRow[vertex] && flag==0) {
            if(setLTable[vertex][i].valueL == a) {
                flag=1;
                setLTable[vertex][i].numberL++;
            } else {
                i++;
            }
        }
        if (flag==0) {
            columnOfThisRow[vertex]++;
            setLTable[vertex][columnOfThisRow[vertex]].valueL=a;
            setLTable[vertex][columnOfThisRow[vertex]].numberL=1;
        }
    }
}

void Tree::cLengthTable()
{
    int curSt;
    int foundP[20];
    Stack st[20];

    for(int i=1; i<=numberOfLines+1; i++) {
        columnOfThisRow[i]=0;
        rowUsedFlag[i]=0;
        for(int j=0; j<numberOfLines; j++) { //set the pre-value
            e[j].used=0; // for every edge, assign used flag 0
        }
        int usedLine=0;
        for(j=1; j<=numberOfLines; j++) {
            st[j].clear(); //Initialize the stacks is the number of
            st[j].setSizeOfStack(numberOfLines); // vertices int the tree minus one
        }
        int w=0;
        int findFlag=0;
        int countSt=0;
        while(w<numberOfLines)
        {

```

```

    if(e[w].p1 == i ) {
        countSt++;
        st[countSt].push(i);
        st[countSt].push(e[w].p2);
        e[w].used=1;
        usedLine++;
        w++;
    } else if(e[w].p2==i) {
        countSt++;
        st[countSt].push(i);
        st[countSt].push(e[w].p1);
        e[w].used=1;
        usedLine++;
        w++;
    } else {
        w++;
    }
}
curSt=countSt;
for(j=1; j<=numberOfLines+1; j++) { //need to creat numberOfLines+1 stacks
    int c=0;
    for(int k=0; k<numberOfLines; k++) {
        if (e[k].used==0) {
            if (e[k].p1 == st[j].curV()) {
                c++;
                foundP[c]=e[k].p2;
                e[k].used=1;
                usedLine++;
            } else if (e[k].p2== st[j].curV()) {
                c++;
                foundP[c]=e[k].p1;
                e[k].used=1;
                usedLine++;
            }
        }
    }
    if (c>0) {
        for(k=1; k<=c; k++) {
            st[++curSt].stackCopy(st[j]);
            st[curSt].push(foundP[k]);
        }
    }
}
for(int k=1; k<=numberOfLines; k++) {
    setLengthTable(st[k].getCur(),i); //getCur( ) returns the length of the path
}
}
}

```

}
 //Find the edges which
 //vertex i. That is, the
 // lengths from the vertex
 // i to other all vertices
 //in the tree will be compute.

In the generation of trees, the algorithm needs to calculate the path length between any two vertices, and uses the set of lengths $L(T)$ to check if the two trees are certainly different.

As mentioned above, if the two trees are the same, they should have the same set of lengths of paths $L(T1) = L(T2)$. So, if two trees do not have the same sets, the two trees are not the same.

Use the tree in Figure 1 to illustrate the algorithm for calculating the length of a path. The set of paths should be as follows.

$$L(T) = \begin{matrix} & \begin{matrix} P1 & p2 & p3 & p4 & p5 & p6 \end{matrix} \\ \begin{matrix} P1 \\ P2 \\ P3 \\ P4 \\ P5 \\ P6 \end{matrix} & \begin{pmatrix} -- & 1 & 2 & 3 & 2 & 2 \\ 1 & -- & 1 & 2 & 1 & 1 \\ 2 & 1 & -- & 1 & 2 & 2 \\ 3 & 2 & 1 & -- & 3 & 3 \\ 2 & 1 & 2 & 3 & -- & 2 \\ 2 & 1 & 2 & 3 & 2 & -- \end{pmatrix} \end{matrix} = \begin{bmatrix} (1^1, 2^3, 3^1)^3 \\ (1^4, 2^1)^1 \\ (1^2, 2^3)^1 \\ (1^1, 2^1, 3^3)^1 \end{bmatrix}$$

Now use the algorithm with the tree in Figure 1. The tree is

e1: p1, p2
e2: p2, p3
e3: p3, p4
e4: p2, p5
e5: p2, p6

For example, to calculate the lengths of paths that begin with p3 and end with all other vertices p1, p2, p4, p5, and p6, first find the edges which include the p3 vertex. Those are e2: p2, p3 and e3: p3, p4. Set up stack[1] and stack[2] with p3 in the stacks, push the vertex p2 (See Figure 7) into stack[1] and assign 1 to the used flag of edge e2 to indicate edge e2 is used; push the vertex p4 into stack[2] and also assign 1 to used flag of edge e3 to indicate edge e3 is used.

Figure 7. Stack[1] and Stack[2]

Next, search the edges that include the top of the stack[1] vertex p2 and are not used in all the edges of the tree. e1, e4, e5 are obtained. Set up three stacks: stack[3], stack[4], stack[5] and copy the stack[1] to these three stacks(Figure 8). e2 is first, then push the p3 into the stack[1] (See Figure 8) and assign the used flag of edge e2 1 to indicate this edge has been used.

Figure 8. Set up three stacks.

After three stacks have been set up, push p1 into stack[3], p5 into stack[4], and p6 into stack[5](Figure 9). At the same time, assign 1 to the used flags of the edges e1, e4,e5.

Figure 9. Push p1, p5, and p6 into three stacks respectively.

Continue to check stacks left one by one. Because in this example, in the stacks left, no edges which connect the top vertices of the stacks can be found, the lengths can be obtained from the current sizes of stacks.

p3 -- p2 1

p3 -- p4 1

p3 -- p1 2

p3 -- p5 2

p3 -- p6 2

7.3. Generation of Trees

After having the two algorithms above, the tree generation algorithm looks simple.

I generate the trees with n -vertices based on the trees with $n-1$ -vertices from the tree with 2-vertices. There is one kind 2-vertex tree that is p1 connecting p2. In the trees

with $n-1$ -vertices, for every kind of tree, from vertex 2 to vertex $n-1$ an edge is added $(2,n), (3,n), \dots, (n-1,n)$, then $n-1$ trees are built.

Then check if the newly generated trees are the same, and save the different trees in the full set of free trees with n -vertices.

