

PARTITIONING THE LABELED SPANNING TREES OF AN  
ARBITRARY GRAPH INTO ISOMORPHISM CLASSES

by

Austin Mohr

Bachelor of Science in Mathematics, Southern Illinois University, 2007

A Research Paper Submitted in Partial Fulfillment  
of the Requirements for the Master of Science Degree

Department of Mathematics  
in the Graduate School  
Southern Illinois University Carbondale  
July, 2008

## AN ABSTRACT OF THE RESEARCH PAPER OF

Austin Mohr, for the Master of Science degree in Mathematics, presented on July 01, 2008, at Southern Illinois University Carbondale.

TITLE: PARTITIONING THE LABELED SPANNING TREES OF AN  
ARBITRARY GRAPH INTO ISOMORPHISM CLASSES

MAJOR PROFESSOR: Dr. Thomas D. Porter

An isomorphism between labeled graphs  $G$  and  $H$  is a mapping  $f$  from the vertices of  $G$  to the vertices of  $H$  such that  $uv$  is an edge of  $G$  if and only if  $f(u)f(v)$  is an edge of  $H$ . We consider the problem of partitioning all labeled spanning trees of an arbitrary graph into equivalence classes under isomorphism. To accomplish this, we employ the algorithm presented in [5] to generate the trees. As the trees are generated, we apply the test for tree isomorphism found in [1] to partition the trees appropriately. When the program terminates, a graphical representative of each isomorphism class is presented, along with the number of trees contained in the class. An implementation of these algorithms, as well as sample output, is available at [6]. Additionally, we present an approach to the problem of finding a closed formula for  $I(K_{s,t})$ , the number of nonisomorphic spanning trees of the complete bipartite graph, based on the aforementioned tree isomorphism test. Using this method, we obtain formulas for  $I(K_{2,t})$  and  $I(K_{3,t})$  involving the partition numbers.

## DEDICATION

To my parents, Dennis and Cindy

To R. Mark Wall

To Ken Fong

To Dr. Thomas D. Porter

To my wife, Mary

## TABLE OF CONTENTS

Abstract . . . . .	i
Dedication . . . . .	ii
List of Figures . . . . .	iv
Introduction . . . . .	1
1 Preliminaries . . . . .	3
2 Generating the Spanning Trees of an Arbitrary Labeled Graph . . . . .	7
2.1 Definitions and Examples . . . . .	7
2.2 Finding the Children of a Spanning Tree . . . . .	9
2.3 Generating all Labeled Spanning Trees of an Arbitrary Graph . . . . .	13
3 Determining Isomorphism Between Trees . . . . .	16
3.1 Rooted Tree Isomorphism . . . . .	16
3.2 General Tree Isomorphism . . . . .	21
4 Partitioning Labeled Spanning Trees into Isomorphism Classes . . . . .	25
5 Some Results . . . . .	27
6 Developing a Closed Formula for $I(K_{s,t})$ . . . . .	31
6.1 Preliminaries . . . . .	31
6.2 A Method for Computing $I(K_{s,t})$ . . . . .	33
References . . . . .	46
Vita . . . . .	47

## LIST OF FIGURES

1.1	A graph and two spanning trees . . . . .	4
1.2	A pair of isomorphic graphs . . . . .	4
2.1	A “tree of trees” for $K_{2,3}$ . . . . .	9
3.1	Two isomorphic rooted trees . . . . .	16
3.2	Example run of the rooted tree isomorphism test . . . . .	20
5.1	The spanning trees of $K_6$ partitioned under isomorphism . . . . .	30
6.1	(Top) Trees rooted at their centers; (Bottom) Trees not rooted at their centers . . . . .	35
6.2	Configuration of $K_{2,t}$ with center in 2-set . . . . .	36
6.3	Configuration of $K_{2,t}$ with center in $t$ -set . . . . .	36
6.4	Configuration of $K_{2,t}$ where center is an edge . . . . .	37
6.5	Configuration of $K_{3,t}$ with center in 3-set (a) . . . . .	37
6.6	Configuration of $K_{3,t}$ with center in 3-set (b) . . . . .	38
6.7	Configuration of $K_{3,t}$ with center in $t$ -set . . . . .	38
6.8	Configuration of $K_{3,t}$ where center is an edge (a) . . . . .	39
6.9	Configuration of $K_{3,t}$ where center is an edge (b) . . . . .	39

## INTRODUCTION

This paper examines the algorithms involved in partitioning all labeled spanning trees of an arbitrary graph into equivalence classes under isomorphism. Two primary algorithms are employed. The first (found in [5]) generates the labeled spanning trees of the input graph, while the second (found in [1]) examines each tree as it is generated to determine the appropriate isomorphism class. As a companion to the paper, a Java implementation of the overall procedure is available at [6]. This program produces a graphical representation of each isomorphism class, as well as the size of the classes (i.e. the number of labeled trees belonging to a given class).

Chapter 1 introduces the prerequisite material needed to understand the problem, as well as establishes the notation that will be used throughout the paper.

Chapter 2 covers the spanning tree generation algorithm. The assumptions made by the algorithm are proven, and a pseudocode description of the algorithm is given.

Chapter 3 covers the test for isomorphism between trees. The algorithm is given first as a test specifically for rooted trees and is then generalized to accept any two trees as input. A pseudocode description of the algorithm is given.

Chapter 4 covers the complete algorithm by making use of the sub-procedures established in the previous two chapters. A pseudocode description of the complete algorithm is given.

Chapter 5 presents results on selected graphs, including an example of the image generating capabilities of the program.

Chapter 6 introduces the problem of finding a closed formula for  $I(K_{s,t})$ , the number of nonisomorphic spanning trees of the complete bipartite graph. An approach to this problem (based on the tree isomorphism test) is presented. This approach is employed to derive closed formulas for  $I(K_{2,t})$  and  $I(K_{3,t})$ .

# CHAPTER 1

## PRELIMINARIES

We use the standard definitions and notation found in [8].

**Definition.** A **graph**  $G$  is a triple consisting of a **vertex set**  $V(G)$ , an **edge set**  $E(G)$ , and a relation that associates with each edge two vertices (not necessarily distinct) called its **endpoints**. A graph is said to be **labeled** if its vertices are given unique identifiers (usually the integers  $1, 2, \dots, |V(G)|$ ). Otherwise, the graph is **unlabeled**.

**Remark.** We will henceforth assume that the two vertices associated with each edge are, in fact, distinct. When such a condition holds, the graph is said to be **simple**.

**Definition.** A **tree**  $T$  is a graph in which, for vertices  $u$  and  $v$  of  $T$ , there is exactly one  $uv$ -path.  $T$  is a **spanning tree** of a graph  $G$  if and only if  $V(T) = V(G)$ .

**Example 1.0.1.** The figure at top is a drawing of a graph. The two figures below it are spanning trees of the graph.

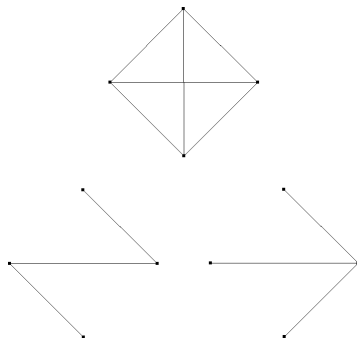


Figure 1.1. A graph and two spanning trees

**Definition.** An **isomorphism** from a graph  $G$  to a graph  $H$  is a bijection  $f : V(G) \rightarrow V(H)$  such that  $uv \in E(G)$  if and only if  $f(u)f(v) \in E(H)$ . We say  $G$  is **isomorphic to  $H$** , written  $G \cong H$ , if there is an isomorphism from  $G$  to  $H$ .

**Example 1.0.2.** The two graphs below are isomorphic under the mapping

$f : \{1, 2, 3, 4\} \rightarrow \{1, 2, 3, 4\}$  where

$$f(1) = 1, f(2) = 4, f(3) = 3, f(4) = 2$$

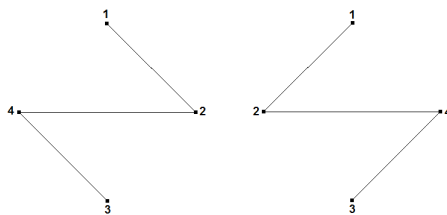


Figure 1.2. A pair of isomorphic graphs

**Definition.** A **relation** on a set  $S$  is a collection of ordered pairs from  $S$ . An **equivalence relation** is a relation that is reflexive, symmetric, and transitive.

An equivalence relation partitions  $S$  into **equivalence classes** with two elements satisfying the relation if and only if they lie in the same class.

**Proposition 1.0.1.** *The isomorphism relation, consisting of the set of ordered pairs  $(G, H)$  such that  $G \cong H$ , is an equivalence relation.*

*Proof.* We show directly that the isomorphism relation is reflexive, symmetric, and transitive.

*Reflexive:* The identity permutation on  $V(G)$  shows  $G \cong G$ .

*Symmetric:* Suppose  $G \cong H$  with isomorphism  $f$ . It follows:

$$uv \in V(G) \Leftrightarrow f(u)f(v) \in V(H)$$

$$f^{-1}(u)f^{-1}(v) \in V(G) \Leftrightarrow uv \in V(H)$$

Hence,  $H \cong G$  with isomorphism  $f^{-1}$ .

*Transitive:* Suppose  $F \cong G$  with isomorphism  $f$  and  $G \cong H$  with isomorphism  $g$ .

