

# Pattern Avoidance in Ternary Trees

Nathan Gabriel, Katie Peske, Sam Tay

July 30, 2010

## Abstract

This paper considers the enumeration of ternary trees (i.e. rooted trees in which each vertex has 0 or 3 children) avoiding a contiguous ternary tree pattern. We begin by finding the recurrence relations for several simple ternary trees; then, for more complex trees, we extend a known algorithm for finding the generating function that counts  $n$ -leaf binary trees avoiding a given pattern. After investigating bijections between these trees' avoidance sequences and other common combinatorial objects, we conclude by finding a bijective method to restructure specific tree patterns that give the same generating function, and generalizing this process to a larger class of ternary trees.

## 1 Introduction

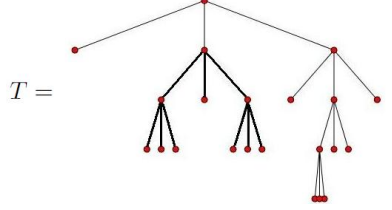
In recent years, pattern avoidance has proven to be a useful language to describe connections between various combinatorial objects. The notion of one object avoiding another has been studied in permutations, word, partitions, and graphs. In 2010, Rowland explored pattern avoidance in binary trees (that is, rooted trees in which each vertex has 0 or 2 children) because of the natural bijection between  $n$ -leaf binary trees and  $n$ -vertex trees. His study had two main objectives. First, he developed an algorithm to find the generating function denoting the number of  $n$ -leaf binary trees avoiding a given tree pattern; he adapted this to count the number of occurrences of the given pattern. Second, he determined equivalence classes for binary tree patterns, classifying two trees  $s$  and  $t$  as equivalent if the same number of  $n$ -leaf binary trees avoid  $s$  as avoid  $t$  for  $n \geq 1$ . He completed the classification for all binary trees with at most six leaves, using these classes to develop replacement bijections between equivalent binary trees [1].

In this paper, we extend Rowland's work by exploring pattern avoidance in ternary trees, i.e. ordered rooted trees in which each vertex has 0 or 3 children. We follow a similar outline to work done in binary trees. As a preface to our work, we define a new system of notation to represent  $m$ -ary trees (that is, trees where each vertex has 0 or  $m$  children), which we use to discuss ternary trees. We then find and explain the recurrence relations that count trees avoiding relatively simple ternary tree patterns (those with at most seven leaves), where the  $n$ th term denotes the number of  $n$ -leaf trees avoiding the given tree pattern. Next, we adapt Rowland's algorithm to find the avoidance generating function for ternary trees; this is followed with a discussion of a Maple package written to produce the terms of the series representation of the generating function for any tree taken as input. Finally, we put forth bijections for several pairs of equivalent tree patterns, and begin generalizing this process to fit a wider class of equivalent trees. The first appendix contains all the equivalence classes of ternary trees with at most nine leaves found using the Maple package; the Maple package itself is given in Appendix Two.

## 1.1 Definitions

### 1.1.1 Avoidance

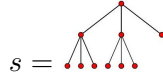
Following Rowland’s definitions of containment and avoidance, a ternary tree  $T$  *contains* a tree pattern  $t$  if there is a contiguous, rooted, and ordered subtree of  $T$  that is a copy of  $t$ . Conversely, it *avoids* the given pattern  $t$  if there is no such subtree of  $T$  that is a copy of  $t$ . For example, consider



$T$  contains the tree



because this pattern occurs beginning at the center child of the root of  $T$  (see bolded subtree). However,  $T$  avoids

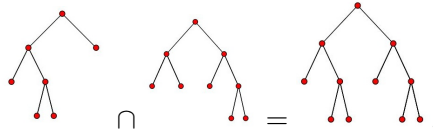


because no vertex in  $T$  has children extending from both its left and center children.

We define  $Av_n(t)$  to be the set of  $n$ -leaf ternary trees that avoid the given  $t$ , and  $av_n(t) = |Av_n(t)|$ . This notation will be used in both the recurrence relations and generating functions of the following sections.

### 1.1.2 Intersections

An important operation in analyzing trees is the intersection. The *intersection* of two trees, denoted by  $\cap$ , is the tree obtained by drawing one tree on top of the other, such that they have the same root. For example,



While “union” is often used for similar operations, we call it “intersection” here because when talking about trees with a certain pattern at the root, the set of trees with a tree pattern  $s$  at the root intersected with the set of trees with another pattern  $t$  at the root is the set of trees with  $s \cap t$  at the root.

The intersection will be used when converting trees to our numeric notation in section 3, as well as when finding a tree’s generating function in section 5.

### 1.1.3 Generating Functions

One of the most useful tools in analyzing pattern avoidance in ternary trees is the *generating function*,  $gf_t(x)$ . The generating function encodes the number of  $n$ -leaf trees that avoid a pattern

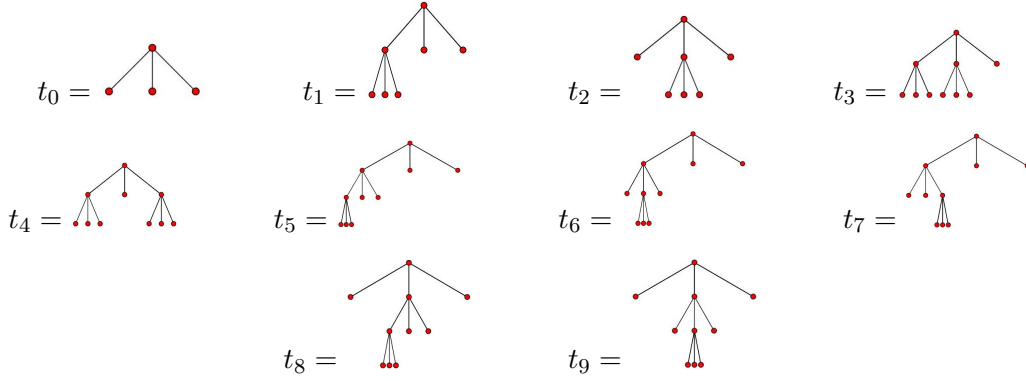
$t$ . In series form, we have

$$gf_t(x) = \sum_{n=0}^{\infty} av_n x^n,$$

where the coefficient  $av_n$  of  $x^n$  is the number of trees with  $n$  leaves avoiding  $t$ .

## 2 Ten Ternary Trees

Before we begin exploring avoidance in ternary trees, we first list all of the 3-, 5-, and 7-leaf trees.\* We will refer back to these by the assigned labels below.



\*Note: The same number of trees avoid the reflection of any given  $t$  as avoid  $t$ ; therefore, to avoid redundancy, we do not explicitly list the trees' reflections.

## 3 List Notation and $TS_m$

In order to discuss pattern avoidance in trees numerically in a clear and concise way, we present an alternate notation for ternary trees. This notation easily extends to  $m$ -ary trees (i.e. tree where each vertex has 0 or  $m$  children). This system, which we will refer to as “list notation,” will be especially useful when we consider bijections between sets of pattern-avoiding trees in section 6.

### 3.1 $m$ -leaf Parents and Sets of Lists

At the foundation of list notation are  $m$ -leaf parents, which will be used as the basis for determining a tree's representation as a set of lists.

**Definition 3.1.** *An  $m$ -leaf parent is an internal vertex,  $v$ , of an  $m$ -ary tree such that  $v$  has exactly  $m$  children, all of which are leaves.*

For example,  $t_6$  has one 3-leaf parent and  $t_3$  has two 3-leaf parents.

List notation represents an  $m$ -ary tree with a set of lists, where each list follows the path from the root to one  $m$ -leaf parent. We construct such a set from the following Tree-Set Algorithm:

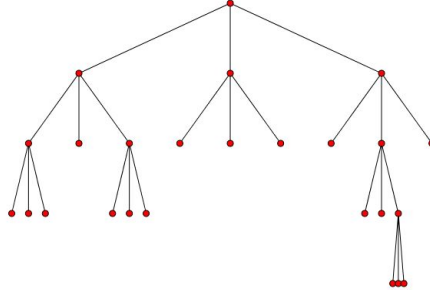
- Since we are dealing with ordered trees, label the children of each internal vertex of an  $m$ -ary tree from left to right, 1 through  $m$ . (In ternary trees, then, a vertex's left child is labeled 1, its center child 2, and its right child 3.)

- Denote a path from the root of a tree to an  $m$ -leaf parent by an ordered list of numbers  $[x_1, \dots, x_k]$ , where  $k$  is the length of the path from the root to the  $m$ -leaf parent, such that  $x_i \in \mathbb{Z}$  and  $1 \leq x_i \leq m$  for  $1 \leq i \leq k$ . The first number  $x_1$  in the list represents the child of the root labeled  $x_1$ ;  $x_i$  then refers to the child of the vertex given by  $x_{i-1}$  that is labeled  $x_i$ .

As an example, let us look again look at  $t_6$  and  $t_3$ . In  $t_6$ , the only 3-leaf parent is reached by a path beginning at the root, going to the root's left child, then to this vertex's center child; in list notation,  $t_6$  is denoted by the set of the single list  $\{[1, 2]\}$ . For  $t_3$ , we reach one of its 3-leaf parents by going to the root's left child, and the other by the root's center child; this becomes a set of two lists,  $\{[1], [2]\}$ .

**Theorem 3.2.** *An ordered  $m$ -ary tree  $T$  is uniquely defined by the set of paths from its root to each  $m$ -leaf parent. The single vertex tree is represented by an empty set,  $\{\}$ , and the 3-leaf tree ( $t_0$ ) by a set containing the empty list,  $\{[]\}$ .*

**Example 3.3.** *According to Theorem 3.2,*



*is uniquely defined by  $\{[1, 1], [1, 3], [2], [3, 2, 3]\}$ .*

To prove Theorem 3.2, we first state two lemmas.

**Lemma 3.4.** *In a finite  $m$ -ary tree, every internal vertex is either an  $m$ -leaf parent or has a descendant that is an  $m$ -leaf parent.*

*Proof.* Assume the lemma is false for the sake of a contradiction. Let  $v_0$  be an internal vertex of a finite  $m$ -ary tree such that neither it nor any of its descendants is an  $m$ -leaf parent. Since all internal vertices of an  $m$ -ary tree have  $m$  children, and since  $v_0$  is not an  $m$ -leaf parent, at least one of its children  $v_1$  must also have children of its own. Because  $v_1$  is an internal vertex, it must for the same reason have a child  $v_2$  that is an internal vertex. Thus, we generate an infinite list  $[v_0, v_1, v_2, \dots]$  of distinct vertices in our finite graph. This is a contradiction, proving the lemma to be true.  $\square$

**Lemma 3.5.** *In a finite  $m$ -ary tree,  $T$ , every leaf is the child of an  $m$ -leaf parent or the child of a vertex on a path to an  $m$ -leaf parent.*

*Proof.* Given a specific leaf  $l_0$ , there are two possible cases:

Case 1:  $l_0$  is the child of an  $m$ -leaf parent. The lemma is true trivially.