```

fullSizeT[1].e[1].p1=1;
fullSizeT[1].e[1].p2=2;
for(int i=3; i<=treePoints; i++) ) } //2-vertices tree
                                     //Generate treePoints-vertices tree
{
    cE++;
    cT=0;
    for(int j=1; j<=fullCT; j++)
    {
        cT++;
        int curCT=1;
        t[curCT]=fullSizeT[j];
        t[curCT].numberOfSetL=0;
        t[curCT].numberOfEdges=cE;
        t[curCT].e[cE].p1=2;
        t[curCT].e[cE].p2=i;
    } // Add a edge 2,n

    for(int k=3; k<i; k++)
    {
        cT++;
        curCT++;
        t[curCT].e[cE].p1=k;
        t[curCT].e[cE].p2=i;
        for(int m=1; m<cE; m++)
        {
            t[curCT].e[m].p1=fullSizeT[j].e[m].p1;
            t[curCT].e[m].p2=fullSizeT[j].e[m].p2;
        }
    }
}

```

7.4 Improve the Running Time

7.4.1 Running on a Computer with a Small Memory

If the computer has a limited memory, the non-isomorphic trees is saved in a result file when they are obtained in the program. To get one non-isomorphic tree `tree_new`, the result file is opened, one tree `tree_old` is read one time and `tree_new` and

tree_old is compared. If tree_old is the same as tree_new, tree_new is thrown away. If tree_new is not as same as tree_old, read the result file to continue to get the next tree_old till result file reaches the end. If no tree in the result file is the same as the tree_new, write tree_new in the result file.

This way, it took 48 hours to get all 16-vertex trees based on the result file with all 15-vertex trees.

7.4.2 Running Program on a Computer with a Large Memory

If a computer has a large memory, a data structure line is used to save the trees obtained. A pointer is used to connect the cells in the line. One pointer is from head to end. The other one is from end to head. All trees which have been obtained are saved in memory. The time of reading from and writing to a file is saved.

The time of computing the length table and degree repeatedly is also saved. Because a tree is saved in a file, the length table and degree table of a tree that is already obtained are thrown away. When comparing tree_new with tree_old in the file, the length table and degree table has to be computed again.

The running time is also based on the compare order. When compared tree_new with the trees which have been obtained by head to end order, it took 40 minutes to get all 16-vertex trees based on the result file with all 15-vertex trees.

However If compared tree_new with tree_olds by end to head order, it took 20 minutes to get the all 16-vertex trees based on the result file with all 15-vertex trees. The reason is that if there is the same kind of tree in the result as tree_new, tree_old is easy to be found around the end other than the head. Most tree_news are thrown away

closer to end. For example, there are 1000 trees which is saved in the line, if a tree 950th is same as tree_new, and the tree_new is compared from 1 to 1000 order, 950 times is needed to find that there is already a same tree as tree_new. But if compared the tree_new by other order, 50 times are needed.

7.4.3 Trade-off on Memory Usage and CPU Usage

If the program to generate all 18-vertex trees uses the way saving all results in memory, the CPU time was 15 hours. However, the real used time was 35 hours, and no any other program is running at the same time. Where were the 20 hours used? It was used in reading from memory. When the size of memory usage is big enough, just 2 percent CPU is used.

For that reason to limit the size of line. When the line is full, the first in first out principle is used to write the oldest tree in the line in a file. Most of the new trees are thrown away because the same trees are found in the line. If any one tree is found which is the same as the new tree, open the file and read the trees in the file to continue to compare. If no tree is the same as the new tree, the new tree will be saved in the line. The head of the line, which is the oldest tree in the line is written to the file.

Certainly, if the size of the line is too small, the time to read and write will be too long; if the size of the line is too big, the memory access time is too long and CPU time is too small. So trade-off is needed. If the size of line is within the limits of 120000 lines, it took 16 hours to get all 18-vertex trees.

Appendix A Generation of Trees Source Code

geneTree.h