It follows:

$$(uv \in E(F) \Leftrightarrow f(u)f(v) \in E(G)) \wedge (xy \in E(G) \Leftrightarrow g(x)g(y) \in E(H))$$

Now, the above holds for any  $x, y \in V(G)$ . In particular, it holds for  $x = f(u)$  and  $y = f(v)$ , which exist since  $f$  is surjective. We then have,

$$(uv \in E(F) \Leftrightarrow f(u)f(v) \in E(G)) \wedge (f(u)f(v) \in E(G) \Leftrightarrow g(f(u))g(f(v)) \in E(H))$$

$$uv \in E(F) \Leftrightarrow g(f(u))g(f(v)) \in E(H)$$

Hence,  $E \cong H$  with isomorphism  $g \circ f$ . □

We can now state the primary concern of the paper quite succinctly. Given an arbitrary labeled graph, we wish to partition all its spanning trees into equivalence classes under isomorphism. To accomplish this, we need an algorithm to generate all the spanning trees of an arbitrary labeled graph (Chapter 2) and an algorithm to decide if two trees are isomorphic (Chapter 3).

**CHAPTER 2**

**GENERATING THE SPANNING TREES OF AN ARBITRARY  
LABELED GRAPH**

The content of this chapter comes from [5] unless otherwise specified.

**2.1 DEFINITIONS AND EXAMPLES**

Consider the following construction.

**Definition.** Given an arbitrary graph  $G$  with  $m$  edges and  $n$  vertices, assign integer labels  $1, 2, \dots, m$  to the edges such that the subgraph induced by the set of edges  $\{1, 2, \dots, (n - 1)\}$  is a spanning tree of  $G$ . We call this special tree  $T^*$  and refer to the integer label of an edge as its **index**.

We now introduce some terminology regarding the labels.

**Definition.** Given a subgraph  $H \subseteq G$ , the edge of  $H$  with smallest index is called the **top edge** of  $H$ , denoted  $top(H)$ . Similarly, the edge of  $H$  with largest index is called the **bottom edge** of  $H$ , denoted  $btm(H)$ .

We also require the following standard definition from [8].

**Definition.** A **cut-edge** of a graph  $G$  is an edge whose deletion increases the number of components of  $G$ . If  $e$  is a cut-edge, we denote the set of edges connecting the components of  $G$  formed by removing  $e$  by  $Cut(G, e)$ .

Now, consider the following function, which takes a spanning tree of  $G$  as input and outputs another spanning tree of  $G$ .

**Definition.** For any spanning tree  $T$  different than  $T^*$ , let  $\phi(T)$  denote the spanning tree obtained by removing  $f = btm(T)$  and adding  $g = top(Cut(T, f))$ . Additionally, we define  $\phi(T^*)$  to be  $T^*$ .

**Remark.** For the remainder of this chapter, we will abuse notation slightly and treat a graph as a set of edges. Furthermore, when dealing with a single edge, we will omit the set braces. Hence, the above definition would be written  $\phi(T) = (T \setminus f) \cup g$

Observe that, since the removal of any edge of a tree  $T$  disconnects the tree (see [8]),  $\phi(T)$  is indeed a tree. Now, it is easy to construct trees  $T_1 \neq T_2$  such that  $\phi(T_1) = \phi(T_2)$  (i.e.  $\phi$  is a many-to-one function). Furthermore, applying  $\phi$  recursively to any tree will eventually produce  $T^*$ , as shown by the following proposition.

**Proposition 2.1.1.** *Let  $T \neq T^*$  be a spanning tree with  $\phi(T) = (T \setminus f) \cup g$ . Then,  $g \in T^* \not\cong f$ .*

*Proof.* The definition of  $\phi$  implies that the edge  $f = btm(T)$  and  $g = top(Cut(T, f))$ . By the definition of  $T^*$ , it is clear that  $f \notin T^*$ . Since  $T^*$  is a spanning tree,  $Cut(T^*, f) \cap T^* \neq \emptyset$  and the index of  $g$  is less than or equal to the index of  $btm(T^*)$ . Then, by the definition of  $T^*$ ,  $g \in T^*$ . □

Based on this, we can construct a “tree of trees”. That is, we can form a tree (in the data structure sense) with each node containing a spanning tree of the original graph. In this structure, a child  $T'$  is related to its parent  $T$  by  $\phi(T') = T$ . The root of the structure is, of course,  $T^*$ .

**Example 2.1.1.** The following is a “tree of trees” for a particular labeling of the edges of  $K_{2,3}$ , the complete bipartite graph with partite sets of size 2 and 3. The labeling of the edges (shown at the bottom) is arbitrary so long as the graph induced by the edge-subset  $\{1, 2, 3, 4\}$  (i.e.  $T^*$ ) is indeed a spanning tree of  $K_{2,3}$ .

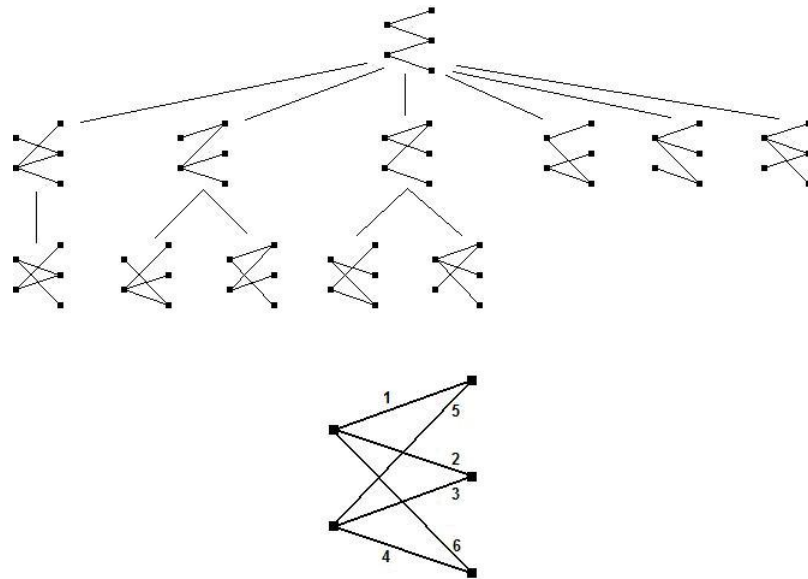


Figure 2.1. A “tree of trees” for  $K_{2,3}$

A natural question arises. Since the root of any such “tree of trees” is  $T^*$ , is there a way to begin at the root and work down the structure? More formally, given a tree  $T$  (the parent), can we find all trees  $T'$  such that  $\phi(T') = T$  (the children)?

## 2.2 FINDING THE CHILDREN OF A SPANNING TREE

Before discussing the algorithm to generate the children of a spanning tree, we require some additional definitions.

**Definition.** Given a spanning tree  $T$  of  $G$ , we say that an edge  $e$  is a **pivot edge** of  $T$  if there exists a child  $T'$  of  $T$  such that  $T' \setminus T = e$ .

**Definition.** Given a spanning tree  $T$ , denote by  $H(T)$  the edge-subset  $\{e \in T \mid e = \text{top}(\text{Cut}(T, e))\}$ .

**Definition.** Let  $T$  be a spanning tree of  $G$  and let  $e \in (G \setminus T)$ . Denote by  $\text{Cycle}(T, e)$  the edge-subset containing the edges of the unique cycle in  $T \cup e$ .

The following proposition allows us to find all the children of a given tree.

**Proposition 2.2.1.** *Let  $T$  be a spanning tree of  $G$  and  $f$  be a pivot edge of  $T$ . The edge-subset  $T' = (T \setminus g) \cup f$  is a child spanning tree of  $T$  if and only if  $g \in \text{Cycle}(T, f) \cap H(T)$ .*

*Proof.* ( $\Rightarrow$ ) Let the edge-subset  $T'$  be a child spanning tree of  $T$ . Suppose, to the contrary, that  $g \notin \text{Cycle}(T, f) \cap H(T)$ . Then, either  $g \notin \text{Cycle}(T, f)$  or  $g \notin H(T)$  (or both). If  $g \notin \text{Cycle}(T, f)$ , then  $T'$  contains a cycle, and so is not a spanning tree. If  $g \notin H(T)$ , then  $T'$  is not a child of  $T$  (by definition of  $\phi$ ). In either case, we arrive at a contradiction. Hence,  $g \in \text{Cycle}(T, f) \cap H(T)$ .

( $\Leftarrow$ ) Let  $g \in (\text{Cycle}(T, f) \cap H(T))$ . Since  $g \in \text{Cycle}(T, f)$ ,  $T'$  is a spanning tree and  $\text{Cut}(T, g) = \text{Cut}(T', f)$ . Since  $g \in H(T)$ ,  $g = \text{top}(\text{Cut}(T, g)) = \text{top}(\text{Cut}(T', f))$ , which implies that  $T$  is the parent of  $T'$ .  $\square$

With this observation, we can propose an algorithm to find the children of a spanning tree  $T$ . Given a pivot edge  $f$  of  $T$ , construct the edge-subset  $\text{Cycle}(T, f) \cap H(T)$ . For each edge  $g \in \text{Cycle}(T, f) \cap H(T)$ , generate the child spanning tree  $(T \setminus g) \cup f$ . Repeat this procedure for every pivot edge of  $T$  to generate all its child spanning trees.

Before expanding on this algorithm, we present a useful characterization of pivot edges.