Case 2:  $l_0$  is not the child of an  $m$ -leaf parent. In this case, at least one of  $l_0$ 's siblings,  $l_1$ , must have children. Then, by Lemma 3.4  $l_1$  or one of its descendants is an  $m$ -leaf parent. Since the path between two vertices is always unique in a tree,  $l_0$ 's parent is on the path to this  $m$ -leaf parent.  $\square$

We now define a Set-Tree Algorithm to reverse the process of creating a set of lists; that is, a method to generate an  $m$ -ary tree from a set of lists:

1. Create an  $m$ -ary tree from each list by the following procedure:
  - (a) Create a root.
  - (b) Give the root  $m$  children, labeled left to right from 1 to  $m$ .
  - (c) For the list  $[x_1, x_2, \dots, x_k]$  give  $x_1$ -st child of the root  $m$  children. Label these children 1 to  $m$  as before, repeating the process at each level where  $x_i$  denotes giving  $m$  children to the  $x_i$ -th child of the vertex that was given children by  $x_{i-1}$ .
2. Take the intersection of all graphs obtained from step one to find the final  $m$ -ary tree.

With this algorithm and the two lemmas, we can now prove Theorem 3.2.

*Proof.* First, we prove that this Set-Tree Algorithm is an inverse map to the Tree-Set Algorithm; that is, given a set of lists produced by inputting a tree into the Tree-Set Algorithm, the Set-Tree Algorithm returns the original tree.

The algorithm gives all of the vertices on each path from an  $m$ -leaf parent to the root and all of those vertices' children by construction. Thus, by our two lemmas, the algorithm produces all of the internal vertices and leaves of the graph that our lists came from.

The algorithm does not give us a tree with extra vertices (i.e. vertices not in the original tree) since all of the vertices created by the algorithm are internal vertices from a path between an  $m$ -leaf parent and the root or the children of such vertices. By our lemmas, then, the algorithm preserves the number of vertices in the original tree.

Furthermore, the ordering of our graph is preserved by nature of the algorithm. Therefore we get the same tree from our set of lists as was originally used to obtain the set of lists.  $\square$

### 3.2 $TS_m$ Notation

**Definition 3.6.** Let  $S$  be a set of lists whose elements are integers 1 through  $m$ , such that no list is a prefix of another list in  $S$ . Namely,  $S$  is an arbitrary set of lists describing an  $m$ -ary tree. We write  $TS_m$  for the set of all such sets of lists  $S$ .

**Theorem 3.7.** Each  $S \in TS_m$  describes a distinct  $m$ -ary tree.

*Proof.* By the Set-Tree Algorithm, we know that the set of lists  $S$  describes some  $m$ -ary tree. Consider two distinct sets  $S_1$  and  $S_2$  of this form and some list  $L_0$  in  $S_1$  but not in  $S_2$ . Since  $L_0$  is not the prefix of another list it represents a path to an  $m$ -leaf parent  $v_0$  in the tree  $S_1$ . Since  $L_0$  is not in  $S_2$  then  $v_0$  is not an  $m$ -leaf parent in the tree given by  $S_2$ .  $\square$

**Theorem 3.8.** By applying our procedure for denoting an  $m$ -ary tree by a set of lists to all  $m$ -ary trees we obtain all  $S \in TS_m$ .

*Proof.* By the Tree-Set Algorithm for obtaining a set of lists from an  $m$ -ary tree, a list  $L_1$  denotes a path to an  $m$ -leaf parent  $v_1$ . If  $L_1$  is a prefix of another list  $L_2$ , where  $L_1 \neq L_2$ , then  $v_1$  is on the path to some other  $m$ -leaf parent  $v_2$ , and  $v_1$  would not be an  $m$ -leaf parent. Thus, we cannot obtain a set of lists from our procedure such that one list is the prefix of another. By Theorem 3.2,

we now know that our procedure produces a distinct  $S \in TS_m$ . Furthermore, by Theorem 3.7, each  $S \in TS_m$  produces a distinct  $m$ -ary tree. Therefore, by applying our procedure to each  $m$ -ary tree we must obtain each  $S \in TS_m$ .  $\square$

### 3.3 Pattern Avoidance in $TS_m$

In order to make this  $TS_m$  notation useful for defining bijections between ternary trees, we must identify exactly what pattern avoidance looks like using sets of lists.

Consider a set of lists  $\{L_i\}_{i=1}^l$  that denotes an  $m$ -ary tree  $t$ . A tree  $T$  denoted by  $\{M_h\}_{h=1}^p$  contains  $t$  if there exists  $\{M_{h_i}\}_{i=1}^l$  where each  $M_{h_i}$  begins with the same prefix (possibly the empty prefix) as all other  $M_{h_i}$ 's, followed by exactly the ordered sequence of elements of  $L_i$ ; this may or may not be then followed an additional sequence of numbers. This understanding of a tree  $T$  containing  $t$  follows directly from our previous understanding of one tree containing another. The prefix that we are concerned with is the path from the root of  $T$  to the root of the pattern contained in  $T$ . If the prefix is not the same for each list in  $T$ , then the paths of  $t$  do not begin at the same vertex in  $T$ ; thus, the pattern is not contained in  $T$ .

**Example 3.9.** The tree pattern  $t_a = \{[1, 3, 2, 3], [1, 2, 2, 3]\}$ , is contained by

$$T_1 = \{[3, 2, 3, \mathbf{1}, \mathbf{3}, \mathbf{2}, \mathbf{3}], [1, 1, 3, 2, 2], [3, 2, 3, \mathbf{1}, \mathbf{2}, \mathbf{2}, \mathbf{3}, 1, 1, 2]\}.$$

Notice that, after the prefix  $[3, 2, 3]$ , the first and third lists of  $T_1$  have exactly the sequence of each list of  $t$ .

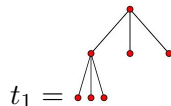
However,  $T_2 = \{[3, 1, 3, 2, 3], [1, 2, 2, 3]\}$  avoids  $t$ . Even though it contains each sequence of numbers from the lists of  $t$ ,  $T_2$ 's lists do not begin with the same prefix before the sequences begin.

Furthermore, the notion of a pattern occurring at a certain level of a tree is easily translated to list notation. If a pattern  $t$  occurs in a tree  $T$  where the lists denoting  $t$  are preceded by the prefix  $p_0$ , then  $p_0$  represents the path to the vertex  $v$  at which  $t$  is rooted. If  $v$  is at the  $i$ -th level of  $T$ , then  $p_0$  will be of length  $i - 1$ ; the converse is also true. Thus when a pattern  $t$  occurs at the  $i$ th level of a tree, the lists giving  $t$  all have a prefix of length  $i - 1$ .

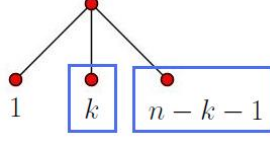
## 4 Finding Pattern Avoidance Recurrences

In this section, we find recurrence relations for the number of trees avoiding each of the ten trees labeled in section 2. For each tree, we discuss the structure of trees that avoid the given tree pattern, how the recurrence and generating function can be found from this structure, and we list any other equivalent tree patterns. As before, we let  $Av_n(t) = \{\text{trees with } n\text{-leaves that avoid } t\}$ , and  $av_n(t) = |Av_n(t)|$ ; that is,  $av_n(t)$  is the number of trees with  $n$  leaves that avoid  $t$ . (If  $T$  is clear in the given context, we will simply write  $Av_n$  and  $av_n$ .)

### 4.1 Avoiding $t_1$ and $t_2$



To find  $av_n(t_1)$ , let us look at how an  $n$ -leaf tree  $T$  must be structured in order to avoid  $t_1$ .



Consider any given internal vertex  $v$  of  $T$ . Its left child can have no descendants, thus it must be a leaf. Its center child can have a subtree of any number of leaves  $k$ , where  $1 \leq k \leq n - 2$  (these bounds ensure that  $v$  will still have three children). Finally,  $v$ 's right child can also have a subtree, but because there are  $n$  total leaves, it is restricted to  $n - k - 1$  leaves. Thus, there are  $av_k$  subtrees beginning at  $v$ 's center child, and  $av_{n-k-1}$  possible subtrees at  $v$ 's right child that also avoid  $t_1$ . Taking the summation of these over the possible values of  $k$  gives the recurrence relation

$$av_n = \sum_{k=1}^{n-2} av_k av_{n-k-1}.$$

Our initial conditions for this recurrence are  $av_0 = 0$ , because there are no trees with 0 leaves;  $av_1 = 1$ , because there is one tree with one leaf, and it avoids any tree pattern with more than one leaf; and  $av_2 = 0$ , because there are no trees with 2 leaves. We can now solve for  $gf_{t_1}(x) = \sum_{k=0}^{\infty} av_k x^k$  as follows:

$$\begin{aligned} av_n &= \sum_{k=1}^{n-2} av_k av_{n-k-1} \\ av_n x^n &= \sum_{k=1}^{n-2} av_k av_{n-k-1} x^n \\ \sum_{n=3}^{\infty} av_n x^n &= \sum_{n=3}^{\infty} \sum_{k=1}^{n-2} av_k av_{n-k-1} x^n \\ gf_{t_1}(x) - (av_2)x^2 - (av_1)x - (av_0) &= x \sum_{n=3}^{\infty} \sum_{k=1}^{n-2} av_k x^k av_{n-k-1} x^{n-k-1} \\ gf_{t_1}(x) - (av_1)x &= x \sum_{k=1}^{\infty} \sum_{n=k+2}^{\infty} av_k x^k av_{n-k-1} x^{n-k-1} \\ gf_{t_1}(x) - (av_1)x &= x \sum_{k=1}^{\infty} av_k x^k \cdot \sum_{n=k+2}^{\infty} av_{n-k-1} x^{n-k-1} \\ gf_{t_1}(x) - (av_1)x &= x(gf_{t_1} - av_0)^2 \\ x(gf_{t_1}(x))^2 - gf_{t_1}(x) + x &= 0 \\ gf_{t_1}(x) &= \frac{1 \pm \sqrt{1 - 4x^2}}{2x} \end{aligned}$$

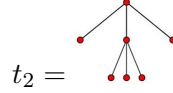
In order to get positive numbers for  $av_n$ , we discard one of these solutions, leaving us with:

$$gf_{t_1}(x) = \frac{1 - \sqrt{1 - 4x^2}}{2x}$$

The first few terms of this sequence are (for  $n \geq 0$ ):

$$0, 1, 0, 1, 0, 2, 0, 5, 0, 14, \dots$$

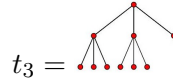
Two things are worth noting about this avoidance sequence. First, the non-zero terms make up the Catalan Sequence (OEIS A000108). Second, it is interpolated by zeros because there are no ternary trees with an odd number of leaves. This second observation will be true for the avoidance sequence of any ternary tree pattern.



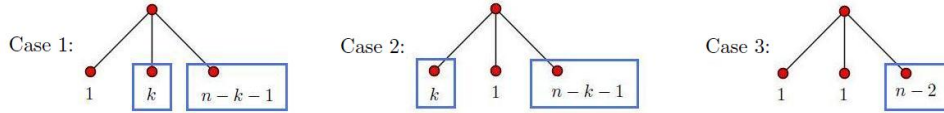
To look at trees avoiding  $t_2$ , we only need to make one alteration; namely, that it is the center child, instead of the leftmost child, of each vertex that cannot have any children. Therefore, we find that

$$gf_{t_1}(x) = gf_{t_2}(x) = \frac{1 - \sqrt{1 - 4x^2}}{2x}.$$

## 4.2 Avoiding $t_3$ and $t_4$



Next, we find the number of  $n$ -leaf trees that avoid  $t_3$  such that  $n \geq 3$ . As before, we consider any internal vertex  $v$  of a tree  $T$  that avoids  $t_3$ .