```

class setOfLength
{
public:
    int valueL;
    int numberL;
};

class Edge
{
public:
    int p1;
    int p2;
    int used; //0 not yet used; 1 used;
};
class simpleTree;

class Cell;

class Tree
{
public:
    Edge e[20];
    int leafV[20];
    int degree[20];
    int numberOfLeaves;
    int numberOfLines;
    int numberOfSetL;
    int columOfThisRow[20];
    int rowUsedFlag[20];
    int numberOfSetV;
    setOfLength setL[20];
    setOfLength setV[20];
    setOfLength setLTable[20][20];

    Tree() { numberOfSetL=0;}
    void setLengthVal(int a);
    void setLengthTable(int a, int leaf);
    void operator =(const Tree t1);
    void operator =(const simpleTree t);
    void operator +=(Tree t1);
    void degreeOfVertieces();
    void cLength();
    void cLengthTable();
    bool ifLeaf(int p);
    bool ifSameTree(Tree t);
};

```

```

class simpleTree
{
public:
    int numberOfLines;
    Edge e[20];
    void operator =(const Tree t1);
    void operator +=(Tree t1);
};

class Stack
{
private:
    int val[20];
    int sizeOfStack;
    int cur;
public:
    Stack() { cur=-1; }
    void push(int a);
    void pop();
    int curV() { return val[cur]; }
    int getCur() { return cur; }
    void setSizeOfStack(int s) { sizeOfStack=s; }
    void stackCopy(Stack s);
    void clear() { cur=-1; }
    friend cLength();
};

class Cell {
private:
    Tree tree;
    Cell* next;
    Cell* forward;
public
    Cell(Tree t);
    Cell();
    Cell* getNext();
    Cell* getForward();
    void setCell(Tree t);
};

class TreeLine {
private:
    Cell* head;
    Cell* end;
    int length;
public:
    TreeLine();
    bool isFull();
    void onLine(Tree t);

```

```

    void offLine();
    int getLineLength();
};

```

geneTree.cpp

```

#include <iostream.h>
#include <fstream.h>
#include "bt1Only.h"

#define LENGTHOFLINE 120000

ofstream myout;

void Tree::setLengthTable(int a, int vertex)
{
    if(columnOfThisRow[vertex]==0) {
        columnOfThisRow[vertex]=1;
        setLTable[vertex][columnOfThisRow[vertex]].valueL=a;
        setLTable[vertex][columnOfThisRow[vertex]].numberL=1;
    } else {
        int i=1;
        int flag=0;
        while (i<=columnOfThisRow[vertex] && flag==0) {
            if(setLTable[vertex][i].valueL == a) {
                flag=1;
                setLTable[vertex][i].numberL++;
            } else {
                i++;
            }
        }
        if (flag==0) {
            columnOfThisRow[vertex]++;
            setLTable[vertex][columnOfThisRow[vertex]].valueL=a;
            setLTable[vertex][columnOfThisRow[vertex]].numberL=1;
        }
    }
}

void Tree::cLengthTable()
{
    int curSt;
    int foundP[20];
    Stack st[20];

    for(int i=1; i<=numberOfLines+1; i++) {
        columnOfThisRow[i]=0;
        rowUsedFlag[i]=0;
        for(int j=0; j<numberOfLines; j++) { //set the pre-value
            e[j].used=0;
        }
    }
}

```

```

int usedLine=0;
for(j=1; j<=numberOfLines; j++) {
    st[j].clear();
    st[j].setSizeOfStack(numberOfLines);
}
int w=0;
int findFlag=0;
int countSt=0;
while(w<numberOfLines)
{
    if(e[w].p1 == i ) {
        countSt++;
        st[countSt].push(i);
        st[countSt].push(e[w].p2);
        e[w].used=1;
        usedLine++;
        w++;
    } else if(e[w].p2==i) {
        countSt++;
        st[countSt].push(i);
        st[countSt].push(e[w].p1);
        e[w].used=1;
        usedLine++;
        w++;
    } else {
        w++;
    }
}
curSt=countSt;
for(j=1; j<=numberOfLines+1; j++) {      //need to creat numberOfLines+1 stacks
    int c=0;
    for(int k=0; k<numberOfLines; k++) {
        if (e[k].used==0) {
            if (e[k].p1 == st[j].curV()) {
                c++;
                foundP[c]=e[k].p2;
                e[k].used=1;
                usedLine++;
            } else if (e[k].p2== st[j].curV()) {
                c++;
                foundP[c]=e[k].p1;
                e[k].used=1;
                usedLine++;
            }
        }
    }
    if (c>0) {
        for(k=1; k<=c; k++) {
            st[++curSt].stackCopy(st[j]);
            st[curSt].push(foundP[k]);
        }
    }
}

```