**Proposition 2.2.2.** *Let  $T$  be a spanning tree of  $G$ . An edge  $e \in (G \setminus T)$  is a pivot edge of  $T$  if and only if*

1. *the index of  $e$  is strictly greater than the index of  $btm(T)$*
2.  *$e$  joins a pair of distinct components of the subgraph  $T \setminus H(T)$*

*Proof.* ( $\Rightarrow$ ) Suppose  $e$  is a pivot edge of  $T$ . Then, there exists child  $T'$  of  $T$  such that  $T' \setminus T = e$ . Let  $f = btm(T')$  and  $g = top(Cut(T', f))$ . Now, since  $T'$  is a child of  $T$ ,  $\phi(T') = (T' \setminus f) \cup g = T$ . Hence,  $e = f$ , which has index strictly less than the index of  $btm(T)$ . Furthermore, since  $Cut(T', f) = Cut(T, g)$ ,  $g = top(Cut(T, g))$ , and so  $g \in H(T)$ . Hence, the pivot edge  $e$  joins two distinct components of  $T \setminus H(T)$ .

( $\Leftarrow$ ) Suppose an edge  $f$  satisfies conditions (1) and (2). Observe that  $Cycle(T, f) \cap H(T) \neq \emptyset$ . Let  $g \in Cycle(T, f) \cap H(T)$ . Clearly,  $T' = (T \setminus g) \cup f$  is a spanning tree of  $G$ . From condition (1),  $f = btm(T')$ . Since  $g \in H(T)$  and  $Cut(T, g) = Cut(T', f)$ ,  $g = top(Cut(T, g)) = top(Cut(T', f))$ . Thus,  $T$  is the parent of  $T'$ , and so  $f$  is a pivot edge of  $T$ . □

So, to find all the pivot edges of a tree  $T$ , we first construct  $T \setminus H(T)$  (we will discuss the construction of  $H(T)$  in greater detail in the next section). Then, we loop through each edge  $e \in G$  such that  $index(e) > index(btm(T))$ . If  $e$  connects two distinct components of  $T \setminus H(T)$ , we accept  $e$  as a pivot edge of  $T$ .

**Remark.** Let  $G$  be a graph on  $n$  vertices. A result found in [7] can be applied to find a specific labeling of the edges of  $G$  such that, if  $2n - 3$  consecutive edges are not pivot edges, we are assured that there are no more pivot edges to be found. Finding this labeling as preprocessing may result in a more efficient implementation in practice (particularly in dense graphs). We did not make use of this labeling in our program, however, and so the details are omitted.

Finally, we present an algorithm to find all the children of a tree, making use of the characterization of pivot edges presented above. Note that we let  $e_i$  denote the edge with index  $i$ .

---

**Algorithm 1** findChildren( $T$ )

---

**Input:** A spanning tree  $T$ **Output:** All child spanning trees of  $T$ Construct  $H := H(T)$ Construct  $D := T \setminus H(T)$ **for**  $i := \text{btm}(T) + 1$  to  $\text{btm}(G)$  **do**  **if**  $e_i$  connects two distinct components of  $D$  **then**    Construct  $I := \text{Cycle}(T, e_i) \cap H$     **for all** edges  $g \in I$  **do**      Output child spanning tree  $(T \setminus g) \cup e_i$     **end for**  **end if****end for**

---

### 2.3 GENERATING ALL LABELED SPANNING TREES OF AN ARBITRARY GRAPH

Expanding on the idea of the findChildren algorithm, we can generate all the labeled spanning trees of an arbitrary graph via recursion. Recall first, that for each child  $T'$  of  $T$ , we will need to construct  $H(T')$ . The following proposition provides an efficient way to generate  $H(T')$  given  $H(T)$ .

**Proposition 2.3.1.** *Let  $T$  be a spanning tree of  $G$  and  $T' = (T \setminus g) \cup f$  be a child of  $T$ . Then,  $H(T') = H(T) \setminus \{e' \in \text{Cycle}(T, f) \cap H(T) \mid \text{index}(e') \geq \text{index}(g)\}$ .*

*Proof.* Clearly,  $T' \cap T^* = (T \cap T^*) \setminus g$ .

Case 1: Let  $e'$  be an edge in  $(T' \cap T^*) \setminus \text{Cycle}(T, f)$ . Then,  $\text{Cut}(T, e') = \text{Cut}(T', e')$ , and so  $e' \in H(T')$  if and only if  $e' \in H(T)$ .

Case 2: Let  $e'$  be an edge in  $(T' \cap T^*) \cap \text{Cycle}(T, f)$ . Then, it is clear that  $\text{Cut}(T', e') = \text{Cut}(T, e') \triangle \text{Cut}(T, g)$ , where  $\triangle$  denotes the symmetric difference.

Case 2.1: Suppose that  $e' \notin H(T)$ . Let  $e'' = \text{top}(\text{Cut}(T, e'))$ . When  $e'' \in \text{Cut}(T, g)$ ,

then  $index(g) < index(e'') < index(e')$ ,  $g \in Cut(T', e')$ , and so  $e' \notin H(T')$ .

If  $e'' \notin Cut(T, g)$ , then  $index(e'') < index(e')$ ,  $e'' \in Cut(T', e')$ , and we have

$e' \notin H(T')$ . Therefore, if  $e' \in (T' \cap T^*) \cap Cycle(T, f)$  and  $e' \notin H(T)$ , then  $e' \notin H(T')$ .

Case 2.2: Consider the case that  $e' \in H(T)$ . Clearly,  $e' = top(Cut(T, e'))$  and

$g = top(Cut(T, g))$ . This implies that  $top(Cut(T', e'))$  is either  $e'$  or  $g$ . Thus,

$e' = top(Cut(T', e'))$  if and only if  $index(e') < index(g)$ .

Summarizing the above, we obtain the required result.  $\square$

We now present an algorithm that, given a tree  $T$  and the edge-subset  $H(T)$ , outputs all trees  $T'$  such that  $\phi^k(T') = T$  for some value of  $k$  (i.e. all the descendants of  $T$ ).

---

**Algorithm 2** generateDescendants( $T, H$ )

---

**Input:** A spanning tree  $T$  and the edge-subset  $H = H(T)$

**Output:** All descendant spanning trees of  $T$

Output  $T$

Construct  $D := T \setminus H$

**for**  $i := btm(T) + 1$  to  $btm(G)$  **do**

**if**  $e_i$  connects two distinct components of  $D$  **then**

    Construct  $I := Cycle(T, e_i) \cap H$

**for all** edges  $g \in I$  **do**

      Construct  $T' := (T \setminus g) \cup e_i$

      Construct  $H' := H \setminus \{e \in I \mid index(e) \geq index(g)\}$

      Call generateDescendants( $T', H'$ )

**end for**

**end if**

**end for**

---

Now, recall that, for a graph  $G$ , the root of any “tree of trees” is  $T^*$ . Furthermore, observe that  $H(T^*) = T^*$ , since, by definition of  $T^*$ ,  $e = top(Cut(T^*, e))$  for all  $e \in T^*$ . Hence, the call generateDescendants( $T^*, T^*$ ) will generate all descendant

spanning trees of  $T^*$ , which is every labeled spanning tree of  $G$ .

**Remark.** Let  $G$  be a graph on  $n$  vertices and  $m$  edges and let  $\tau$  denote the number of labeled spanning trees of  $G$ . Then, to output all labeled spanning trees of  $G$  explicitly via `generateDescendants` requires  $O(n + m + n\tau)$  time, which has been shown to be optimal in [4].

Before concluding this chapter, we present two corollaries to Proposition 2.2.2 that, while not used by the algorithm in [5], may serve to reduce runtime in practice.

**Corollary 2.3.2.** *Let  $T$  be a tree. If  $H(T) = \emptyset$ , then  $T$  has no children.*

*Proof.* If  $H(T) = \emptyset$ , then  $T \setminus H(T) = T$ , which is connected. Hence, no edge can join two distinct components of  $T \setminus H(T)$ . Therefore,  $T$  has no pivot edges, and so no children. □

**Corollary 2.3.3.** *Let  $T$  be a tree of the graph  $G$ . If  $btm(G)$  is an edge of  $T$ , then  $T$  has no children.*

*Proof.* When considering a potential pivot edge  $f$ , we require that  $index(f) > index(btm(T))$ . If  $btm(G)$  is an edge of  $T$ ,  $btm(T) = btm(G)$ . Hence, there is no edge of  $G$  with index greater than  $index(btm(T))$ . Therefore,  $T$  has no pivot edges, and so no children. □

## CHAPTER 3

### DETERMINING ISOMORPHISM BETWEEN TREES

#### 3.1 ROOTED TREE ISOMORPHISM

Before solving the general problem of determining whether any two trees on  $n$  vertices are isomorphic, we first solve the simpler problem of determining whether two *rooted* trees on  $n$  vertices are isomorphic. We first introduce some definitions found in [8].

**Definition.** A **rooted tree** is a tree with one vertex  $r$  chosen as the **root**. For each vertex  $v$ , let  $P_r(v)$  be the unique path from  $r$  to  $v$ . The **descendants** of  $v$  are the vertices  $u$  such that  $P_r(u)$  contains  $v$ . If  $T$  is a rooted tree, the **subtree of  $T$  rooted at  $v$**  is the subtree rooted at  $v$  induced by the descendants of  $v$ . The vertex  $v$  is said to be at **depth  $i$**  if  $P_r(v)$  is of length  $i$ .

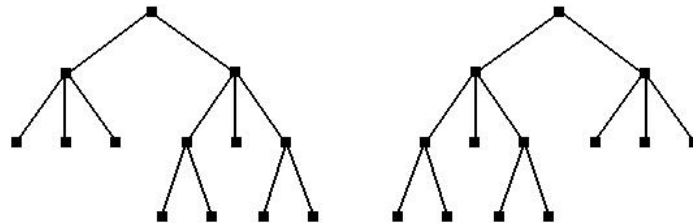


Figure 3.1. Two isomorphic rooted trees

The following proposition will be the basis of our algorithm.