There are two nonexclusive possibilities for which of  $v$ 's children have leaves. First,  $v$ 's leftmost child has no children, but both its center and right children can. In the second case,  $v$ 's center child has no children, but both its left and right children can. These two cases are equivalent to avoiding  $t_1$  and  $t_2$ , respectively. However, this double-counts one instance: that is, when both the left and the center leaf of the  $t_0$  pattern have no children. There are exactly  $av_{n-2}$  trees counted by both of the first two cases. Subtracting this from the recurrence relation, we are left with:

$$av_n = 2 \sum_{k=1}^{n-2} av_k av_{n-k-1} - av_{n-2}$$

Our initial conditions for this recurrence relation are again  $av_0 = 0$ ,  $av_1 = 1$ , and  $av_2 = 0$ . We can now solve for  $gf_{t_3}(x) = \sum_{k=0}^{\infty} av_k x^k$  as follows:

$$av_n = 2 \sum_{k=1}^{n-2} av_k av_{n-k-1} - av_{n-2}$$



$$\begin{aligned}
av_n x^n &= 2 \sum_{k=1}^{n-2} av_k av_{n-k-1} x^n - av_{n-2} x^n \\
\sum_{n=3}^{\infty} av_n x^n &= 2 \sum_{n=3}^{\infty} \sum_{k=1}^{n-2} av_k av_{n-k-1} x^n - \sum_{n=3}^{\infty} av_{n-2} x^n \\
gf_{t_3}(x) - (av_2)x^2 - (av_1)x - (av_0) &= 2x \sum_{n=3}^{\infty} \sum_{k=1}^{n-2} av_k x^k av_{n-k-1} x^{n-k-1} - x^2 \sum_{n=3}^{\infty} av_{n-2} x^{n-2} \\
gf_{t_3}(x) - x &= 2x \sum_{k=1}^{\infty} av_k x^k \sum_{n=k+2}^{\infty} av_{n-k-1} x^{n-k-1} - x^2 (gf_{t_3}(x) - av_0) \\
gf_{t_3}(x) - x &= 2x(gf_{t_3}(x) - av_0)^2 - x^2 gf_{t_3}(x) \\
2x(gf_{t_3}(x))^2 - (x^2 + 1)gf_{t_3}(x) + x &= 0
\end{aligned}$$

After checking both solutions given by the quadratic equation, we find the generating function to be:

$$gf_{t_3}(x) = \frac{x^2 + 1 - \sqrt{x^4 - 6x^2 + 1}}{4x},$$

which gives the Little Schroeder numbers, interpolated by zeros: 1,0,1,3,0,11,0,45,0,197,0,... (OEIS A001003)

This is also the avoidance sequence for



As before, two cases exist for avoiding  $t_4$  (either the left and center or the right and center children of  $v$  have descendants), as well as a term that needs to be subtracted to avoid double-counting (when neither the left nor the right children of  $v$  have their own children). Thus, we have

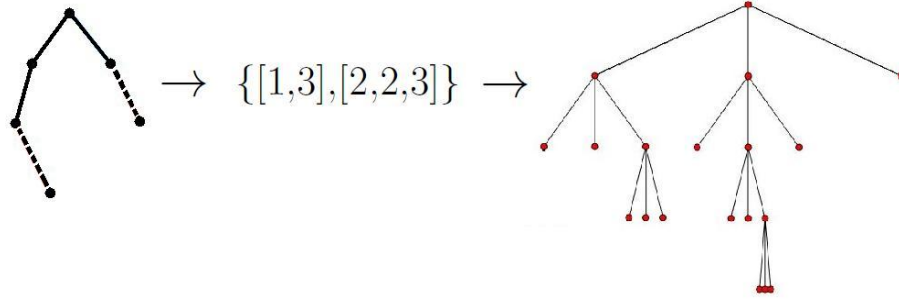
$$gf_{t_3}(x) = gf_{t_4}(x) = \frac{x^2 + 1 - \sqrt{x^4 - 6x^2 + 1}}{4x}.$$

#### 4.2.1 A Schroeder Bijection

In this subsection, we further examine the connection between trees avoiding  $t_4$  and the Little Schroeder number. To do this, we look at one well-known combinatorial interpretation of the Little Schroeder numbers:  $s_n$  is the number of binary trees with  $n$  vertices and with each right edge colored either red or blue (represented here as solid and dashed lines, respectively) [2]. To map these structures to the ternary trees avoiding  $t_4$ , we define the following bijection:

Begin by creating a list of numbers to represent each path from the root to one leaf of the colored binary tree  $b$ . Each list will consist of numbers 1, 2, or 3: a solid right edge is translated to a 1; any left edge, a 2; and a dashed right edge, a 3. Next, generate a ternary tree by using the aforementioned Tree-Set Algorithm (see Section 3.1).

For example, the colored binary tree below is mapped first to a set of lists, which is then translated to the shown ternary tree.



Let  $BR$  be the set of binary trees with  $n$  vertices and each right edge either solid or dashed, and  $B_s(b)$  be the stated Schroeder bijection for mapping  $b \in BR$  to a ternary tree avoiding  $t_4$ .

**Claim 1.** *For each  $b \in BR$ ,  $B_s(b)$  will always give a ternary tree avoiding  $t_4$ .*

*Proof.* In  $b$ , there are three options for each vertex's children: a left child, a dashed edge leading to a right child, or a solid edge leading to a right child. These can be combined in any way, except that the vertex can never have both a dashed and a solid edge leading to its right child. Therefore, any ternary tree pattern can be generated by our Schroeder bijection, except  $[1,3]$ , which corresponds to  $t_4$ , ensuring that all trees generated by this bijection avoid  $t_4$ .  $\square$

**Theorem 4.1.**  *$B_s(b)$  is a bijection, i.e. it is one-to-one and onto.*

*Proof.* First we show that  $B_s(b)$  is one-to-one. Given two binary trees,  $b_1$  and  $b_2 \in BR$ , if  $B_s(b_1) = t$  and  $B_s(b_2) = t$ , then for each of  $t$ 's lists, there is a corresponding path of left, solid right, or dashed right edges from the root of  $b_1$  to one of its leaves. Similarly, there is a corresponding path in  $b_2$ , and this path must be the same path as that in  $b_1$ . This implies that  $b_1$  and  $b_2$  have the same set of paths from the roots to each of their leaves. Therefore,  $b_1 = b_2$ .

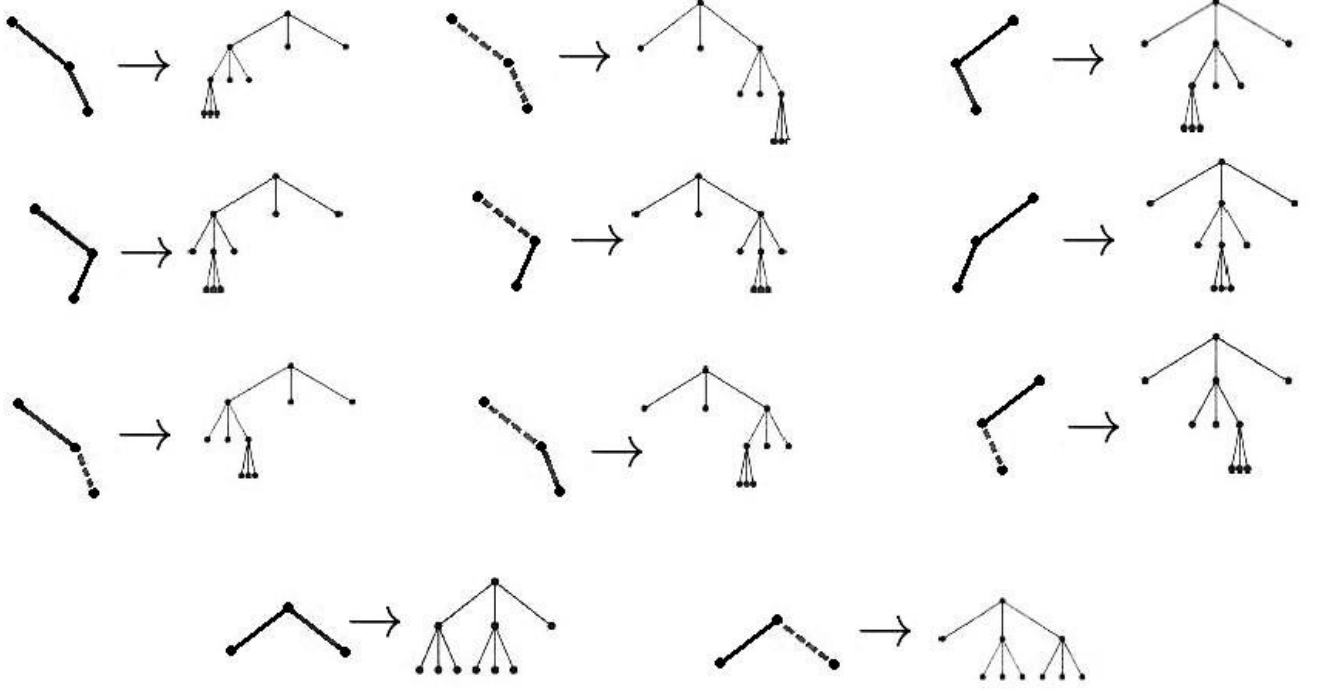
In order to prove that  $B_s(b)$  is onto, we define  $B_s^{-1}(t)$  to map a ternary tree  $t$  that avoids  $t_4$  to a binary tree  $b \in BR$ :

From each list  $L$  in  $t$ 's set of lists, draw a path from the root of  $b$  to a leaf in the following manner: for a 1, draw a solid right edge; for a 2, draw a left edge; for a 3, draw a dashed right edge.

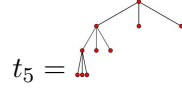
Now we show that the Schroeder bijection  $B_s(b)$  is onto. Let  $B_s^{-1}(t) = b$ . If  $t$  is a tree avoiding  $t_4$ , it does not have the pattern  $[1,3]$  in any of its lists. This implies that, in creating  $b$ , there will never be a case where a vertex in  $b$  should have both a dashed right edge and a solid right edge. Thus,  $b \in BR$ . Furthermore, because  $B_s(b) = t$ , that is  $B_s(B_s^{-1}(t)) = t$ , our Schroeder bijection is onto.  $\square$

As an example, look at  $s_3 = 11$ . There are eleven 3-vertex colored binary trees, and eleven 5-leaf trees avoiding  $t_4$ . Under the defined bijection  $B_s(b)$ , each colored binary tree  $b$  is mapped

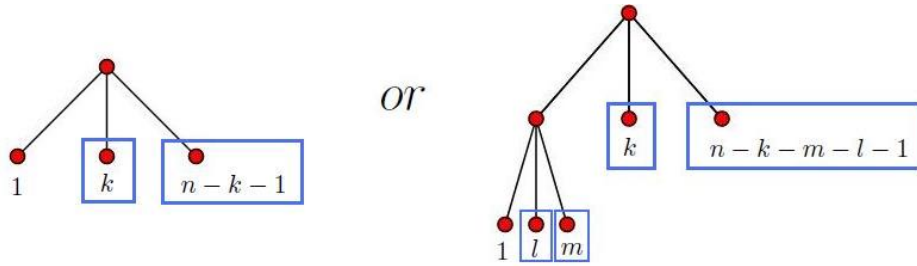
onto a tree avoiding  $t_4$  in the following way:



### 4.3 Avoiding $t_5$ and $t_9$

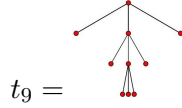


To find the number of  $n$ -leaf trees that avoid  $t_5$ ,  $n \geq 3$ , we consider two cases for any internal vertex  $v$  of a tree  $T$  that avoids  $t_5$ .