```

        for(int k=1; k<=numberOfLines; k++) {
            setLengthTable(st[k].getCur(),i);
        }
    }
}

```

```

void Tree::setLengthVal(int a)
{
    if(numberOfSetL==0) {
        numberOfSetL=1;
        setL[numberOfSetL].valueL=a;
        setL[numberOfSetL].numberL=1;
    } else {
        int i=1;
        int flag=0;
        while (i<=numberOfSetL && flag==0) {
            if(setL[i].valueL == a) {
                flag=1;
                setL[i].numberL++;
            } else {
                i++;
            }
        }
        if (flag==0) {
            numberOfSetL++;
            setL[numberOfSetL].valueL=a;
            setL[numberOfSetL].numberL=1;
        }
    }
}

```

```

void Tree::operator =(const Tree t1)
{
    for(int i=0; i<=t1.numberOfLines; i++) {
        e[i].p1=t1.e[i].p1;
        e[i].p2=t1.e[i].p2;
    }
    for(i=1; i<=t1.numberOfLeaves; i++) {
        leafV[i]=t1.leafV[i];
    }
    for(i=1; i<=t1.numberOfSetL; i++) {
        setL[i].numberL=t1.setL[i].numberL;
        setL[i].valueL=t1.setL[i].valueL;
    }
    for(i=1; i<=t1.numberOfSetV; i++) {
        setV[i].numberL=t1.setV[i].numberL;
        setV[i].valueL=t1.setV[i].valueL;
    }
    numberOfLeaves=t1.numberOfLeaves;
    numberOfLines=t1.numberOfLines;
    numberOfSetL=t1.numberOfSetL;
}

```

```

    numberOfSetV=t1.numberOfSetV;
}

void Tree::operator =(simpleTree t1)
{
    for(int i=0; i<=t1.numberOfLines; i++) {
        e[i].p1=t1.e[i].p1;
        e[i].p2=t1.e[i].p2;
    }
    numberOfLines=t1.numberOfLines;
    numberOfLeaves=numberOfLines+1;
}

void Tree::operator +=(Tree t1)
{
    int numVertices=t1.numberOfLines+1;
    for(int i=0; i<t1.numberOfLines; i++) {
        if(t1.e[i].p1==1)
            t1.e[i].p1=numVertices;
        else if (t1.e[i].p2==1)
            t1.e[i].p2=numVertices;
        else if(t1.e[i].p1==numVertices)
            t1.e[i].p1=1;
        else if (t1.e[i].p2==numVertices)
            t1.e[i].p2=1;

        e[i].p1=t1.e[i].p1;
        e[i].p2=t1.e[i].p2;
    }
}

void simpleTree::operator =(const Tree t1)
{
    for(int i=0; i<=t1.numberOfLines; i++) {
        e[i].p1=t1.e[i].p1;
        e[i].p2=t1.e[i].p2;
    }
    numberOfLines=t1.numberOfLines;
}

void simpleTree::operator +=(Tree t1)
{
    int numVertices=t1.numberOfLines+1;
    for(int i=0; i<=t1.numberOfLines; i++) {
        e[i].p1=t1.e[i].p1;
        e[i].p2=t1.e[i].p2;
        if(e[i].p1==1)
            e[i].p1=numVertices;
    }
}

```

```

        else if (t1.e[i].p2==1)
            e[i].p2=numVertices;
        else if(e[i].p1==numVertices)
            e[i].p1=1;
        else if (e[i].p2==numVertices)
            e[i].p2=1;
    }
    numberOfLines=t1.numberOfLines;
}

/* count every vertex edge value and then count leave number and
the same vertex edge value*/
void Tree::degreeOfVertieces()
{
    int cSetV=0;
    int leaveC=0;
    for (int i=1; i<=numberOfLines+1; i++) {
        int c=0;
        for (int j=0; j<numberOfLines; j++) {
            if(e[j].p1==i || e[j].p2==i)
                c++;
        }
        degree[i]=c;
        if (c==1) {
            leaveC++;
            leafV[leaveC]=i;
        }
        int flag=0;
        int k=1;
        while(k<=cSetV && flag==0) {
            if(setV[k].valueL==c) {
                flag=1;
                setV[k].numberL++;
            }
            k++;
        }
        if (flag==0) {
            cSetV++;
            setV[cSetV].valueL=c;
            setV[cSetV].numberL=1;
        }
    }
    numberOfLeaves=leaveC;
    numberOfSetV=cSetV;
}