**Proposition 3.1.1.** *Given two rooted trees  $T_1$  and  $T_2$  on  $n$  vertices, a mapping  $f : V(T_1) \rightarrow V(T_2)$  is an isomorphism if and only if for every vertex  $v \in V(T_1)$ , the subtree of  $T_1$  rooted at  $v$  is isomorphic to the subtree of  $T_2$  rooted at  $f(v)$ .*

*Proof.* ( $\Rightarrow$ ) Let  $f : V(T_1) \rightarrow V(T_2)$  be an isomorphism. Suppose, to the contrary, that there exists a vertex  $v \in V(T_1)$  such that the subtree  $T_1$  rooted at  $v$  (call it  $S_1$ ) is not isomorphic to the subtree of  $T_2$  rooted at  $f(v)$  (call it  $S_2$ ). Then, either there exists some edge  $uw \in E(S_1) \subseteq E(T_1)$  such that  $f(u)f(w) \notin E(S_2) \subseteq E(T_2)$  or there exists some edge  $uw \notin E(S_1) \subseteq E(T_1)$  such that  $f(u)f(w) \in E(S_2) \subseteq E(T_2)$ . In either case,  $f$  is not an isomorphism, which is a contradiction.

( $\Leftarrow$ ) Let  $f : V(T_1) \rightarrow V(T_2)$  be a mapping such that, for every vertex  $v \in V(T_1)$ , the subtree of  $T_1$  rooted at  $v$  is isomorphic to the subtree of  $T_2$  rooted at  $f(v)$ . Suppose, to the contrary, that  $f$  is not an isomorphism. Then, either there exists some edge  $uw \in E(T_1)$  such that  $f(u)f(w) \notin E(T_2)$  or there exists some edge  $uw \notin E(T_1)$  such that  $f(u)f(w) \in E(T_2)$ . Suppose it is the case that  $uw \in E(T_1)$  and  $f(u)f(w) \notin E(T_2)$ . Without loss of generality, let  $w$  be at a lower depth than  $u$ . Now, consider the subtree of  $T_1$  rooted at  $u$  (call it  $S_1$ ) and the subtree of  $T_2$  rooted at  $f(u)$  (call it  $S_2$ ). We have that  $uw \in E(S_1)$  and  $f(u)f(w) \notin E(T_2) \supseteq E(S_2)$ . Hence,  $S_1$  is not isomorphic to  $S_2$ , which is a contradiction. Similarly, if  $uw \notin E(T_1)$  and  $f(u)f(w) \in E(T_2)$ , we also arrive at a contradiction.  $\square$

Based on this proposition, we can construct an algorithm to determine whether rooted trees  $T_1$  and  $T_2$  are isomorphic. Starting at the lowest depth of  $T_1$  and  $T_2$ , assign labels to their vertices such that, for  $v \in V(T_1)$  and  $w \in V(T_2)$ , the labels of  $v$  and  $w$  are the same if and only if the subtree of  $T_1$  rooted at  $v$  is isomorphic to the subtree of  $T_2$  rooted at  $w$ .

To begin, assign the label 0 to all the leaves of  $T_1$  and  $T_2$  (the purpose of this

will become clear soon). Now, every vertex at the lowest depth of a tree (that is, the depth furthest from the root) is a leaf, and so the subtree rooted at that vertex is the empty tree. Hence, we begin by simply making certain that  $T_1$  and  $T_2$  have the same number of vertices at their lowest depths. Next, we move one level up. For each vertex  $v$  at this depth in both trees, assign the tuple label  $(a_1, a_2, \dots, a_k)$ , where the  $a_i$  are the labels of the  $k$  children of  $v$  arranged in nondecreasing order. Note that if we encounter another leaf, it will already be labeled 0, so we do not relabel it. Now, scan the vertices of  $T_1$  at this depth a second time. For each vertex  $v$  scanned, pair with  $v$  any vertex  $w$  of  $T_2$  such that  $v$  and  $w$  are labeled with the same tuple. We require that, once a vertex has been paired, it may not be paired again with another vertex. By the above proposition, the subtrees rooted at  $v$  and  $w$  of  $T_1$  and  $T_2$ , respectively, are isomorphic (since the subtrees rooted at their children are isomorphic). If we cannot pair all the vertices at the current depth of  $T_1$  and  $T_2$  (including the case when  $T_1$  and  $T_2$  do not have the same number of vertices at the current depth), we terminate with the knowledge that  $T_1 \not\cong T_2$ . Otherwise, we continue upward in the same way until we finally pair the root of  $T_1$  with the root of  $T_2$ . In this case, we know that  $T_1 \cong T_2$ .

**Remark.** In practice, we replace the tuples assigned to the vertices with integer labels before moving up (else we would have a tuple of tuples of tuples of ...). The labels can be assigned in any way so long as vertices  $v$  and  $w$  have the same integer label if and only if they have the same tuple label. An efficient way to accomplish this is to sort the vertices by considering their tuples as strings before assigning the

integer labels.

We present the following algorithm found in [1]. Note that leaves are never assigned true tuple labels, and so the phrase “tuple label” in reference to a leaf indicates the integer label 0 assigned to it at the beginning of the algorithm.

---

**Algorithm 3** `rootedIsomorphic( $T_1, T_2$ )`

---

**Input:** Two rooted trees  $T_1$  and  $T_2$  on  $n$  vertices (denote height of  $T_1$  by  $h$ )

**Output:** `true` if  $T_1 \cong T_2$ ; `false`, otherwise

Assign to all leaves of  $T_1$  and  $T_2$  the integer label 0

**for**  $i := h$  to 0 **do**

**for all** nonleaf vertices  $v$  of  $T_1$  and  $T_2$  at depth  $i$  **do**

    Assign to  $v$  the tuple label  $(a_1, a_2, \dots, a_k)$ , where the  $a_j$  are the integer labels of the  $k$  children of  $v$  arranged in nondecreasing order

**end for**

**for all** vertices  $v$  of  $T_1$  at depth  $i$  **do**

**if** there exists an unpaired vertex  $w$  of  $T_2$  at depth  $i$  such that  $v$  and  $w$  have the same tuple labels **then**

    Pair  $v$  with  $w$

**else**

**return false**

**end if**

**end for**

**if** there still exists an unpaired vertex of  $T_1$  or  $T_2$  at depth  $i$  **then**

**return false**

**end if**

Sort the vertices of  $T_1$  and  $T_2$  at depth  $i$  by treating their tuple labels as strings

  Traverse the sorted list and assign integer labels so that vertices  $v$  and  $w$  have the same integer label if and only if they have the same tuple label

**end for**

**return true**

---

**Example 3.1.1.** We present an example run of `rootedIsomorphic` on two isomorphic trees. Since the algorithm assigns the same label to the roots of both trees, it correctly determines that they are isomorphic.

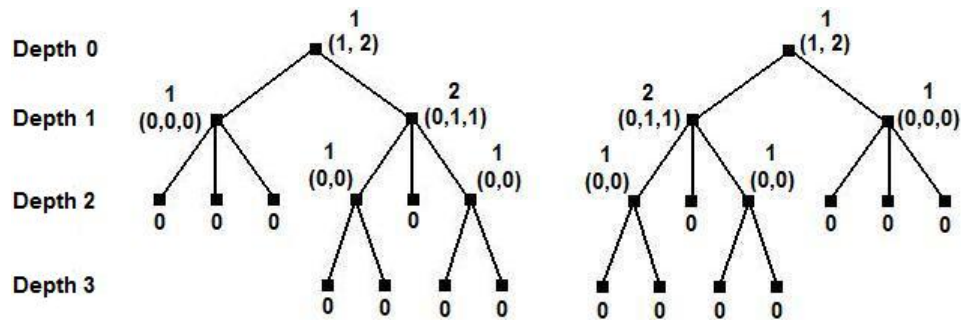


Figure 3.2. Example run of the rooted tree isomorphism test

**Remark.** Let  $T_1$  and  $T_2$  be trees on  $n$  vertices. Then, `rootedIsomorphic` runs in  $O(n)$  time (see [1]).

**Remark.** In this paper, we are not concerned with generating the isomorphism itself. We simply want to know whether such an isomorphism exists (i.e. if the two trees are isomorphic). If we wish to generate an isomorphism  $f$ , however, we need only take note of which vertices are associated with each other in the step “Pair  $v$  with  $w$ ”. That is, for each  $v_i \in V(T_1)$  paired with some  $w_j \in V(T_2)$ , we let  $f(v_i) = w_j$ .

### 3.2 GENERAL TREE ISOMORPHISM

In order to make use of the `rootedIsomorphic` algorithm to determine isomorphism between any two trees  $T_1$  and  $T_2$ , we must choose a vertex  $u \in V(T_1)$  and  $v \in V(T_2)$  such that any isomorphism  $f$  maps  $u$  to  $v$ . The concept of the center of a tree will allow us to find such vertices (with a slight technicality, to be explained later). The following definition comes from [8].

**Definition.** Let  $d(u, v)$  denote the distance between vertices  $u$  and  $v$ . The **eccentricity** of a vertex  $u$ , written  $\epsilon(u)$ , is  $\max_{v \in V(G)} d(u, v)$ . The **center** of a graph is the subgraph induced by the vertices of minimum eccentricity.

We make use of the following well-known result of Jordan, also found in [8].

**Theorem 3.2.1.** *The center of a tree is a vertex or an edge.*

*Proof.* We use induction on the number of vertices in a tree  $T$ , denoted  $n(T)$ .