There are two possible structures that  $v$ 's children can follow. First,  $v$ 's left child has no children, while the center and right can have a combination of  $k$  leaves and  $n - k - 1$  leaves,  $1 \leq k \leq n - 2$ . The second case is when  $v$ 's left child has three children; to avoid  $t_5$ , a left-vertex child cannot have another consecutive left-vertex child. The four other vertices can have combinations of  $l$ ,  $m$ ,  $k$ , and  $n - l - m - k - 1$  children,  $1 \leq l, m, k \leq n - 4$ . Therefore,  $av_n(t_5)$  is given by the sum of these two cases:

$$av_n = \sum_{k=1}^{n-2} av_k av_{n-k-1} + \sum_{l=1}^{n-4} \sum_{m=1}^{n-l-3} \sum_{k=1}^{n-l-m-2} av_l av_m av_k av_{n-l-m-k-1}$$



To find the recurrence relation for trees avoiding  $t_9$ , we see that instead of avoiding two consecutive left-children vertices, we avoid two consecutive middle-children vertices. Therefore,  $av_n(t_5) = av_n(t_9)$  for  $n \geq 1$ .

Clearly, it would be extremely difficult to solve this recurrence directly for the generating function  $gf_{t_5}(x)$ . For trees  $t_6$ ,  $t_7$ , and  $t_8$ , complex problems arise with overcounting and undercounting, and we have not even been able to find their recurrence relations by hand. Instead, we adapt the generating function algorithm invented by Rowland for trees avoiding binary tree patterns to ternary tree patterns.


## 5 A Generating Function Algorithm

While finding a ternary tree's avoidance generating function from its recurrence relation works well when the trees are relatively small, the increasing complexity and intricacy soon make recurrence relations very inefficient. For this reason, we developed an algorithm to find the generating function  $gf_t(x)$  for any given tree  $t$ . First however, one clarification needs to be made. As mentioned in the introduction,  $gf_t(x)$  represents in series form the number of  $n$ -leaf ternary trees that avoid  $t$ . In this section, we omit the variable  $x$  to extend the generating function notation in a new direction. Now, let  $gf_t(p)$  be the generating function whose series form denotes the number of  $n$ -leaf ternary trees that (1) avoid  $t$ , and (2) contain the tree pattern  $p$  at their root. Therefore, the generating function for all trees avoiding  $t$  is given by  $gf_t(\bullet)$ , because all ternary trees begin with the single vertex root. (For simplicity, though, we will usually abbreviate  $gf_t(\bullet) = gf_t$ .)

The algorithm we use to find  $gf_t(\bullet)$  is very similar to Rowland's algorithm for binary trees [1], but it accounts for an additional child at each internal vertex. The process defines a series of generating functions using a recursive method. Initially,  $gf_t(\bullet)$ , the generating function we are interested in, is written in terms of another generating function. Then, for each new generating function  $gf_t(p)$  introduced in the recursive step, we must deduce another recurrence for  $gf_t(p)$  in terms of other generating functions. If  $t$  is a tree, we will use  $t_l$ ,  $t_c$ , and  $t_r$  to denote the left, center, and right subtrees of  $t$  respectively. With this new notation, our algorithm to find  $gf_t(\bullet)$  is as follows:

1.  $gf_t(\bullet) = x + gf_t(\text{root with three children})$
2.  $gf_t(p) = gf_t(p_l) \cdot gf_t(p_c) \cdot gf_t(p_r) - gf_t(p_l \cap t_l) \cdot gf_t(p_c \cap t_c) \cdot gf_t(p_r \cap t_r)$
3. For each new variable introduced in the right hand side of the equation in step 2, deduce a recurrence for  $gf_t(p)$  using step 2.
4. Solve the resulting system of equations for  $gf_t(\bullet)$ .

The first line of the algorithm defines  $gf_t(\bullet) = x + gf_t(\text{root with three children})$ . This is because, unless  $t$  is the single vertex, the generating function for trees with a single vertex at the root will always account

for the one tree with one leaf; from there, the rest of the trees avoiding  $t$  have a  pattern at the root, so  $x + gf_t(\text{root})$  constitutes the entire series of  $gf_t(\bullet)$ .

Next, we need to derive a recurrence for each unknown generating function, beginning with  $gf_t(\text{root})$ . To do this, we recognize that the generating function for trees with pattern  $p$  at their root,  $gf_t(p)$ , is made up of all of the possible combinations of the generating functions of its children's subtrees:  $gf_t(p_l)$ ,  $gf_t(p_c)$ , and  $gf_t(p_r)$ . However,  $gf_t(p_l)$ ,  $gf_t(p_c)$ , and  $gf_t(p_r)$  each account for the series of trees avoiding  $t$  individually, which overcounts when trees begin with roots of the intersections  $p_l \cap t_l$ ,  $p_c \cap t_c$ , and  $p_r \cap t_r$  respectively. Therefore, we have  $gf_t(p) = gf_t(p_l) \cdot gf_t(p_c) \cdot gf_t(p_r) - gf_t(p_l \cap t_l) \cdot gf_t(p_c \cap t_c) \cdot gf_t(p_r \cap t_r)$ .

We must derive a recurrence for any new generating function that arises from applying step 2 of the algorithm to a different generating function, until we have a complete system of equations. We then eliminate all unwanted variables until we have an equation consisting only of the variables  $gf_t(\bullet)$  and  $x$ . For very simple trees, we can usually solve directly for  $gf_t(\bullet)$ ; however, even with the 7-leaf tree,  $t_5 = \text{root}$ , we get a quartic equation from this procedure. To avoid solving directly for the whole function, we can substitute the  $k^{\text{th}}$  order series representation  $\sum_{n=0}^k av_n x^n$  for each  $gf_t(\bullet)$ , isolate the coefficients of each power of  $x$ , and set them each equal to zero. With these equations, we can solve for each coefficient  $av_n$ ,  $1 \leq n \leq k$ .

For example, we will show the process of finding the generating function for  $t_5$ :

$$gf_{t_5}(x) = gf_{t_5}(\bullet) = x + gf_{t_5}(\text{root})$$

$$gf_{t_5}(\text{root}) = (gf_{t_5}(\bullet))^3 - gf_{t_5}(\text{root}) \cdot (gf_{t_5}(\bullet))^2$$

$$gf_{t_5}(\text{root}) = gf_{t_5}(\text{root}) \cdot (gf_{t_5}(\bullet))^2 - gf_{t_5}(\text{root}) \cdot (gf_{t_5}(\bullet))^2$$

Let  $a = gf_{t_5}(\bullet)$ ,  $b = gf_{t_5}(\text{root})$ , and  $c = gf_{t_5}(\text{root})$ . Then,

$$a = x + b$$

$$b = a^3 - ca^2$$

$$c = ba^2 - ca^2$$

Eliminating  $b$  and  $c$  gives the equation  $a - x - a^4x - a^2x = 0$ . Substituting, for example, the 25<sup>th</sup> order series representation for  $a$  gives

$$\sum_{n=0}^{25} av_n x^n - x - \left( \sum_{n=0}^{25} av_n x^n \right)^4 x - \left( \sum_{n=0}^{25} av_n x^n \right)^2 x = 0.$$

Expanded, the first few powers of  $x$  have the following coefficients, which we set equal to zero:

$$x^0 : -q_0 = 0$$

$$x^1 : q_0^2 + 1 - q_1 + q_0^4 = 0$$

$$x^2 : 4 \cdot q_1 \cdot q_0^3 + 2 \cdot q_1 \cdot q_0 - q_2 = 0$$


$$x^3 : 6 \cdot q_1^2 \cdot q_0^2 + 4 \cdot q_2 \cdot q_0^3 + 2 \cdot q_2 \cdot q_0 + q_1^2 - q_3 = 0$$

$$x^4 : 4 \cdot q_1^3 \cdot q_0 + 2 \cdot q_1 \cdot q_2 + 2 \cdot q_3 \cdot q_0 + 12 \cdot q_1 \cdot q_2 \cdot q_0^2 + 4 \cdot q_3 \cdot q_0^3 - q_4 = 0$$

$[0, 1, 0, 1, 0, 3, 0, 11, 0, 46, 0, 207, 0, 979, 0, 4797, 0, 24138, 0, 123998, 0, 647615, 0, 3428493, 0, 18356714]$

## 5.1 Programming with Maple

To automate the algorithm for finding the generating function for ternary trees avoiding a certain tree pattern, we program the procedure in Maple. To perform operations on trees (i.e. finding left, center, and right subtrees), we represent them in a form consisting of lists of 1's and 0's. A single vertex is denoted  $[1, 0]$ . If a vertex has three children, there will be lists  $L_l$ ,  $L_c$ , and  $L_r$  in the second, third, and fourth positions within its list, representing the subtrees of its left,

center, and right children, denoted  $[1, L_l, L_c, L_r, 0]$ . For example,  $t_6 =$   is represented by  $[1, [1, [1, 0], [1, [1, 0], [1, 0], [1, 0], 0], [1, 0], 0], [1, 0], [1, 0], 0]$ . Clearly, writing trees in this form gets complicated quickly and can be very tedious. One of our programs in Maple converts trees from the list notation of Section 3 to this  $[1, 0]$ -form.



### 5.1.1 Notation Conversion

The conversion from “list notation” to  $[1,0]$ -notation uses two procedures. The first procedure,  $\text{Listt}(B)$ , takes one list of 1’s, 2’s, and 3’s called  $B$  as an input. Notice this procedure will produce trees that have precisely one  $m$ -leaf parent; that is, for any vertex in a tree produced by  $\text{Listt}(B)$ , at most one of its children will have three leaves. The base case states

```
if B=[] then return [1,[1,0],[1,0],[1,0],0]; end if;.
```

The list to be returned is then recursively defined with if statements concerning the first integer of the input list:

```

if B[1]=1, then return [1,TS(B[2..nops(B)]),[1,0],[1,0],0]; end if;
if B[1]=2, then return [1,[1,0],TS(B[2..nops(B)]),[1,0],0]; end if;
if B[1]=3, then return [1,[1,0],[1,0],TS(B[2..nops(B)]),0]; end if;

```

The first integer determines if the root's left, center, or right child has children. Then the program recursively calls itself to determine the structure of the subtree whose root is that chosen vertex. The procedure continues until it completes the length of the path to the 3-leaf parent,  $|B|$ .

The second procedure, **Sett**(**A**), takes one set called *A* as an input. Two base cases account for our convention for the empty set and the set containing the empty list,

```

    if A={} then return [1,0]; end if;
if A=[] then return [1,[1,0],[1,0],[1,0],0]; end if;

```

From here, we simply intersect the trees, as explained in the next section, given by each list produced by the `Listt(B)` procedure.

### 5.1.2 Intersection

The intersection procedure, `interstree(T1, T2)`, is essential for `Listt(A)` and the main avoidance program (the algorithm adapted from Rowland). It is based on the fact that when aligning the roots of two trees, the entire intersection can be found by combining the intersections of subtrees with overlapping vertices. If one of the overlapping vertices has no children, we can simply return the subtree given by the other tree, even if it is also a single vertex. Therefore, with two trees  $T1$  and  $T2$  (in  $[1,0]$ -form) as inputs, our base cases are

```
if T1=[1,0] then return T2; end if;
if T2=[1,0] then return T1; end if;
```

If a single vertex is not found, then there are still overlapping vertices descending from  $T1$  and  $T2$ , and we intersect them recursively. When neither of the above if statements are true, we know each tree has three children:

```
return [1, interstree(T1[2], T2[2]), interstree(T1[3], T2[3]),
        interstree(T1[4], T2[4]), 0]
```

The list positions  $[2,3,4]$  are the left, center, and right vertices' subtrees respectively. Whenever an overlapping vertex has no children, the other vertex's subtree contributes to the overall intersection.