void Tree::cLength()
{
    int curSt;
    int foundP[20];
    Stack st[20];

```

```

for(int i=1; i<=numberOfLines+1; i++) { //curLeaf as head of stack
    for(int i=0; i<numberOfLines; i++) {
        e[i].used=0;
    }
    for(i=1; i<=numberOfLines+1; i++) {
        st[i].clear();
        st[i].setSizeOfStack(numberOfLines);
    }
    int w=0;
    int iOfSt=0;
    while(w<numberOfLines)
    {
        if(e[w].p1 == i ) {
            iOfSt++;
            st[iOfSt].push(i);
            st[iOfSt].push(e[w].p2);
            e[w].used=1;
        } else if(e[w].p2==i) {
            iOfSt++;
            st[iOfSt].push(i);
            st[iOfSt].push(e[w].p1);
            e[w].used=1;
        } else {
            w++;
        }
    }
    curSt=iOfSt;
    int chkdLines=0;
    for(i=1; i<=numberOfLines; i++) { //need to creat numberOfLines stacks
        int exitFlag=0;
        int c=0;
        for(int j=0; j<numberOfLines; j++) {
            if (e[j].used==0) {
                if (e[j].p1 == st[i].curV()) {
                    c++;
                    foundP[c]=e[j].p2;
                    e[j].used=1;
                    chkdLines++;
                } else if (e[j].p2== st[i].curV()) {
                    c++;
                    foundP[c]=e[j].p1;
                    e[j].used=1;
                    chkdLines++;
                }
            }
        }
        if (c>0) {
            for(int k=1; k<=c; k++) {
                st[++curSt].stackCopy(st[i]);
                st[curSt].push(foundP[k]);
            }
        }
    }
}

```



```

    }
    for(i=1; i<numberOfLines; i++) {
        setLengthVal(st[i].getCur());
    }
}

bool Tree::ifLeaf(int p)
{
    for(int i=1; i<=numberOfLeaves; i++) {
        if(p==leafV[i]) {
            return 1;
        }
    }
    return 0;
}

bool Tree::ifSameTree(Tree t) {
    for(int b=0; b<=numberOfLines+1; b++) {
        t.rowUsedFlag[b]=0;
    }
    if(numberOfLeaves==t.numberOfLeaves &&
        numberOfSetV == t.numberOfSetV) {
        int f1=0;
        int f2=0;
        for (int i=1; i<=numberOfSetV; i++) {
            for (int inew=1; inew<=numberOfSetV; inew++) {
                if(setV[i].valueL==t.setV[inew].valueL
                    && setV[i].numberL==t.setV[inew].numberL) {
                    f2++;
                }
            }
        }
        if(f2==numberOfSetV) {
            int f3=0;
            for(i=1; i<=numberOfLines+1; i++) {
                int exitf=0;
                int q=1;
                while(q<=numberOfLines+1 && exitf==0) {
                    if(t.rowUsedFlag[q]==0) {
                        if(columnOfThisRow[i]==t.columnOfThisRow[q]) {
                            int f4=0;
                            for(int j=1; j<=columnOfThisRow[i]; j++) {
                                for(int k=1;
                                    k<=t.columnOfThisRow[q]; k++) {
                                    if((setLTable[i][j].valueL==
                                        t.setLTable[q][k].valueL)&&
                                        (setLTable[i][j].numberL==
                                        t.setLTable[q][k].numberL))
                                        f4++;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        if(f4==columnOfThisRow[i]){
            f3++;
            exitf=1;
            t.rowUsedFlag[q]=1;
        }
    }
    } else {
        q++;
    }
}
}
if(f3>=numberOfLines+1)
    return 1;
}
}
return 0;
}

void Stack::push(int a)
{
    if(cur<=sizeofStack) {
        cur=cur++;
        val[cur]=a;
    }
}

void Stack::pop() {
    if (cur>=0) {
        cur=cur-1;
    }
}

void Stack::stackCopy(Stack s) //copy original stack n-1 value
{
    cur=s.cur;
    sizeofStack=s.sizeOfStack;
    for(int i=0; i<=s.cur; i++) {
        val[i]=s.val[i];
    }
}

Cell::Cell() {
    next=NULL;
}

```

```

Cell::Cell(Tree t) {
    next=NULL;
    forward=NULL;
    for(int i=0; i<=t.numberOfLines; i++) {
        tree.e[i].p1=t.e[i].p1;
        tree.e[i].p2=t.e[i].p2;
    }
    for(i=1; i<=t.numberOfLeaves; i++) {
        tree.leafV[i]=t.leafV[i];
    }
    for(i=1; i<=t.numberOfSetL; i++) {
        tree.setL[i].numberL=t.setL[i].numberL;
        tree.setL[i].valueL=t.setL[i].valueL;
    }
    for(i=1; i<=t.numberOfSetV; i++) {
        tree.setV[i].numberL=t.setV[i].numberL;
        tree.setV[i].valueL=t.setV[i].valueL;
    }
    tree.numberOfLeaves=t.numberOfLeaves;
    tree.numberOfLines=t.numberOfLines;
    tree.numberOfSetL=t.numberOfSetL;
    tree.numberOfSetV=t.numberOfSetV;
    for(int j=1; j<=tree.numberOfLeaves; j++) {
        tree.cLengthTable();
    }
}