Basis step:  $n(T) \leq 2$ . With at most two vertices, the center is the entire tree.

Induction step:  $n(T) > 2$ . Form  $T'$  by deleting every leaf of  $T$ . Clearly,  $T'$  is a tree. Since the internal vertices on paths between leaves of  $T$  remain,  $T'$  has at least one vertex.

Every vertex at maximum distance in  $T$  from a vertex  $u \in V(T)$  is a leaf (otherwise, the path reaching it from  $u$  can be extended farther). Since all the leaves have been removed and no path between two other vertices uses a leaf,  $\epsilon_{T'}(u) = \epsilon_T(u) - 1$  for every  $u \in V(T')$ . Also, the eccentricity of a leaf in  $T$  is greater than the eccentricity of its neighbor in  $T$ . Hence, the vertices minimizing  $\epsilon_T(u)$  are the same as the vertices minimizing  $\epsilon_{T'}(u)$ .

We have shown that  $T$  and  $T'$  have the same center. By the induction hypothesis, the center of  $T'$  is a vertex or an edge.  $\square$

**Remark.** When convenient, when the center of a tree is the edge  $uv$ , we will henceforth say instead that the tree has two centers  $u$  and  $v$ .

Jordan's proof not only asserts his claim, but also gives us a means to find the center itself. Given a tree, remove all of its leaves simultaneously. If any leaves remain, remove all of these simultaneously. Continue in this fashion until only one or two vertices remain, which will be the center(s) of the tree. In practice, however, we do not literally remove the leaves, as this destroys the tree we are trying to study. Instead, we maintain the degree of each vertex of the tree, decrementing them as necessary to simulate the removal of leaves.

---

**Algorithm 4** findCenter( $T$ )

---

**Input:** A tree  $T$  on  $n$  vertices**Output:** The center(s) of  $T$  (either one or two vertices)Set  $d_i$  to be the degree of vertex  $i$ **while** at least three of the  $d_i$  are positive **do**  **for**  $i := 1$  to  $n$  **do**    **if**  $d_i = 1$  **then**      Mark the vertex  $v_i$     **end if**  **end for****for all** marked vertices  $v_i$  **do**   $d_i := d_i - 1$   **for all** vertices  $v_j$  adjacent to  $v_i$  **do**     $d_j := d_j - 1$   **end for****end for****end while****return** vertices  $v_i$  where  $d_i$  is positive

---

Now, if  $T_1$  and  $T_2$  have unique centers  $c_1$  and  $c_2$ , respectively, we root  $T_1$  at  $c_1$  and  $T_2$  at  $c_2$ , then call `rootedIsomorphic( $T_1, T_2$ )`. If, on the other hand,  $T_1$  and  $T_2$  each have two centers (say,  $c_1, c'_1$  and  $c_2, c'_2$ , respectively), we may have to call `rootedIsomorphic` twice. We begin by rooting  $T_1$  at  $c_1$  and  $T_2$  at  $c_2$ . If `rootedIsomorphic( $T_1, T_2$ )` returns “true”, then  $T_1 \cong T_2$ . Otherwise, we leave  $T_1$  as it is and root  $T_2$  at  $c'_2$  before calling `rootedIsomorphic( $T_1, T_2$ )` again. As before, if this call returns “true”,  $T_1 \cong T_2$ . Otherwise,  $T_1 \not\cong T_2$ . Observe that if  $T_1$  and  $T_2$  do not have the same number of centers,  $T_1 \not\cong T_2$ .

Using the above description along with `rootedIsomorphic` and `findCenter`, we can formalize an algorithm to determine if any two trees are isomorphic.

---

**Algorithm 5**  $\text{isomorphic}(T_1, T_2)$ 

---

**Input:** Two trees  $T_1$  and  $T_2$

**Output:** **true** if  $T_1 \cong T_2$ ; **false**, otherwise

Let  $c_1$  and  $c'_1$  be the center(s) returned by  $\text{findCenter}(T_1)$

Let  $c_2$  and  $c'_2$  be the center(s) returned by  $\text{findCenter}(T_2)$

Root  $T_1$  at  $c_1$

Root  $T_2$  at  $c_2$

**if**  $\text{rootedIsomorphic}(T_1, T_2)$  returns **true** **then**

**return true**

**else if**  $c'_2$  exists **then**

    Root  $T_2$  at  $c'_2$

**if**  $\text{rootedIsomorphic}(T_1, T_2)$  returns **true** **then**

**return true**

**else**

**return false**

**end if**

**else**

**return false**

**end if**

---

**Remark.** All of the algorithms presented in this chapter are described at a lower level (that is, a level nearer implementation in a programming language) in [3].

CHAPTER 4  
PARTITIONING LABELED SPANNING TREES INTO  
ISOMORPHISM CLASSES

We now return to our original problem: Given an arbitrary labeled graph, partition all its spanning trees into equivalence classes under isomorphism. We accomplish this by combining `generateDescendants` (Chapter 2) and `isomorphic` (Chapter 3) in the following way. Place the tree  $T^*$  in a subset  $S_1$  and call `generateDescendants( $T^*, T^*$ )`. Now, for the first tree  $T_1$  output, call `isomorphic( $T^*, T_1$ )`. If we find that  $T^* \cong T_1$ , then we place  $T_1$  in  $S_1$ . Otherwise, we create a new subset  $S_2$  and place  $T_1$  in it. As each new tree is produced, we compare it (via `isomorphic`) to any representative from each of the  $S_i$  until we add it to one of the subsets or are forced to create a new subset.

---

**Algorithm 6** partitionSpanningTrees( $T, H$ )

---

**Input:** A spanning tree  $T$  and the edge-subset  $H = H(T)$

**Output:** Subsets  $S_i$  constituting a partition of the descendant spanning trees of  $T$  under isomorphism

**while**  $T$  belongs to no subset **and** there remains an unconsidered subset  $S_i$  **do**

    Select any tree  $T' \in S_i$

**if** isomorphic( $T, T'$ ) returns **true** **then**

        Add  $T$  to  $S_i$

**end if**

**end while**

**if**  $T$  still belongs to no subset **then**

    Create a new subset  $S_j$

    Add  $T$  to  $S_j$

**end if**

Construct  $D := T \setminus H$

**for**  $i := \text{btm}(T) + 1$  to  $\text{btm}(G)$  **do**

**if**  $e_i$  connects two distinct components of  $D$  **then**

        Construct  $I := \text{Cycle}(T, e_i) \cap H$

**for all** edges  $g \in I$  **do**

            Construct  $T' := (T \setminus g) \cup e_i$

            Construct  $H' := H \setminus \{e \in I \mid \text{index}(e) \geq \text{index}(g)\}$

            Call partitionSpanningTrees( $T', H'$ )

**end for**

**end if**

**end for**

---

**Remark.** As a technical point, the recursive nature of `partitionSpanningTrees` requires that the subsets  $S_i$  either exist outside the scope of the algorithm (i.e. in some calling program) or must be passed as arguments. We have chosen the former approach.

As with `generateDescendants`, if we wish to generate and partition *all* labeled spanning trees of  $G$ , our initial call must be `partitionSpanningTrees( $T^*, T^*$ )`, since  $T^*$  is the root of the “tree of trees”.

## CHAPTER 5

### SOME RESULTS

In this chapter, we present some results for selected graphs obtained by running our Java implementation of `partitionSpanningTrees`. In addition, the program generates an image of a representative tree from each isomorphism class. The source code, additional numbers, and images are available at [6]. In the tables below, the top number is  $\tau(G)$ , the number of labeled graphs (for which closed formulas are often known). The bottom number is  $I(G)$ , the number of unlabeled graphs (i.e. the number of isomorphism classes).

Spanning trees of the Complete Graph $K(n)$								
<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
1	3	16	125	1,296	16,807	262,144	4,782,969	100,000,000
1	1	2	3	6	11	23	47	106

Spanning trees of the Complete Bipartite Graph $K(s,t)$										
	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1	1
2	-	4	12	32	80	192	448	1,024	2,304	5,120
	-	1	2	2	3	3	4	4	5	5
3	-	-	81	432	2,025	8,748	35,721	139,968	531,441	1,968,300
	-	-	3	7	10	14	19	24	30	37
4	-	-	-	4,096	32,000	221,184	1,404,928	8,388,608		
	-	-	-	9	28	45	73	105		
5	-	-	-	-	390,625	4,050,000	37,515,625			
	-	-	-	-	37	132	242			
6	-	-	-	-	-	60,466,176				
	-	-	-	-	-	168				

Selected Graphs	
Petersen Graph	2,000 20
Tetrahedron	16 2
Cube	384 6
Octahedron	384 5
Icosahedron	5,184,000 434

As a demonstration of the image generating capabilities of the program, we present the output for  $K_6$ , the complete graph on six vertices.