### 5.1.3 Avoidance Algorithm

The full avoidance program, `Av(A,k)`, is dependent on the sets  $U$ ,  $V$ , and  $E$ .  $U$  is the set of variables for which the algorithm has already derived a recurrence (i.e., the  $gf_t(p)$  in line two of the algorithm),  $V$  is the set of all of the variables found in the equations (i.e.,  $gf_t(p)$ ,  $gf_t(p_r)$ ,  $gf_t(p_r \cap t_r)$ , etc., in line two of the algorithm), and  $E$  is the set of all equations in the algorithm. Operations on trees are done through subscripts of the variable  $a$ , so that we can use specific generating functions as variables of algebraic equations when solving for  $gf_t(\bullet)$ , which is denoted  $a_{[1,0]}$ . First, the first and second lines of the algorithm are defined as  $z_1$  and  $z_2$ , respectively. We then have a loop that continues while  $V \setminus U \neq \{\}$ , defining each unknown value from the set  $V \setminus U$ , using the avoidance algorithm described earlier. Each equation is assigned to names of  $z_c$ , and the left-hand side of these equations are each of the  $a$  variables that have been defined, with trees in the subscript.

When the loop is finished and all variables are defined, we eliminate the left-hand side of each equation  $z_c$ ,  $2 \leq c \leq |E|$ , finding an expression of  $a_{[1,0]}$ 's and  $x$ 's set equal to zero. As mentioned earlier, solving for the generating function directly from this expression gets very complicated; instead, we substitute the series representation to the order of the input  $k$ ,  $\sum_{n=0}^k q_n x^n$ , for each  $a_{[1,0]}$ . We then isolate each coefficient of  $x$ , set each of them equal to zero, and simultaneously solve for each  $q_n$ . The program then outputs  $q_n$ ,  $0 \leq n \leq k$ , which are the first  $k$  integers of the avoidance sequence. To see more detail, refer to the second appendix which has the complete Maple code.

## 6 Bijections on Ternary Trees

Now that we have discussed two methods for enumerating pattern-avoiding trees, we look for connections between specific sets of those trees. Recall that several of the ternary trees in this paper have had the same pattern avoidance sequence as one or more other trees. That is, for some distinct  $i, j$ , we found that  $gf_{t_i}(x) = gf_{t_j}(x)$ . We will now give some explanation of why this is the case. As Rowland did, we accomplish this through finding bijections, or "replacement rules," between the members of  $Av_n(t_i)$  and those of  $Av_n(t_j)$ .

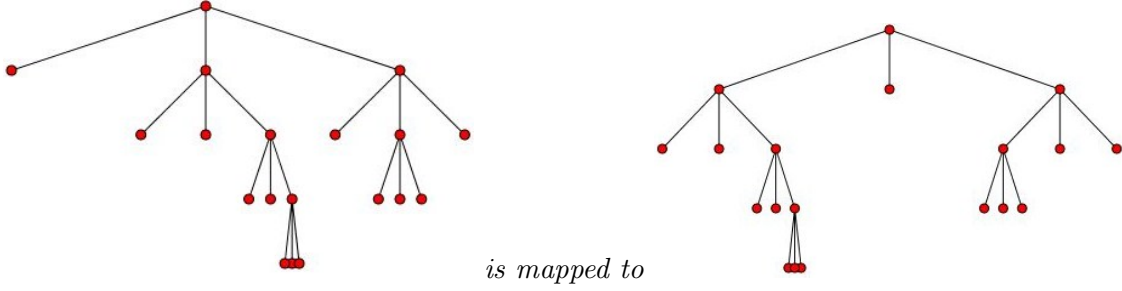
## 6.1 $B_{t_1, t_2}(T)$

Before we look at a bijection between  $t_1$  and  $t_2$ , we define the notation for such bijections.  $B_{t_i, t_j} : TS_m \rightarrow TS_m$  will denote a bijection between trees avoiding  $t_i$  and trees avoiding  $t_j$ . We find such maps by analyzing the list notation for our pattern-avoiding trees.



A tree avoids  $t_1$  if none of its left vertices have children, and avoids  $t_2$  if none of its center vertices have children. In order to map a tree  $T$  avoiding  $t_1$  to a tree avoiding  $t_2$ , we define our bijection  $B_{t_1, t_2}(T)$  to “switch” the center subtree of every vertex with the left subtree of the same vertex. In terms of list notation, a tree avoids  $t_1$  if it has no 1’s in its lists, and avoids  $t_2$  if it has no 2’s in its lists. Thus, we define  $B_{t_1, t_2}(T)$  to replace every 1 in  $T$ ’s lists with 2, and every 2 with a 1.

### Example 6.1.



The same example in list notation is  $B_{t_1, t_2}(\{[2, 3, 3], [3, 2]\}) = \{[1, 3, 3], [3, 1]\}$ .

To prove that  $B_{t_1, t_2}(T)$  does send trees avoiding  $t_1$  to trees avoiding  $t_2$ , we note that the trees avoiding  $t_1$  are exactly the sets that do not contain any 1’s in any of the sets’ lists. Similarly, the sets denoting trees avoiding  $t_2$  are exactly the sets that do not contain any 2’s in any of their lists. Because of this,  $B_{t_1, t_2}(T)$  will map a set of lists containing no 1’s to a set of lists containing no 2’s. This is equivalent to saying that  $B_{t_1, t_2}(T)$  does, in fact, send trees avoiding  $t_1$  to trees avoiding  $t_2$ .

### Theorem 6.2. $B_{t_1, t_2}$ is a bijection.

*Proof.* Consider  $T_1, T_2 \in TS_3$ . If  $B_{t_1, t_2}(T_1) = T_3 \in TS_3$  and also  $B_{t_1, t_2}(T_2) = T_3$ , then for each list  $L_3 \in T_3$ , there exists  $L_1 \in T_1$ , and  $L_2 \in T_2$  that are both the same as  $L_3$  except with 1’s in  $L_3$  replaced with 2’s and 2’s with 1’s. So  $L_1 = L_2$ . Therefore  $B_{t_1, t_2}(T)$  is one-to-one.

Let  $B_{t_1, t_2}^{-1}(T) = B_{t_1, t_2}(T)$ . We originally defined  $B_{t_1, t_2}(T)$  as a function on the set  $TS_3$ , so, though we later restricted our function to sets of lists that denote a tree avoiding  $t_1$ , it is still sensible to talk about  $B_{t_1, t_2}(T)$  in a more general way. If a tree avoids  $t_2$  it contains no 2’s in its set of lists. By applying  $B_{t_1, t_2}^{-1}(T)$  to such a set of lists, we will obtain a set of lists containing no 1’s, thus avoiding  $t_1$ . From this, we can see that  $B_{t_1, t_2}^{-1}(B_{t_1, t_2}(T)) = T$ . Therefore,  $B_{t_1, t_2}(T)$  is onto.



We also must be sure that  $B_{t_1,t_2}(T)$  preserves the number of leaves of a tree  $T$ ; that is, that the number of leaves of  $T$  is equal to number of leaves of  $B_{t_1,t_2}(T)$ . It is enough to show that for each internal vertex  $v_1$  in  $T$  there is a unique internal vertex  $v_2$  in  $B_{t_1,t_2}(T)$  that has the same number of leaves as  $v_1$ . With list notation, there is a prefix  $p_1$  describing the path to  $v_1$  in  $T$ . If  $x$  is the number of lists in  $T$  that have the prefix  $p_1$ , then  $v_1$  has  $3 - x$  children that are leaves. After the bijection is completed,  $p_1$  is mapped to the prefix  $p_2$  of some list(s) in  $B_{t_1,t_2}(T)$ . Since nothing else is mapped to  $p_2$ , we will have exactly  $x$  lists in  $B_{t_1,t_2}(T)$  with the prefix  $p_2$ . Thus,  $p_2$  denotes a path to an internal vertex  $v_2$  that will also have  $3 - x$  leaves. Because this holds true for every vertex  $v$  in  $T$ , the number of leaves is preserved by the bijection  $B_{t_1,t_2}(T)$ .  $\square$

## 6.2 $B_{t_3,t_4}(T)$



For a tree  $T$  to avoid  $t_3$ , no vertex  $v$  can have children descending from both its left and center children; to avoid  $t_4$ , no vertex can have children descending from both its left and right children. Therefore, we define a bijection  $B_{t_3,t_4}(T)$  to switch the right and center subtrees of each vertex. Using list notation, this is equivalent to defining  $B_{t_3,t_4}(T)$  to replace every 2 with a 3 and every 3 with a 2.

### Example 6.3.



*is mapped to*

Using lists,  $B_{t_3,t_4}(\{[1, 2, 1], [1, 2, 3, 2], [3, 2, 2], [3, 3, 1]\}) = \{[1, 3, 1], [1, 3, 2, 3], [2, 2, 1], [2, 3, 3]\}$

To show that  $B_{t_3,t_4}(T)$  does send trees avoiding  $t_3$  to trees avoiding  $t_4$ , we note that  $t_3$  is denoted  $\{[1], [2]\}$  and  $t_4$  is denoted  $\{[1], [3]\}$ . A tree denoted by  $T = \{L_i\}_{i=1}^l$  avoids  $t_3$  if there are no two lists  $L_1, L_2$  such that both lists begin with the same prefix followed by a 1 in  $L_1$  and a 2 in  $L_2$ . Thus, after the mapping,  $B_{t_3,t_4}(T)$  is a set of lists with no two lists  $M_1, M_2$  such that both begin with the same prefix followed by a 1 in  $M_1$ , and a 3 in  $M_2$ . This is exactly the requirement for a set of lists to denote a tree that avoids  $t_4$ .

**Theorem 6.4.**  $B_{t_3,t_4}(T)$  is both one-to-one and onto.

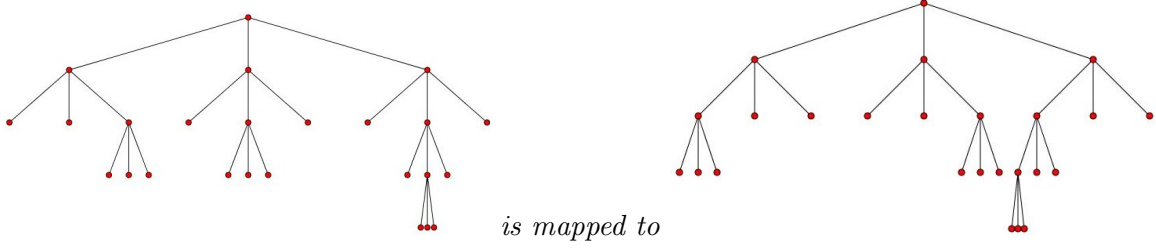
*Proof.* Since this is another replacement rule, the proof that  $B_{t_3,t_4}(T)$  is a bijection is nearly identical to that of  $B_{t_1,t_2}(T)$ . Only a few slight differences have to be noted. First, in the proof that  $B_{t_3,t_4}(T)$  is one-to-one, the lists  $L_1 \in T_1$ , and  $L_2 \in T_2$  are both the same as  $L_3 \in T_3$  except with 3's in  $L_3$  replaced with 2's and 2's with 3's. Second, to prove that  $B_{t_3,t_4}(T)$  is onto, we let  $B_{t_3,t_4}^{-1}(T) = B_{t_3,t_4}(T)$ , and after applying the function  $B_{t_3,t_4}^{-1}(T)$  to such a tree, we get a set of lists such that there are no two lists  $L_1, L_2$  such that both lists begin with the same prefix followed by a 1 in  $L_1$  and a 2 in  $L_2$ . From this, it can be seen that  $B_{3,4}(B_{3,4}(T)) = T$ .  $\square$

### 6.3 $B_{t_5, t_9}(T)$



A tree avoids  $t_5$  if no two consecutive left vertices have children, and avoids  $t_9$  if no two consecutive center vertices have children. In list notation, then, a tree avoids  $t_5$  if it has no pairs of consecutive 1's in its lists, and it avoids  $t_2$  if it has no pairs of consecutive 2's. Thus, we define  $B_{t_5, t_9}(T) = B_{t_1, t_2}$ . That is, for a tree  $T$  avoiding  $t_5$ ,  $B_{t_5, t_9}(T)$  replaces each 1  $T$ 's set of lists with a 2, and each 2 with a 1. Because we originally defined  $B_{t_1, t_2}(T)$  on  $TS_3$  defining  $B_{t_5, t_9}(T)$  in this way is reasonable.