Cell* Cell::getNext() {
    return next;
}

void Cell::setCell(Tree t) {
    next=NULL;
    for(int i=0; i<=t.numberOfLines; i++) {
        tree.e[i].p1=t.e[i].p1;
        tree.e[i].p2=t.e[i].p2;
    }
    for(i=1; i<=t.numberOfLeaves; i++) {
        tree.leafV[i]=t.leafV[i];
    }
    for(i=1; i<=t.numberOfSetL; i++) {
        tree.setL[i].numberL=t.setL[i].numberL;
        tree.setL[i].valueL=t.setL[i].valueL;
    }
    for(i=1; i<=t.numberOfSetV; i++) {
        tree.setV[i].numberL=t.setV[i].numberL;
        tree.setV[i].valueL=t.setV[i].valueL;
    }
    tree.numberOfLeaves=t.numberOfLeaves;
    tree.numberOfLines=t.numberOfLines;
    tree.numberOfSetL=t.numberOfSetL;
    tree.numberOfSetV=t.numberOfSetV;
    for(int j=1; j<=tree.numberOfLeaves; j++) {

```

```

        tree.cLengthTable();
    }
}

TreeLine::TreeLine() {
    length=0;
}

bool TreeLine::isFull() {
    if(length>LENGTHOFLINE)
        return 1;
    else
        return 0;
}

void TreeLine::onLine(Tree t) {
    Cell *c;
    if(length==0) {
        c=new Cell(t);
        length++;
        head=c;
        end=c;
    } else if(!isFull()) {
        c=new Cell(t);
        c->forward=end;
        end->next=c;
        end=c;
        length++;
    } else {
        Cell *temp;
        myout<<"\n";
        for(int j=0; j<head->tree.numberOfLines; j++)
            myout<<" "<<head->tree.e[j].p1<<" "<<head->tree.e[j].p2;

        myout.flush();

        temp=head;
        head=head->next;
        head->forward=NULL;

        end->next=temp;
        temp->forward=end;
        temp->next=NULL;
        end=temp;
        end->setCell(t);
    }
}

int TreeLine::getLineLength() {
    return length;
}

```

```

void main()
{
    Tree t[35], tempTree, onlyTree;
    int totalCT, newCT, curCT;
    int sameFlag[35];
    int treePoints, cE;
    TreeLine* tLine;
    Cell* tempCell;

    cout<<"Please input the number of tree points: ";
    cin>>treePoints;

    totalCT=0;
    newCT=1;
    cE=treePoints-1;
    tLine= new TreeLine;

    char outfileName[12]="result";
    outfileName[6]=treePoints/10+'0';
    outfileName[7]=(treePoints-treePoints/10*10)%10+'0';
    outfileName[8]='\0';
    char infilename[9]="result";
    infilename[6]=(treePoints-1)/10+'0';
    infilename[7]==((treePoints-1)-(treePoints-1)/10*10)%10+'0';
    infilename[8]='\0';
    ifstream myin(infileName);
    ofstream myout(outfileName);
    int oldT=0;
    while(!myin.eof()) {
        oldT++;
        curCT=0;
        totalCT++;
        for(int i=0; i<cE-1; i++) {
            myin>>t[curCT].e[i].p1>>t[curCT].e[i].p2;
            t[curCT].e[i].used=0;
        }
        if(oldT==8) {
            int test=0;
        }
        t[curCT].numberOfLines=cE-1;
        t[curCT+treePoints-2]+=t[curCT];
        t[curCT].numberOfSetL=0;
        t[curCT].numberOfLines=cE;
        t[curCT].e[cE-1].p1=2;
        t[curCT].e[cE-1].p2=treePoints;
        totalCT++;
        t[curCT+treePoints-2].numberOfSetL=0;
        t[curCT+treePoints-2].numberOfLines=cE;
        t[curCT+treePoints-2].e[cE-1].p1=2;
        t[curCT+treePoints-2].e[cE-1].p2=treePoints;
    }
}

```