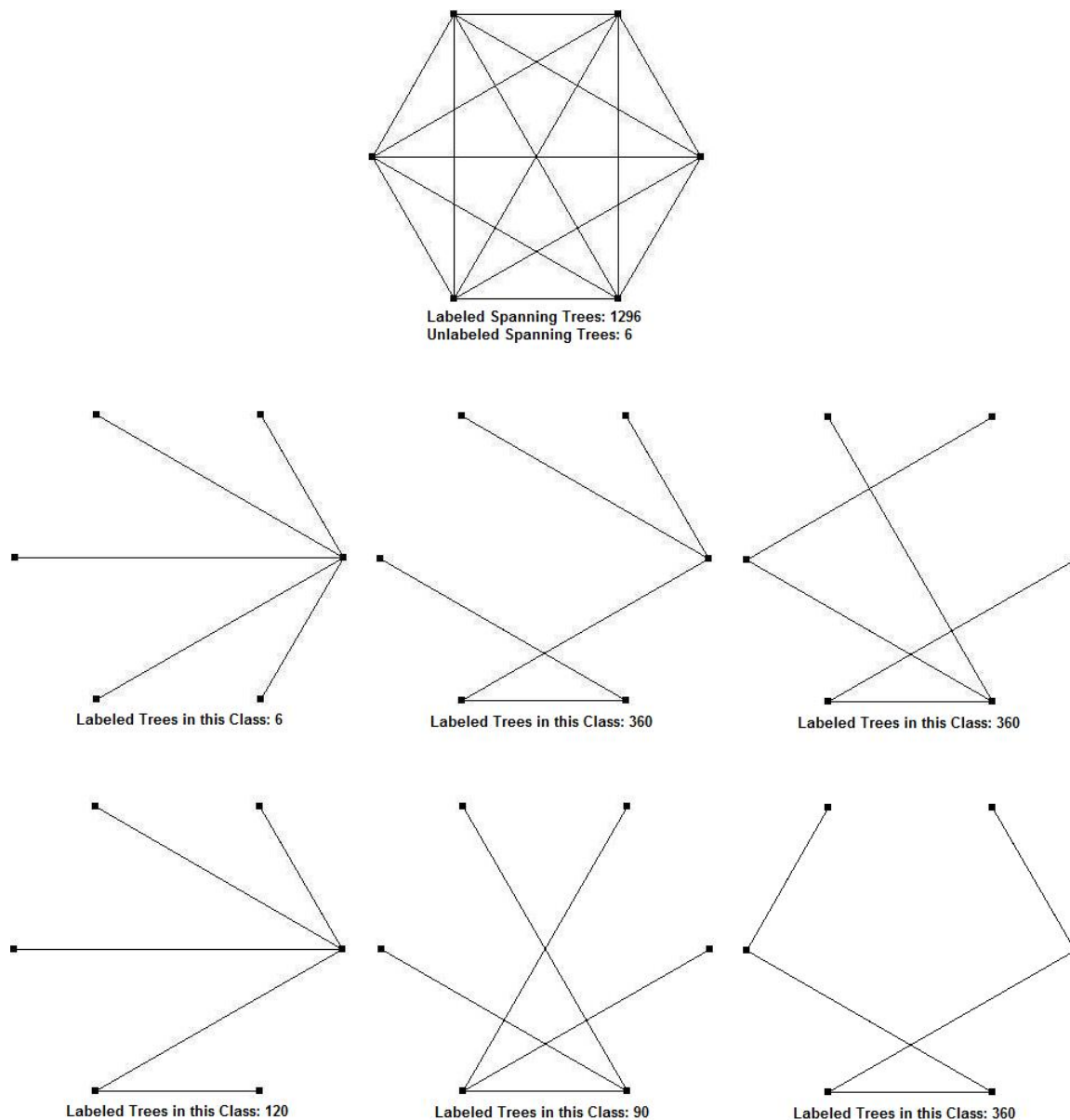


Figure 5.1. The spanning trees of  $K_6$  partitioned under isomorphism

## CHAPTER 6

### DEVELOPING A CLOSED FORMULA FOR $I(K_{S,T})$

#### 6.1 PRELIMINARIES

A separate but related problem is that of finding a closed formula for  $I(G)$ , the number of nonisomorphic spanning trees of a graph  $G$ . We consider here the case where  $G$  is  $K_{s,t}$ , the complete bipartite graph. In this section, we present some tools from [2] which are useful in approaching this problem.

When discussing partitions of vertices into subtrees, we will often refer to placing unlabeled balls (vertices) into either labeled or unlabeled buckets (subtrees). The following formulas will be used frequently.

**Definition.** Let  $p_k(n)$  be the number of partitions of  $n$  into at most  $k$  parts.

**Example 6.1.1.** We list the partitions of 6 into at most 3 parts to show that  $p_3(6) = 7$ .

$$6, 5+1, 4+2, 4+1+1, 3+3, 3+2+1, 2+2+2$$

**Proposition 6.1.1.** *The number of ways to arrange  $n$  unlabeled balls into  $k$  unlabeled buckets is given by  $p_k(n)$ .*

*Proof.* Let  $a_1 + a_2 + \cdots + a_k$  be a partition of  $n$ . This is in one-to-one correspondence with  $k$  buckets where bucket  $i$  contains  $a_i$  balls. Note that the commutativity of addition is equivalent to the buckets being unlabeled.  $\square$

The following case will occur frequently, so we make special mention of it here.

**Corollary 6.1.2.** *The number of ways to arrange  $n$  unlabeled balls into  $k$  unlabeled buckets ( $n, k \geq 2$ ) such that at least two buckets are nonempty is given by  $p_k(n) - 1$ .*

*Proof.* Combine the above proposition with the fact that there is only one illegal arrangement (all  $n$  balls in a single bucket).  $\square$

**Proposition 6.1.3.** *The number of ways to arrange  $n$  unlabeled balls into  $k$  labeled buckets is given by  $\binom{n+k-1}{n}$ .*

*Proof.* Let the buckets be labeled  $B_1 B_2 \dots B_k$ . Then, each assignment of the  $n$  balls to these buckets can be determined by a list of the  $n$  names of the buckets into which the balls go. For example, if two balls go into bucket  $B_1$ , one ball in bucket  $B_2$ , none in  $B_3$ , and four in  $B_4$ , we denote this arrangement by the list  $B_1 B_1 B_2 B_4 B_4 B_4 B_4$ . The number of ways of putting these balls into buckets can then be seen as the number of ways of choosing a redundant  $n$ -set of  $B$ 's from the  $k$ -set  $B_1 B_2 \dots B_k$ , which is given by  $\binom{n+k-1}{n}$  (see [2]).  $\square$

We point out another special case which occurs frequently.

**Corollary 6.1.4.** *The number of ways to arrange  $n$  unlabeled balls into  $k$  labeled buckets ( $n, m \geq 2$ ) such that at least two buckets are nonempty is given by  $\binom{n+k-1}{n} - k$ .*

*Proof.* Combine the above proposition with the fact that there are  $k$  illegal arrangements (all  $n$  balls placed in any one of the  $k$  labeled buckets).  $\square$

## 6.2 A METHOD FOR COMPUTING $I(K_{S,T})$

Recall from Chapter 3 that every tree has a unique center which is either a vertex or an edge. In  $K_{s,t}$ , this center is either a vertex in the  $s$ -set, a vertex in the  $t$ -set, or is an edge connecting the two. With this simple observation, we can break the nonisomorphic spanning trees of  $K_{s,t}$  into disjoint sets that are easier to count. We present two more facts that will allow us to work within each of these sets.

**Proposition 6.2.1.** *Let  $T$  be a tree rooted at  $r$  and let  $P_r(v)$  denote the length of the unique  $rv$ -path. The vertex  $r$  is the unique center of  $T$  if and only if there exists at least two vertices  $v_i$  such that:*

1.  $P_r(v_i)$  is maximal among all the vertices of  $T$
2. the  $rv_i$ -paths are disjoint except for the common vertex  $r$

*Proof.* ( $\Rightarrow$ ) Let  $r$  be the unique center of  $T$ .

Suppose, to the contrary, that condition (1) does not hold for at least two vertices. Then, some path (call it the  $rv$ -path) has maximum length. Perform the leaf removal algorithm on  $T$  as in `findCenter` (Chapter 3). It follows that all paths not containing  $v$  will be pruned up to  $r$  before the path containing  $v$  reaches  $r$ . If, at this point, we have a single edge remaining, the center of  $T$  is an edge. Otherwise, the next pruning will remove  $r$  from  $T$ . In either case, we see that  $r$  is not the unique center of  $T$ , which is a contradiction.

Now suppose, again to the contrary, that condition (1) holds, but condition (2) does not hold for at least two vertices. Then, all  $rv_i$  paths have some common

edge  $e$ . Perform the leaf removal algorithm on  $T$ . Eventually, we will prune all the  $rv_i$  paths up to  $e$ . We now have a tree with a single path of maximum length, so condition (1) ceases to hold. Hence,  $r$  is not the unique center of  $T$ , which is a contradiction.

( $\Leftarrow$ ) Let  $T$  be such that conditions (1) and (2) hold for at least two vertices. Denote these paths by  $rv_1, rv_2, \dots, rv_k$ . Perform the leaf removal algorithm on  $T$  as in `findCenter`. It follows that all other paths not among the  $rv_i$  will be pruned up to  $r$  before any of the  $rv_i$  reach  $r$ . Now, since there were at least two disjoint  $rv_i$  paths maximizing  $P_r$ , we know that  $r$  is not itself a leaf. Rather, we know the tree is now simply  $k$  paths of equal length radiating out from  $r$ . Continuing the leaf pruning, we see that  $r$  is indeed the center of  $T$ .  $\square$

**Corollary 6.2.2.** *Let  $T$  be a tree and denote the two trees obtained by the removal of the edge  $uv$  by  $S_1$  and  $S_2$ . Without loss of generality, let  $u \in S_1$  and  $v \in S_2$ . The edge  $uv$  is the center of  $T$  if and only if the maximum value of  $P_u$  in  $S_1$  is equal to the maximum value of  $P_v$  in  $S_2$ .*

*Proof.* It is clear that  $uv$  is the center of  $T$  if and only if the leaf removal algorithm reaches  $u$  at the same time it reaches  $v$ . The fact that the maximal paths are of the same length assures that this occurs at the same time in both  $S_1$  and  $S_2$ , leaving only the edge  $uv$ , which must then be the center of  $T$ .  $\square$

**Example 6.2.1.** The trees at the top are rooted at their centers, while the trees at the bottom are not. Note that we take some liberty here with the notion of “rooted” by rooting a tree at an edge.