#### Example 6.5.



In list notation,  $B_{t_5, t_9}([1, 3], [2, 2], [3, 2, 2]) = [2, 3], [1, 1], [3, 1, 1]$ .

Again, we assert that  $B_{t_5, t_9}(T)$  does send trees avoiding  $t_5$  to trees avoiding  $t_9$ . By the way we defined it, the function  $B_{t_5, t_9}(T)$  will map a set of lists containing no pairs of consecutive 1's to a set of lists containing no pairs of consecutive 2's by its replacing every 1 with a 2, and every 2 with a 1. This is equivalent to sending trees avoiding  $t_5$  to trees avoiding  $t_9$ .

**Theorem 6.6.**  $B_{t_5, t_9}(T)$  is both one-to-one and onto.

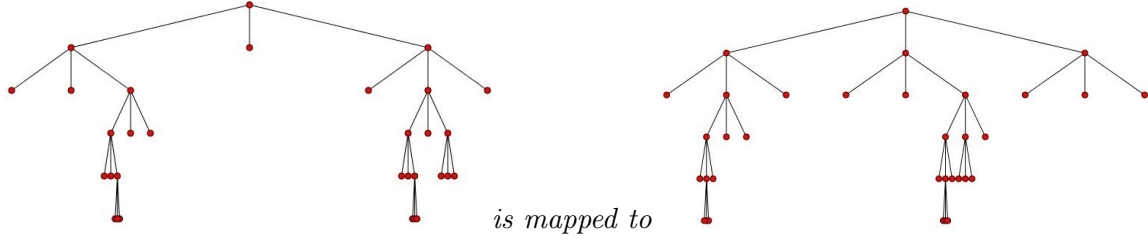
*Proof.* Once again, the proof that the replacement rule  $B_{t_5, t_9}(T)$  is a bijection is nearly identical to that of  $B_{t_1, t_2}(T)$ , with only one difference. In the proof that  $B_{t_5, t_9}(T)$  is onto, by applying  $B_{t_5, t_9}^{-1}(T) = B_{t_5, t_9}(T)$  to a set of lists avoiding  $t_9$  (that is, containing no pairs of consecutive 2's), we obtain a set of lists containing no pairs of consecutive 1's, thus avoiding  $t_5$ . □

### 6.4 $B_{t_6, t_7}$



Using list notation, a tree avoids  $t_6$  if none of its lists have a 1 followed by a 2; it avoids  $t_7$  if none of its lists have a 1 followed by a 3. Therefore, we define  $B_{t_6, t_7}(T) = B_{t_3, t_4}(T)$ . That is,  $B_{t_6, t_7}(T)$  replaces each 2  $T$ 's lists with a 3, and each 3 with a 2. (Because we originally defined  $B_{t_3, t_4}(T)$  on  $TS_3$  defining  $B_{t_6, t_7}(T)$  in this way is reasonable.)

**Example 6.7.**



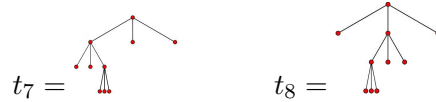
In list notation,  $B_{t_6, t_7}([1, 3, 1, 3], [3, 2, 1, 3], [3, 2, 3]) = [1, 2, 1, 2], [2, 3, 1, 2], [2, 3, 2]$ .

We will first show that  $B_{t_6, t_7}(T)$  sends trees avoiding  $t_6$  to trees avoiding  $t_7$ . The trees avoiding  $t_6$  are exactly the sets of lists that do not contain the sequence  $[1, 2]$  in any list. Similarly, trees avoiding  $t_7$  are exactly the sets that do not contain the sequence  $[1, 3]$  in any list. Thus, our function will map a set of lists not containing the sequence  $[1, 2]$  to a set of lists not containing the sequence  $[1, 3]$ . Because  $B_{t_6, t_7}(T)$  replaces every 2 with a 3, and every 3 with a 2, it achieves this goal, sending trees avoiding  $t_6$  to trees avoiding  $t_7$ .

**Theorem 6.8.**  $B_{t_6, t_7}$  is both one-to-one and onto.

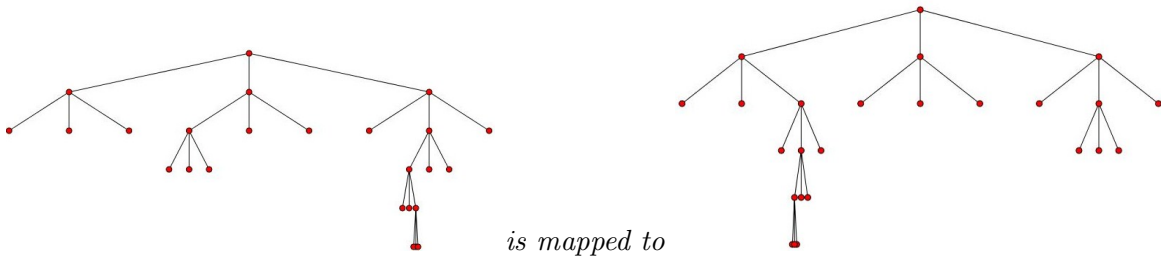
*Proof.* The proof that the replacement rule  $B_{t_6, t_7}(T)$  is a bijection is nearly identical to that of our first three bijections. The proof of one-to-one is the same as that of  $B_{t_3, t_4}(T)$ , and the proof of onto differs only in that by applying  $B_{t_6, t_7}^{-1}(T) = B_{t_6, t_7}(T)$  to a set of lists that avoids  $t_7$  (that is, not containing the sequence  $[1, 2]$ ), we will obtain a set of lists not containing the sequence  $[1, 3]$ , thus avoiding  $t_6$ . □

### 6.5 $B_{t_7, t_8}$



Using list notation, a tree avoids  $t_7$  if none of its lists have a 1 followed by a 3; it avoids  $t_8$  if none of its lists have a 2 followed by a 1. Therefore, we define  $B_{t_7, t_8}(T)$  to replace every 1 with a 2, every 2 with a 3, and every 3 with a 1.

**Example 6.9.**



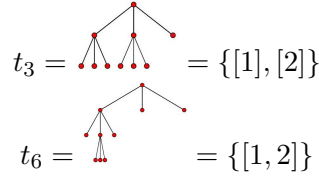
In list notation,  $B_{t_7, t_8}([1], [2, 1], [3, 2, 1, 3]) = [2], [3, 2], [1, 3, 2, 1]$ .

We will first show that  $B_{t_7, t_8}(T)$  sends trees avoiding  $t_7$  to trees avoiding  $t_8$ . The trees avoiding  $t_7$  are exactly the sets of lists that do not contain the sequence  $[1, 3]$  in any list. Similarly, trees avoiding  $t_8$  are exactly the sets that do not contain the sequence  $[2, 1]$  in any list. Thus, our function will map a set of lists not containing the sequence  $[1, 3]$  to a set of lists not containing the sequence  $[2, 1]$ . Because  $B_{t_7, t_8}(T)$  replaces every 1 with a 2, every 2 with a 3, and every 3 with a 1, it achieves this goal, sending trees avoiding  $t_7$  to trees avoiding  $t_8$ .

**Theorem 6.10.**  $B_{t_7, t_8}$  is a bijection.

*Proof.* In proving that the replacement rule  $B_{t_6, t_7}(T)$  is a bijection, we again have a nearly identical proof to that of the previous bijections. To prove that the bijection is one-to-one, however, we note that  $L_1 \in T_1$   $L_2 \in T_2$  are the same as  $L_3$  except with each 1, 2, and 3 in  $L_3$  replaced with a 3, 1, and 2, respectively. Also, when we apply  $B_{t_7, t_8}^{-1}(T) = B_{t_7, t_8}(T)$  to a set of lists avoiding  $t_7$  (that is, not containing the sequence  $[1, 3]$  in any of its lists), we will obtain a set of lists not containing the sequence  $[2, 1]$ , thus avoiding  $t_8$ . □

## 6.6 $B_{t_3, t_6}$



Consider the function  $B_{t_3, t_6}(T)$  defined by the following procedure applied each list  $L \in T$ :

1. If  $L$  contains no 1 followed by a 2, do nothing to  $L$ .
2. Otherwise, let  $L = [x_1, \dots, x_k, 1, 2, y_1, \dots, y_l]$  where the displayed  $[1, 2]$  is the first occurrence of  $[1, 2]$ . We map  $L$  to the pair of lists  $[x_1, \dots, x_k, 1], [x_1, \dots, x_k, 2, y_1, \dots, y_l]$ . So  $[x_1, \dots, x_k, 1]$  contains no occurrence of  $[1, 2]$ .
3. Iterate step 2 for each new list created until we have produced  $L_1, L_2, \dots, L_p$ , none of which contain a 1 followed by a 2.
4. Throw away any  $L_i$  that is a prefix of another one of the lists that we created.

**Example 6.11.** If  $T_1 = \{[1, 2, 3, 2, 3, 1, 1, 1, 2, 1]\}$ , then our first iteration maps  $T_1$  to  $[1], [2, 3, 2, 3, 1, 1, 1, 2, 1]$ .

Our second iteration gives  $[1], [2, 3, 2, 3, 1, 1, 1], [2, 3, 2, 3, 1, 1, 2, 1]$ .

Our third iteration gives  $[1], [2, 3, 2, 3, 1, 1, 1], [2, 3, 2, 3, 1, 1], [2, 3, 2, 3, 1, 2, 1]$ .

Our fourth iteration gives  $[1], [2, 3, 2, 3, 1, 1, 1], [2, 3, 2, 3, 1, 1], [2, 3, 2, 3, 1], [2, 3, 2, 3, 2, 1]$ .

Since  $[2, 3, 2, 3, 1, 1], [2, 3, 2, 3, 1]$  are both prefixes of  $[2, 3, 2, 3, 1, 1, 1]$  we get rid of them. So we have

$$B_{t_3, t_6}(T_1) = \{[1], [2, 3, 2, 3, 1, 1, 1], [2, 3, 2, 3, 2, 1]\}.$$

The following claim will be useful when we prove  $B_{t_3, t_6}$  is a bijection later in this section.