```

for(i=3; i<treePoints; i++) {
    totalCT++;
    curCT++;
    t[curCT].numberOfSetL=0;
    t[curCT].numberOfLines=cE;
    t[curCT].e[cE-1].p1=i;
    t[curCT].e[cE-1].p2=treePoints;
    for(int j=0; j<cE-1; j++) {
        t[curCT].e[j].p1=t[0].e[j].p1;
        t[curCT].e[j].p2=t[0].e[j].p2;
    }
    totalCT++;
    t[curCT+treePoints-2].numberOfSetL=0;
    t[curCT+treePoints-2].numberOfLines=cE;
    t[curCT+treePoints-2].e[cE-1].p1=i;
    t[curCT+treePoints-2].e[cE-1].p2=treePoints;
    for(j=0; j<cE-1; j++) {
        t[curCT+treePoints-2].e[j].p1=t[treePoints-2].e[j].p1;
        t[curCT+treePoints-2].e[j].p2=t[treePoints-2].e[j].p2;
    }
}
for(i=0; i<=curCT+treePoints-2; i++) {
    t[i].degreeOfVertieces();
    t[i].cLengthTable();
}
if(tLine->getLineLength()==0) {
/*      for(int i=0; i<cE; i++) {
            myout<<" "<<t[0].e[i].p1<<" "<<t[0].e[i].p2;
        }
    myout.flush();*/
    tLine->onLine(t[0]);
}

for(i=0; i<=curCT+treePoints-2; i++)
    sameFlag[i]=0;
int totalSameFlag=0;
for(i=1; i<=curCT+treePoints-2; i++) {
    for(int k=0; k<i; k++){
        if(i==7) {
            int test=0;
        }
        if(sameFlag[i]==0) {
            sameFlag[i]=t[i].ifSameTree(t[k]);
            totalSameFlag=totalSameFlag+sameFlag[i];
        }
    }
}

tempCell=tLine->end;
while(tempCell!=NULL && totalSameFlag!=curCT+treePoints-1) {
    for(int j=0; j<=curCT+treePoints-2; j++){
        if(sameFlag[j]==0)
            if(t[j].ifSameTree(tempCell->tree)) {

```

```

        totalSameFlag++;
        sameFlag[j]=1;
    }
}
tempCell=tempCell->forward;
}

if(totalSameFlag<curCT+treePoints-1 && tLine->isFull()) {
    ifstream mycheck(outfileName);
    while(!mycheck.eof() && totalSameFlag<curCT+treePoints-1) {
        for(int j=0; j<cE; j++) {
            mycheck>>onlyTree.e[j].p1>>onlyTree.e[j].p2;
            onlyTree.e[j].used=0;
        }
        onlyTree.numberOfLines=cE;
        onlyTree.degreeOfVertieces();
        onlyTree.cLengthTable();
        for(j=0;j<=curCT+treePoints-2; j++){
            if(sameFlag[j]==0) {
                if(t[j].ifSameTree(onlyTree)) {
                    sameFlag[j]=1;
                    totalSameFlag++;
                }
            }
        }
    }
    mycheck.close();
}

cout<<newCT<<" * here "<<totalCT<<" * "<<oldT<<"\n";
cout.flush();
for(i=0; i<=curCT+treePoints-2; i++) {
    if(sameFlag[i]==0) {
        newCT++;
        tLine->onLine(t[i]);
    }
}

tempCell=tLine->head;
while(tempCell!=NULL) {
    myout<<"\n";
    for(int j=0; j<cE; j++)
        myout<<" "<<tempCell->tree.e[j].p1<<"
        "<<tempCell->tree.e[j].p2;
    myout.flush();
    tempCell=tempCell->next;
    cout.flush();
}

cout<<"total Tree: "<<totalCT<<"\n";
cout<<"new only tree: "<<newCT<<"\n";
cout<<"the tLine: "<<tLine->getLineLength()<<"\n";

```

```
char stop;  
cin>>stop;  
}
```


Appendix B Search for Super Edge-Magic Labelings Source Code

```

// test a tree with 16 vertices
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <iostream.h>
#include <fstream.h>

int* v;
int* e;
int s[15][2]; // define the tree
int eSize, vSize;
int depth;
int count;
int display, scale;
int exitflag;

FILE* fp;
ofstream myout;

int equal_edge(int n)
{
    int i;
    for(i=0; i<n; i++)
    {
        if(e[n]==e[i])
            return 1;
    }
    return 0;
}

int magic()
{
    int i, min, max;
    min=1000000;
    max=0;
    for(i=0; i<eSize; i++)
    {
        if(e[i]<min) min=e[i];
        if(e[i]>max) max=e[i];
    }
    if((max-min)==eSize-1)
        return 1;
    else
        return 0;
}

void f(const int* in, int cnt)
{

```

```

int i,j,k,t;
int *new_in;

if( cnt > 0 && exitflag==0)
{
    new_in = (int*)malloc((cnt-1)*sizeof(int));
    t=vSize-cnt; // decide the t'th vertex
    for(i=0; i<cnt; i++)
    {
        v[t] = in[i]; // update vertices

        // update edges
        if(t>0)
        {
            e[t-1]=v[s[t-1][0]]+v[s[t-1][1]];
        }