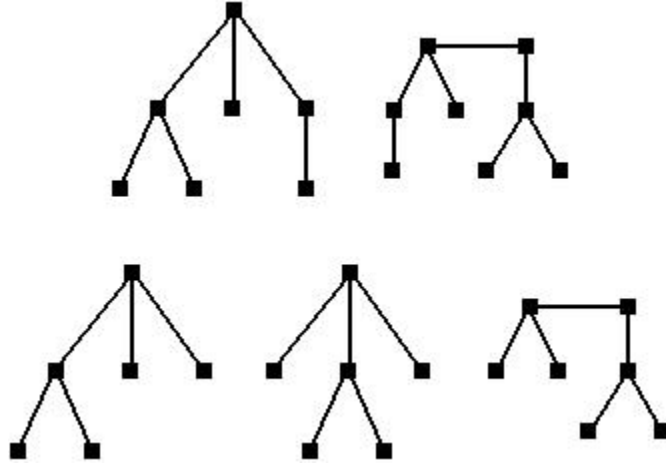


Figure 6.1. (Top) Trees rooted at their centers; (Bottom) Trees not rooted at their centers

We employ the ideas above to find a closed formula for  $I(K_{2,t})$  and  $I(K_{3,t})$ .

Note that  $I(K_{1,t}) = 1$ , since  $K_{1,t}$  is itself a tree, and so has only one spanning tree (namely, the entire graph  $K_{1,t}$ ).

**Proposition 6.2.3.**  $I(K_{2,t}) = p_2(t - 1)$ ,  $t \geq 2$

*Proof.* For the purposes of illustration, denote the 2-set by  $\{1, 2\}$  and the  $t$ -set by  $\{v_1, v_2, \dots, v_t\}$  (certainly, the vertices are not truly labeled, as we are considering classes up to isomorphism). Now, consider any spanning tree  $T$  of  $K_{2,t}$ . The center of  $T$  can be a vertex in the 2-set, a vertex in the  $t$ -set, or an edge adjoining the two partite sets.

Case unique center in 2-set: Without loss of generality, let the vertex 1 be the center of  $T$ . Observe in the figure below that there is no spanning tree of  $K_{2,t}$  with the center in the 2-set, as there are not two disjoint paths maximizing  $P_1$ .

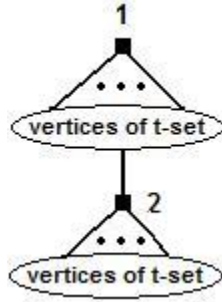


Figure 6.2. Configuration of  $K_{2,t}$  with center in 2-set

Case unique center in  $t$ -set: Without loss of generality, let the vertex  $v_1$  be the center of  $T$ . Observe in the figure below the only possible configuration of  $T$  that allows for the two disjoint paths required by Proposition 6.2.1. Of course, we require that neither of the ovals labeled “vertices of  $t$ -set” be empty, so the number of such trees is equal to the number of ways to partition the remaining  $t-1$  unlabeled vertices (balls) into these two unlabeled sets (buckets), both nonempty. Hence, there are  $p_2(t-1) - 1$  such trees.

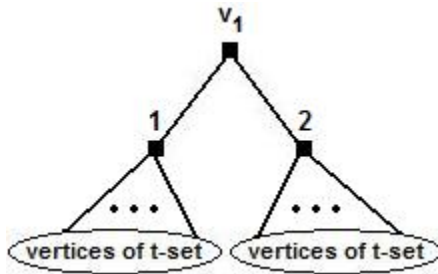


Figure 6.3. Configuration of  $K_{2,t}$  with center in  $t$ -set

Case center is an edge: Without loss of generality, let the edge  $1v_1$  be the center of  $T$ . Observe in the figure below that there is only a single tree that meets the criterion of Corollary 6.2.2.

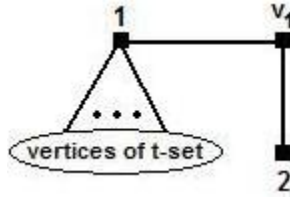


Figure 6.4. Configuration of  $K_{2,t}$  where center is an edge

Summing up the disjoint cases above, we see that  $I(K_{2,t}) = p_2(t-1) - 1 + 1 = p_2(t-1)$ . □

**Proposition 6.2.4.**  $I(K_{3,t}) = \sum_{k=2}^{t-2} p_2(k) + p_3(t-1) + 2, t \geq 4$

*Proof.* As before, denote the 3-set by  $\{1, 2, 3\}$  and the  $t$ -set by  $\{v_1, v_2, \dots, v_t\}$ .

Case unique center in 3-set: There are two valid configurations for  $T$ , shown in the figures below.

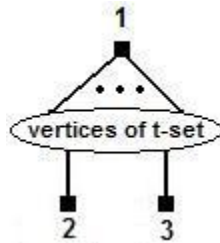


Figure 6.5. Configuration of  $K_{3,t}$  with center in 3-set (a)

Clearly, there is only one tree corresponding to the above configuration.

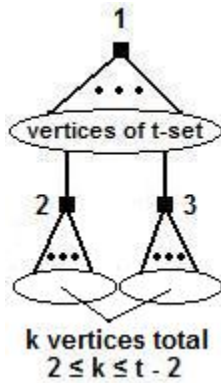


Figure 6.6. Configuration of  $K_{3,t}$  with center in 3-set (b)

For this configuration, we require  $k \geq 2$  (else the maximal paths are not the same length) and  $k \leq t - 2$  (else the maximal paths are not disjoint). Now, for a given  $k$ , we may partition them however we like between the vertices 2 and 3, which is equivalent to arranging  $k$  unlabeled balls into 2 unlabeled buckets, both nonempty. Observe that the buckets are considered to be unlabeled since switching the vertices 2 and 3 results in an identical tree. Hence, for this configuration, the number of trees is given by

$$\sum_{k=2}^{t-2} (p_2(k) - 1)$$

$$\sum_{k=2}^{t-2} p_2(k) - t + 3$$

Case unique center in  $t$ -set: There is a single valid configuration for  $T$ .

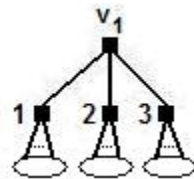


Figure 6.7. Configuration of  $K_{3,t}$  with center in  $t$ -set

Here, we simply arrange  $t - 1$  unlabeled balls into 3 unlabeled buckets, at least two nonempty. Hence, there are  $p_3(t - 1) - 1$  such trees.

Case center is an edge: We have two possible configurations.

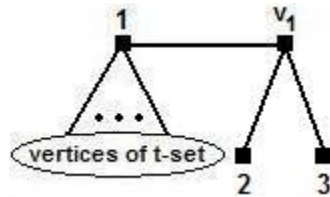


Figure 6.8. Configuration of  $K_{3,t}$  where center is an edge (a)

Clearly, there is only one such tree.

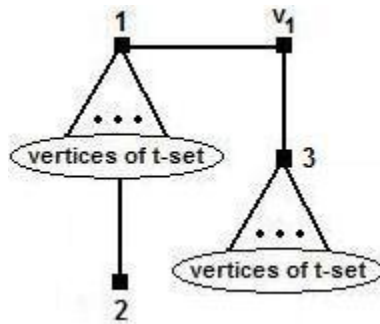


Figure 6.9. Configuration of  $K_{3,t}$  where center is an edge (b)

The number of trees here is equivalent to arranging  $t - 1$  labeled balls into two *labeled* buckets, since, the two “buckets” cannot be freely permuted. Of course, we also require that both sets be nonempty, so the number of trees is given by

$$\binom{(t-1)+2-1}{t-1} - 2$$

$$\binom{t}{t-1} - 2$$

$$t - 2$$

Summing up the disjoint cases above, we see that

$$\begin{aligned}
 I(K_{3,t}) &= 1 + \sum_{k=2}^{t-2} p_2(k) - t + 3 \\
 &\quad + p_3(t-1) - 1 \\
 &\quad + 1 + t - 2 \\
 &= \sum_{k=2}^{t-2} p_2(k) + p_3(t-1) + 2
 \end{aligned}$$

□

Using the method of generating functions, we can determine a closed form for  $p_2(n)$  and  $p_3(n)$ , allowing us to write the equations in Proposition 6.2.3 and Proposition 6.2.4 in terms of  $n$  only. We make use of the standard notation that  $[x^n]f$  denotes the coefficient of  $x^n$  in the power series expansion of  $f$ .

**Proposition 6.2.5.** *The value of  $p_k(n)$  is given by  $[x^n] \frac{1}{(1-x)(1-x^2)\dots(1-x^k)}$ .*

*Proof.* See [9].