**Claim 2.** Let  $p_0$  be a prefix of length  $k$ , and let  $x$  be the number of consecutive 1's at the end of  $p_0$  (so if  $p_0 = [1, 2, 1, 3, 1, 1]$  then we would have  $x = 2$ ). If  $B_{t_3, t_6}(\{L\}) = \{L_1, \dots, L_r\}$  then we see that some  $L_i$  is of the form  $[p_0, 1]$  and all of the other  $L_j$ 's are of the form  $[p_1, 2, s_j]$ , where  $p_1$  is the first  $k - x$  terms of  $p_0$ .

proof We see this from step 2 in  $B_{t_3, t_6}(T)$  identifying the prefix before the first occurrence of  $t_6$  in  $L$  as occurring at the vertex given by  $p_0$ . The next few iterations of step 2 send each subsequent list, with  $[2, s_i]$  at its end, one level lower until we reach the vertex given by  $p_1$  and we have one fewer occurrences of  $t_6$  than before. Since we discard the prefixes of  $[p_0, 1]$  created in this process and we know that this was the lowest level occurrence of  $t_6$  in  $L$  we have shown that our  $L_j$ 's excluding  $L_i$  all have the prefix  $[p_1, 2]$ .

In order to prove that  $B_{t_3, t_6}$  is a bijection, first construct an inverse function,  $B_{t_3, t_6}^{-1}(T)$ , given by the following process:

- At each level of the tree  $T$  consider all occurrences of  $t_3$ ; that is, lists of the form  $[p_0, 1, s_0], [p_0, 2, s_2]$ , where  $p_0$  denotes a common prefix in the two lists and  $s_1, s_2$  are suffixes. Note that for each occurrence of  $t_3$  it is possible for there to be multiple lists of the form  $[p_0, 2, s_i]$ .
- At the first level we replace each occurrence of  $t_3$  at the vertex given by the path  $p_0$  we replace each  $[p_0, 2, s_i]$  with  $[p_0, 1, 2, s_i]$ . If there are multiple occurrences of  $t_3$  on the same level, then applying step 2 to one occurrence of  $t_3$  does not affect the lists denoting any other occurrence of  $t_3$ . Therefore, the order with which we apply this step to each occurrence of  $t_3$  at the  $i$ th level is irrelevant.
- Iterate step 2 at each consecutive level, beginning with the root.
- Discard any lists that are a prefix of another in  $T$ 's set of lists.

**Example 6.12.** For  $T_2 = \{[1], [2, 3, 2, 3, 1, 1, 1], [2, 3, 2, 3, 2, 1]\}$ , at the first level we have an occurrence of  $t_3$  given by  $[1] \cap [2, 3, 2, 3, 1, 1, 1]$  and from  $[1] \cap [2, 3, 2, 3, 2, 1]$ . So, from step 2, we replace  $[2, 3, 2, 3, 1, 1, 1], [2, 3, 2, 3, 2, 1]$  with  $[1, 2, 3, 2, 3, 1, 1, 1], [1, 2, 3, 2, 3, 2, 1]$ .

We now have the set  $\{[1], [1, 2, 3, 2, 3, 1, 1, 1], [1, 2, 3, 2, 3, 2, 1]\}$ ; step 3 requires that we check the next levels in order from lowest to highest and we see that  $t_3$  does not occur until the sixth level, and is given by the lists  $[1, 2, 3, 2, 3, \mathbf{1}, 1, 1], [1, 2, 3, 2, 3, \mathbf{2}, 1]$ . Thus, we replace  $[1, 2, 3, 2, 3, 2, 1]$  with  $[1, 2, 3, 2, 3, \mathbf{1}, 2, 1]$ .

With the third iteration of step 2, we replace  $[1, 2, 3, 2, 3, 1, 2, 1]$  with  $[1, 2, 3, 2, 3, \mathbf{1}, 1, 2, 1]$ . The fourth iteration replaces  $[1, 2, 3, 2, 3, 1, 1, 2, 1]$  with  $[1, 2, 3, 2, 3, \mathbf{1}, 1, 1, 2, 1]$ , and we are left with the set  $\{[1], [1, 2, 3, 2, 3, 1, 1, 1], [1, 2, 3, 2, 3, 1, 1, 1, 2, 1]\}$ . After applying step 4, we see that  $B_{t_3, t_6}^{-1}(T_2) = \{[1, 2, 3, 2, 3, 1, 1, 1, 2, 1]\}$ .

We note that  $B_{t_3, t_6}^{-1}(T_2) = T_1$  as expected from example 6.11.

We now begin our proof that  $B_{t_3, t_6}(T)$  is bijection. It is easy to see that  $B_{t_3, t_6}(T)$  does in fact map from trees avoiding  $t_3$  to trees avoiding  $t_6$ .  $B_{t_3, t_6}(T)$  is defined on trees avoiding  $t_3$  and it eliminates all occurrences of  $t_6$ . Likewise we see that  $B_{t_3, t_6}^{-1}(T)$  is defined on trees avoiding  $t_6$  and eliminates all occurrences of  $t_3$ . So  $B_{t_3, t_6}^{-1}(T)$  does in fact map from trees avoiding  $t_6$  to trees avoiding  $t_3$ . Now we must show that  $B_{t_3, t_6}^{-1}(B_{t_3, t_6}(T)) = T$ . First let us examine  $B_{t_3, t_6}(T)$ . We see that each occurrence of  $t_6$  is replaced with an occurrence of  $t_3$ . Furthermore we are given that no  $t_3$  occurs in a tree in the domain of  $B_{t_3, t_6}(T)$ . We claim that the only way for  $t_3$  to occur in  $B_{t_3, t_6}(T)$ , where  $T$  is an arbitrary tree avoiding  $t_3$ , is for our  $t_3$  in  $B_{t_3, t_6}(T)$  to be produced by an occurrence of  $t_6$ . This is easy to see, since we know that there are no copies of  $t_3$  in  $T$  before our mapping, and  $B_{t_3, t_6}$  either sends a list to itself or ‘‘splits’’ a list into two lists wherever a  $t_6$  occurs. Consider a list  $L \in T$ . By claim 2 we know that  $B_{t_3, t_6}(\{L\}) = \{L_1, \dots, L_r\}$  where some  $L_i$  (say  $L_1$ ) is of the

form  $[p_0, 1]$  and all other  $L_j$ 's are of the form  $[p_1, 2, s_j]$ . If  $x$  is the number of consecutive 1's at the end of  $p_0$  then we see that with  $x + 1$  iterations of step 2 of  $B_{t_3, t_6}^{-1}(T)$  we map our  $L_j$ 's to lists of the form  $[p_0, 1, 2, s_j]$  and  $L_1$  is discarded for being a prefix of the others. This was a valid step since we know that there could not be any occurrences of  $t_3$  in  $\{L_1, \dots, L_r\}$  that were not produced by a  $t_6$  when  $B_{t_3, t_6}(T)$  was applied to  $T$ . This same reasoning applies to each occurrence of  $t_6$  in  $T$ . Thus we see that  $B_{t_3, t_6}^{-1}(B_{t_3, t_6}(T)) = T$ . This concludes our proof that  $B_{t_3, t_6}(T)$  is a bijection.

**Claim 3.**  $B_{t_3, t_6}(T)$  preserves the numbers of leaves of  $T$ .

*Proof.* Step 2 in our bijection is the only step that changes the structure of  $T$ . Consider an arbitrary occurrence of  $t_6$  whose root is the vertex given by the prefix  $p_0$  in  $T$  such that there is no occurrence of  $t_6$  in  $p_0$ . Then step 2 in our bijection will map all lists with the prefix  $[p_0, 1, 2]$  to the list  $[p_0, 1]$  and lists with the prefix  $[p_0, 2]$ . As a result of no longer having any lists with the prefix  $[p_0, 1, 2]$ , we see that the vertex given by the path  $[p_0, 1]$  has one more leaf in  $B_{t_3, t_6}(T)$  than it did in  $T$  (namely its second child is a leaf, but was not a leaf in  $T$ ). However we also see that the vertex given by the path  $p_0$  has one less child that is a leaf as a result of having lists with  $[p_0, 2]$  as a prefix. There could not have already been a list in  $T$  with  $[p_0, 2]$  as a prefix since this would entail having an occurrence of  $t_3$  at the vertex given by the path  $p_0$ . Having The list  $[p_0, 1]$  does not add to or subtract from the number of leaves we have since it is a prefix of the list it replaces and thus creates no new vertices. With each iteration of step 2 on an occurrence of  $t_6$  this same reasoning holds. So we see that the number of leaves in  $B_{t_3, t_6}(T)$  is the same as the number of leaves in  $T$ .  $\square$

## 6.7 General Approaches to Bijections

In this section, we take what we have found from the previous six bijections, and begin generalizing this to a bijection between any two equivalent tree patterns. We state a bijection between certain tree patterns that (1) have the same avoidance generating function and (2) have only one  $m$ -leaf parent. To determine when this generalization applies to two distinct tree patterns, we first define a *super-pattern*. We conclude with a conjecture about further generalizations of such bijections.

### 6.7.1 Super Patterns

**Definition 6.13.** Consider a list  $L$  whose entires are members of  $\{1, 2, \dots, m\}$  (and thus represents an  $m$ -ary tree). A super pattern  $P$  of  $L$  is a list with elements of the set  $\{\alpha_i\}_{i=1}^m$ , when there exists a function  $f(\alpha_i)$  assigning each  $\alpha_i$  to a distinct integer 1 through  $m$  such that when  $f(\alpha_i)$  is applied to each element of  $P$  we obtain  $L$ . Also, if  $P$  is a super-pattern of  $L$ , we say that  $L$  is a pattern of the form  $P$ .

**Example 6.14.** If  $P = [\alpha_1, \alpha_1, \alpha_2, \alpha_1, \alpha_3]$ , then  $L_1 = [1, 1, 2, 1, 3]$ ,  $L_2 = [2, 2, 3, 2, 1]$  and  $L_3 = [1, 1, 3, 1, 2]$  are all patterns of the form  $P$ .

### 6.7.2 Bijections for Single List Patterns

**Theorem 6.15.** Let  $P$  be a list of arbitrary length whose elements are elements of  $\{\alpha_i\}_{i=1}^m$ . Also, let lists  $L_1, L_2$  be patterns of the form  $P$ , where  $L_1$  is given by a function replacing each  $\alpha_i$  with  $x_{\alpha_i}$ , each a distinct integer 1 through 3, and  $L_2$  is given by a function replacing each  $\alpha_i$  with  $y_{\alpha_i}$ , each a distinct integer 1 through 3. There exists a bijection,  $B_{L_1, L_2}(T)$ , mapping the set of trees avoiding  $\{L_1\}$  to the set of trees avoiding  $\{L_2\}$ , defined as the function mapping each  $x_{\alpha_i}$  to  $y_{\alpha_i}$  in each list in  $T$ .

**Example 6.16.** Consider two patterns  $L_1 = [1, 1, 2, 1, 3]$ ,  $L_2 = [2, 2, 3, 2, 1]$  of the form  $P = [\alpha_1, \alpha_1, \alpha_2, \alpha_1, \alpha_3]$ . There exists a bijection  $B_{L_1, L_2}(T)$  mapping the set of trees avoiding  $\{L_1\}$  to the set of trees avoiding  $\{L_2\}$ .  $B_{L_1, L_2}(T)$  is given by performing the following mapping in each list of  $T$ :

$$1 \rightarrow 2$$

$$2 \rightarrow 3$$

$$3 \rightarrow 1$$

To show that  $B_{L_1, L_2}(T)$  maps trees avoiding  $\{L_1\}$  to trees avoiding  $\{L_2\}$ , let  $T$  be a tree denoted by a set of lists that avoids  $\{L_1\}$ . By definition,  $T$  contains no list in which  $L_1$  occurs. Thus, when we apply the bijection described, we have  $B_{L_1, L_2}(T)$  to be a set of lists in which no list contains any instance of  $P_2$ , since  $P_2$  is  $P_1$  with each  $x_{\alpha_i}$  replaced with  $y_{\alpha_i}$ . This, of course, means that  $B_{L_1, L_2}(T)$  denotes a tree that avoids the pattern  $\{P_2\}$ .