        // use symmetry
        if( t==2 || t==3 || t==6 || t==7 ||
            t==9 || t==10 || t==12 || t==13 || t==15 )
        {
            if( v[t] < v[t-1] )
                continue;
        }

        // use indexability
        if( equal_edge(t-1) )
            continue;
        for(j=0,k=0;j<cnt;j++)
        { // set new_in = set in - {in[i]}
            if( in[j] != in[i] )
            {
                new_in[k]=in[j];
                k++;
            }
        }
        f(new_in, cnt-1);
    }
    free(new_in);
}
else if(exitflag==0)
{
    scale++;
    if(scale==100000)
    {
        scale=0;
        display++;
        cout<<display<<" ";
        cout.flush();
    }
}

```

```

    }
    if(magic())
    {
        for(i=0; i<vSize;i++)
            myout<<v[i]<<" ";
        myout<<"E = ";
        for (i=0;i<eSize;i++) myout<<e[i]<<" ";
        myout<<"\n";

        //
        for(i=0; i<vSize;i++)
            cout<<v[i]<<" ";
        cout<<"E = ";
        for (i=0;i<eSize;i++) cout<<e[i]<<" ";
        cout<<"\n";
        //
        exitflag=1;
        count++;
        cout<<"***"<<count<<" ";
        cout.flush();
        myout.flush();
    }
}

void main()
{
    int* input;
    int i;

    vSize=16;
    eSize=vSize-1;
    v=(int*)malloc(vSize*sizeof(int));
    e=(int*)malloc(eSize*sizeof(int));

    input=(int*)malloc(vSize*sizeof(int));
    for(i=0;i<vSize;i++)
        input[i]=i+1;
    count=0;
    display=0;
    scale=0;
    ofstream myout("magic16");

    ifstream myin("result16");
    while(!myin.eof()) {
        for (int i=0; i<15; i++) {
            myin>>s[i][0]>>s[i][1];
            s[i][0]--;
            s[i][1]--;
            cout<<s[i][0]<<" & "<<s[i][1]<<"\n";
        }
        cout.flush();
    }
    // define_tree();
}

```

```
        exitflag=0;
        f(input, vSize);
    }
//    fprintf(fp, "count=%d\n", count);
    free(input);
    free(v);
    free(e);
    fclose(fp);
}
```

References

- Acharya, B.D., & Hedge, S.M. (1991). Strongly indexable graphs. *Discrete Math*, 93, 123-129.
- Arumugam, S., & Germina, K.A. (1996). On indexable graphs. *Discrete math*, 161, 285-289.
- Beyer, T., & Hedetniemi, S.M. (1980). Constant time generation of rooted trees. *SIAM Journal Computing*, 9, 706-712.
- Chen, Z. On Super edge-magic graphs. Manuscript.
- Enomoto, H., Llado, A.S., Nakamigawa, T., & Ringel, G. (1998). Super edge-magic graphs. *SUT Journal of Mathematics*, 34(2), 105-109.
- Gallian, J. (1998). A dynamic survey of graph labeling. *Electronic Journal of Combinatorics*, 5, DS6.
- Hartsfield, N., & Ringel, G. (1989). Supermagic and antimagic graphs. *Journal of Recreational Mathematics*, 21, 116-124.
- Kotzig, A., & Rosa, A. (1970). Magic valuations of finite graphs. *Canad Math Bull*, 13, 451-461.
- Lee, S.M., Seah, E., & Tan, S.K. (1992). On edge-magic graphs. *Congressus Numerantium*, 86, 179-191.
- Lee, S.M., Seah, E., & Tong S.M. (1999). On the edge-graceful and edge-magic total graphs conjecture. *Congressus Numerantium*, 141, 37-48.
- Mckay, B.D.(2002). Nauty. Retrieved September 11, 2001, from Austrilian National University, Department of Computer Science Web site:
<http://cs.anu.edu.au/people/bdm/nauty/>
- Ruskey, F. (2002). Information on free trees. Retrieved September 10, 2001, from University of Victoria, Department of Computer Science Web site:
<http://www.theory.csc.uvic.ca/~cos/inf/tree/FreeTrees.html>
- Sedlacek, J. (1963). Problem 27. *Theory of graphs and its applications*, Smolenice: 163-167. (publ. House Czechoslovak Acad. Sci., Prague, 1964).
- Stewart, B.M.(1996). Magic Graphs. *Canad Journal Math*, 18, 1031-29.

- Tsuchiya, M., & Yokomura, K. (1999). On some families of edge-magic graphs. *Combinatorics, graph theory and algorithms*, 2, 817-822.
- Wallis, W.D., Baskoro, T., Miller, M., & Slamin. Edge-magic total labelings. *Australasian Journal of Combinatorics*.
- Wright, R.A., Richmond, L.B., Odlyzko, A., & McKay, B.D. (1986). Constant time generation of free trees. *SIAM Journal Computing*, 15, 540-548.