□

**Proposition 6.2.6.**  $p_2(n) = \lfloor \frac{n}{2} \rfloor + 1$

*Proof.* Using the method of generating functions (combined with partial fraction

decomposition), we see that

$$\begin{aligned}
p_2(n) &= [x^n] \frac{1}{(1-x)(1-x^2)} \\
&= [x^n] \left( \frac{1}{2} \cdot \frac{1}{(1-x)^2} + \frac{1}{4} \cdot \frac{1}{1-x} + \frac{1}{4} \cdot \frac{1}{1-(-x)} \right) \\
&= \frac{1}{2} \binom{n+1}{1} + \frac{1}{4} + \frac{1}{4} (-1)^n \\
&= \frac{1}{4} (2n+3 + (-1)^n) \\
&= \left\lfloor \frac{n}{2} \right\rfloor + 1
\end{aligned}$$

□

**Proposition 6.2.7.**  $p_3(n) = \frac{1}{72}(6n^2 + 36n + 47 + 9(-1)^n + 8c)$ , where

$$c = \begin{cases} 2 & \text{if } n \equiv 0 \pmod{3} \\ -1 & \text{if } n \equiv 1, 2 \pmod{3} \end{cases}$$

*Proof.* Using the method of generating functions (combined with partial fraction decomposition), we see that

$$\begin{aligned}
p_3(n) &= [x^n] \frac{1}{(1-x)(1-x^2)(1-x^3)} \\
&= [x^n] \left( \frac{1}{6} \cdot \frac{1}{(1-x)^3} + \frac{1}{4} \cdot \frac{1}{(1-x)^2} + \frac{17}{72} \cdot \frac{1}{1-x} + \frac{1}{8} \cdot \frac{1}{1-(-x)} + \frac{1}{9} \cdot \frac{2+x}{1+x+x^2} \right) \\
&= \frac{1}{6} \binom{n+2}{2} + \frac{1}{4} \binom{n+1}{1} + \frac{17}{72} + \frac{1}{8} (-1)^n + \frac{1}{9} \cdot [x^n] \frac{2+x}{1+x+x^2}
\end{aligned}$$

Claim:  $\frac{2+x}{1+x+x^2} = 2 - x - x^2 + 2x^3 - x^4 - x^5 + \dots$

*Proof of Claim:* Multiplying both sides by  $1+x+x^2$  yields  $2+x$  on the left-hand side and, on the right-hand side, the telescoping sum

$$\begin{array}{cccccccc}
2 & -x & -x^2 & +2x^3 & -x^4 & -x^5 & \dots & \\
& +2x & -x^2 & -x^3 & +2x^4 & -x^5 & \dots & \\
& & +2x^2 & -x^3 & -x^4 & +2x^5 & \dots & \\
\hline
2 & +x & & & & & & 
\end{array}$$

which proves the claim.

By the claim, we see that  $[x^n]_{1+x+x^2} \frac{2+x}{1+x+x^2} = c$ , where  $c$  is as defined in the proposition. It follows

$$\begin{aligned}
p_3(n) &= \frac{1}{6} \frac{(n+2)(n+1)}{2} + \frac{1}{4}(n+1) + \frac{17}{72} + \frac{1}{8}(-1)^n + \frac{1}{9}c \\
&= \frac{1}{72}(6n^2 + 36n + 47 + 9(-1)^n + 8c)
\end{aligned}$$

□

**Corollary 6.2.8.**  $p_3(n) = \frac{1}{12}(n+3)^2$  (rounded to the nearest integer).

*Proof.*

$$\begin{aligned}
\left| \frac{1}{72}(6n^2 + 36n + 47 + 9(-1)^n + 8c) - \frac{1}{12}(n+3)^2 \right| &= \left| \frac{1}{72}(9(-1)^n + 8c - 7) \right| \\
&\leq \left| \frac{1}{72}(-9 - 8 - 7) \right| \\
&= \frac{1}{3}
\end{aligned}$$

□

We now rewrite the equations in Proposition 6.2.3 and Proposition 6.2.4.

**Proposition 6.2.9.**  $I(K_{2,t}) = \lceil \frac{t}{2} \rceil$ ,  $t \geq 2$

*Proof.*

$$\begin{aligned}
 I(K_{2,t}) &= p_2(t-1) \\
 &= \left\lfloor \frac{t-1}{2} \right\rfloor + 1 \\
 &= \left\lceil \frac{t}{2} \right\rceil
 \end{aligned}$$

□

**Proposition 6.2.10.**  $I(K_{3,t}) = \frac{1}{9}(3t^2 + 3t + 1 + c)$ ,  $t \geq 4$ , where

$$c = \begin{cases} 2 & \text{if } t \equiv 1 \pmod{3} \\ -1 & \text{if } t \equiv 0, 2 \pmod{3} \end{cases}$$

*Proof.*

$$\begin{aligned}
 I(K_{3,t}) &= \sum_{k=2}^{t-2} p_2(k) + p_3(t-1) + 2 \\
 &= \sum_{k=2}^{t-2} \left( \left\lfloor \frac{k}{2} \right\rfloor + 1 \right) + \frac{1}{72}(6(t-1)^2 + 36(t-1) + 47 + 9(-1)^{t-1} + 8c) + 2 \\
 &= \sum_{k=2}^{t-2} \left\lfloor \frac{k}{2} \right\rfloor + \frac{1}{72}(6(t-1)^2 + 36(t-1) + 47 + 9(-1)^{t-1} + 8c) + t - 1
 \end{aligned}$$

Claim

$$\sum_{k=2}^n \left\lfloor \frac{k}{2} \right\rfloor = \begin{cases} \frac{n^2}{4} & \text{if } n \text{ even} \\ \frac{n^2-1}{4} & \text{if } n \text{ odd} \end{cases}$$

*Proof of Claim*

Case n even

$$\begin{aligned}\sum_{k=2}^n \left\lfloor \frac{k}{2} \right\rfloor &= 2 \sum_{k=1}^{\frac{n}{2}} k - \frac{n}{2} \\ &= 2 \frac{n(\frac{n}{2} + 1)}{4} - \frac{n}{2} \\ &= \frac{n^2}{4}\end{aligned}$$

Case n odd

$$\begin{aligned}\sum_{k=2}^n \left\lfloor \frac{k}{2} \right\rfloor &= 2 \sum_{k=1}^{\frac{n-1}{2}} k \\ &= 2 \frac{(n-1)(\frac{n-1}{2} + 1)}{4} \\ &= \frac{n^2 - 1}{4}\end{aligned}$$

We now consider the value of  $I(K_{3,t})$  for the cases when  $t$  is even and when  $t$  is odd. In both cases, we arrive at the same result.

Case t even

$$\begin{aligned}I(K_{3,t}) &= \frac{(t-2)^2}{4} + \frac{1}{72}(6(t-1)^2 + 36(t-1) + 47 - 9 + 8c) + t - 1 \\ &= \frac{1}{9}(3t^2 + 3t + 1 + c)\end{aligned}$$

Case t odd

$$\begin{aligned}I(K_{3,t}) &= \frac{(t-2)^2 - 1}{4} + \frac{1}{72}(6(t-1)^2 + 36(t-1) + 47 + 9 + 8c) + t - 1 \\ &= \frac{1}{9}(3t^2 + 3t + 1 + c)\end{aligned}$$

□

**Corollary 6.2.11.**  $I(K_{3,t}) = \frac{1}{3}(t^2 + t + 1)$  (rounded to the nearest integer),  $t \geq 4$ .

*Proof.*

$$\begin{aligned} \left| \frac{1}{9}(3t^2 + 3t + 1 + c) - \frac{1}{3}(t^2 + t + 1) \right| &= \left| \frac{1}{9}(c - 2) \right| \\ &\leq \left| \frac{1}{9}(-1 - 2) \right| \\ &= \frac{1}{3} \end{aligned}$$

□

## REFERENCES

- [1] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [2] Cohen, Daniel I. A., *Basic Techniques of Combinatorial Theory*, John Wiley & Sons, New York, NY, 1978.
- [3] Crépeau, Claude, Tree Isomorphism,  
<http://crypto.cs.mcgill.ca/~crepeau/CS250/2004/HW5+.pdf>.
- [4] Gabow, H. and E. Myers, *Finding all spanning trees of directed and undirected graphs*, SIAM Journal of Computing, **7** (1978), 280–287.
- [5] Matsui, Tomomi, *A Flexible Algorithm for Generating All the Spanning Trees in Undirected Graphs*, Algorithmica, **18.4** (1997), 530–544.
- [6] Mohr, Austin, Isomorphism Classes of Spanning Trees of Selected Graphs,  
<http://www.austinmohr.com/work/trees>.
- [7] Nagamochi, Hiroshi and Toshihide Ibaraki, *A Linear-Time Algorithm for Finding a Sparse  $k$ -Connected Spanning Subgraph of a  $k$ -Connected Graph*, Algorithmica **7** (1992), 583–596.
- [8] West, Douglas B., *Introduction to Graph Theory*, Second Edition, Prentice-Hall, Upper Saddle River, NJ, 2001.
- [9] Wilf, Herbert S., *generatingfunctionology*, Third Edition, A K Peters, Wellesley, MA, 2006.

## VITA

Graduate School  
Southern Illinois University

AUSTIN MOHR

Date of Birth: June 26, 1985

602 SOUTH RUSSELL STREET, MARION, ILLINOIS 69259

Southern Illinois University at Carbondale  
Bachelor of Science, Mathematics, May 2007  
Bachelor of Science, Computer Science, May 2007

Special Honors and Awards:

Goldwater Scholarship in Pure Mathematics  
SIUC Master's Fellowship  
SIUC Chancellor's Scholarship  
NSF Graduate Research Fellowship Honorable Mention  
Carl G. Townsend Memorial Scholarship  
Outstanding Senior in Computer Science

Research Paper Title:

Partitioning the Labeled Spanning Trees of an Arbitrary Graph into Isomorphism  
Classes

Major Professor: Dr. Thomas D. Porter

Publications:

Clark, L. H., A. T. Mohr, and T. D. Porter, *Some applications of Spanning Trees in  $K(s,t)$* , Journal of Combinatorial Mathematics and Combinatorial Computing, **62** (2007), 139–146.