**Theorem 6.17.** The general bijection  $B_{L_1, L_2}(T)$  is one-to-one, onto, and preserves the number of leaves of  $T$ .

*Proof.* First we show that  $B_{L_1, L_2}(T)$  is one-to-one. Note that it is important for the function that maps each  $\alpha_i$  to  $x_{\alpha_i}$  to map all  $\alpha_i$ 's, regardless of whether or not an  $\alpha_i$  actually occurs in the super-pattern  $P$ ; the same is true for functions mapping  $\alpha_i$ 's to  $y_{\alpha_i}$ 's.

Given  $T_1, T_2 \in TS_3$ . If  $B_{L_1, L_2}(T_1) = T_3 \in TS_3$  and also  $B_{L_1, L_2}(T_2) = T_3$ , then for each list  $M_3 \in T_3$  there exists lists  $M_1$  in  $T_1$  and  $M_2$  in  $T_2$  that are the same as  $M_3$  except with each  $y_{\alpha_i}$  in  $M_3$  replaced with  $x_{\alpha_i}$ . The note at the beginning of this proof is relevant here because, if  $B_{L_1, L_2}$  did not map every number in  $T_1$  and  $T_2$ , it may not have been clear whether some number in  $T_3$  came from mapping  $B_{L_1, L_2}(T)$  or was already in the original list. Because this is not the case, we have  $M_1 = M_2$ . Therefore,  $B_{L_1, L_2}(T)$  is one-to-one.

Now we show that  $B_{L_1, L_2}(T)$  is onto. To do this, we define an inverse function of  $B_{L_1, L_2}(T)$ , denoted by  $B_{L_1, L_2}^{-1}(T)$ , to be the function that maps each  $y_{\alpha_i}$  to  $x_{\alpha_i}$ . It is clear that  $B_{L_1, L_2}^{-1}(T)$  is defined on the sets of lists avoiding  $\{L_2\}$ . We also see that it follows that  $B_{L_1, L_2}^{-1}(B_{L_1, L_2}(T)) = T$ . Therefore,  $B_{L_1, L_2}(T)$  is onto.

Finally, we show that  $B_{L_1, L_2}(T)$  preserves the number of leaves of  $T$ . It is enough to show that for each internal vertex  $v_1$  in a given tree  $T$  there is a unique internal vertex  $v_2$  in  $B_{L_1, L_2}(T)$  that has the same number of leaves as  $v_1$ . Using list notation, there is a prefix  $p_1$  describing the path to  $v_1$  in  $T$ . Let  $x$  be the number of lists in  $T$  that have the prefix  $p_1$  followed by at least one additional number. Then  $v_1$  has  $3 - x$  children that are leaves. Under the bijection  $B_{L_1, L_2}(T)$ ,  $p_1$  is mapped to the prefix  $p_2$  of some list(s) given by our bijection. Because  $B_{L_1, L_2}(T)$  is one-to-one, nothing else is mapped to  $p_2$ , giving exactly  $x$  lists in  $B_{L_1, L_2}(T)$  with the prefix  $p_2$  followed by at least one number. Thus  $p_2$  denotes a path to an internal vertex  $v_2$  that will also have  $3 - x$  leaves since there are  $x$  lists that have the prefix  $p_2$ . This holds true for every vertex of  $T$ , proving that the number of leaves remains the same.  $\square$

## 7 Conclusion

Throughout this paper, we have investigated ternary trees, extending previous work in pattern avoidance. We began by finding the recurrence relations and generating functions by hand for

several simple ternary tree patterns. To make finding trees' avoidance sequences easier, however, we developed an algorithm, based on Rowland's algorithm for binary trees, to find the generating function for trees avoiding any given tree pattern. After exploring these sequences and their connections to other combinatorial objects, such as the Little Schroeder numbers, we classified the tree patterns, grouping together those with the same avoidance sequence. From here, we were able to find bijections between the sets of trees avoiding two equivalent tree patterns,  $t_i$  and  $t_j$ ; these allow us to transform any tree avoiding  $t_i$  into one that avoids  $t_j$ . By stating several bijections between specific pairs of tree patterns, we then generalized this to any two tree patterns of the same form (i.e. with the same super-pattern).

To extend our generalization, we state the following conjecture:

[INSERT CONJECTURE ON "CUTTING" SUPER-PATTERNS]

Areas for further research include:

1. Finding more relationships between pattern-avoiding ternary trees and other combinatorial objects:
  - Class 5 - Catalan Numbers, OEIS A000108
  - Class 7.1 - Little Schroeder Numbers, OEIS A001003
  - Class 7.2 - Number of modes of connections of  $2n$  points, OEIS A006605
2. Demonstrating the partitioning of all ternary tree patterns based on equivalence classes (that is, tree patterns with the same avoidance sequence).
3. Adapting and studying the ternary tree generating function algorithm, such that it counts the number of occurrences of a given tree pattern.

## 8 Appendix 1 - Table of Equivalence Classes

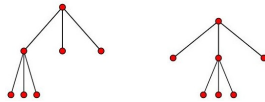
This appendix lists ternary trees with at most nine leaves, classifying them by their avoidance generating function and sequence. Listed at the beginning of each class is the first 20 terms (including zeros) of their avoidance sequence, or, where the Maple package could not produce the terms, an expression of  $a$ 's and  $x$ 's, where  $a = gft(x)$ . For brevity, left-right reflections are omitted.

*Class 5*

$$gft(x) = \frac{1 - \sqrt{1 - 4x^2}}{2x}$$

$$-a + x + a^2x = 0$$

[0, 1, 0, 1, 0, 2, 0, 5, 0, 14, 0, 42, 0, 132, 0, 429, 0, 1430, 0, 4862, ...]



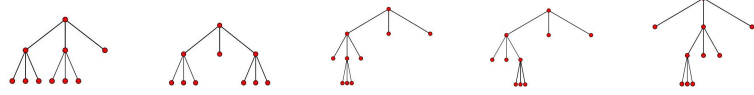


Class 7.1

$$gft(x) = \frac{(x^2 + 1) - \sqrt{(x^2 + 1)^2 - 8x^2}}{4x}$$

$$-a + x + 2a^2x - ax^2 = 0$$

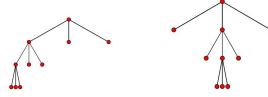
[0, 1, 0, 1, 0, 3, 0, 11, 0, 45, 0, 197, 0, 903, 0, 4279, 0, 20793, 0, 103049, ...]



Class 7.2

$$a - x - a^4x - a^2x = 0$$

[0, 1, 0, 1, 0, 3, 0, 11, 0, 46, 0, 207, 0, 979, 0, 4797, 0, 24138, 0, 123998, ...]



Class 9.1

$$-a + x + a^2x + a^6x + a^4x = 0$$

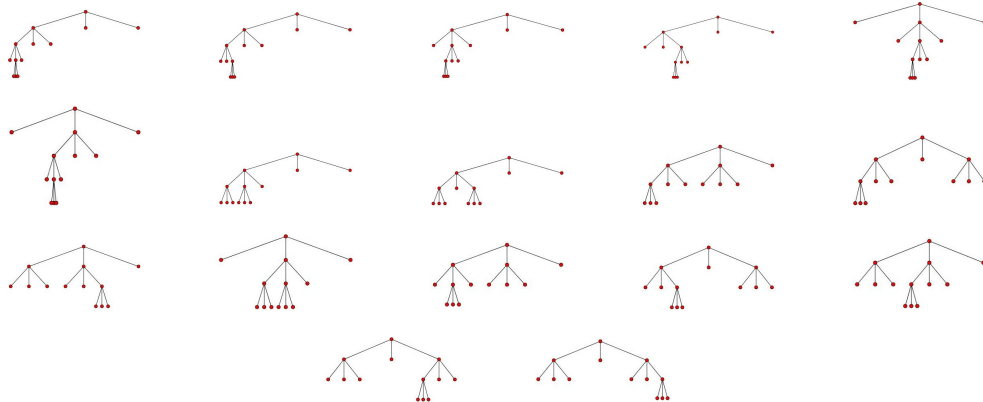
[0, 1, 0, 1, 0, 3, 0, 12, 0, 54, 0, 262, 0, 1337, 0, 7072, 0, 38426, 0, 213197, 0, ...]



Class 9.2

$$\pm(a^3x^2 + ax^2 - 2a^2x - x - a^4x + a) = 0$$

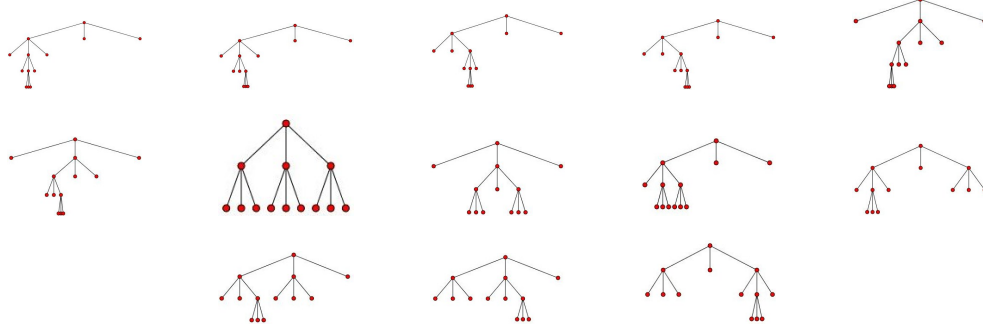
[0, 1, 0, 1, 0, 3, 0, 12, 0, 54, 0, 261, 0, 1324, 0, 6954, 0, 37493, 0, 206316, 0, ...]



Class 9.3

$$\pm(a - x - 3a^2x + 3ax^2 - x^3) = 0$$

[0, 1, 0, 1, 0, 3, 0, 12, 0, 54, 0, 261, 0, 1323, 0, 6939, 0, 37341, 0, 205011, 0, ...]



## 9 Appendix 2 - Maple Code

### 9.1 Maple Programs

In this appendix, we give the Maple code for the intersection operation and generating function algorithm described in section 5.

```

> interstree := proc (T1::list, T2::list)
option remember;
if T1 = [1, 0] then return T2
else if T2 = [1, 0] then return T1
end if end if;
return [1, interstree(T1[2], T2[2]), interstree(T1[3], T2[3]), interstree(T1[4],
T2[4]), 0]
end proc;
> Listt := proc (B::list)
local a, c, V;
option remember;
if B = [] then return [1, [1, 0], [1, 0], [1, 0], 0] end if;
if B[1] = 1 then return [1, Listt(B[2 .. nops(B)]), [1, 0], [1, 0], 0] end if;
if B[1] = 2 then return [1, [1, 0], Listt(B[2 .. nops(B)]), [1, 0], 0] end if;
if B[1] = 3 then return [1, [1, 0], [1, 0], Listt(B[2 .. nops(B)]), 0] end if
end proc;
> Sett := proc (A::set)
local c, k, t, F;
option remember;
if A = {} then return [1, 0] end if;
if A = {[[]]} then return [1, [1, 0], [1, 0], [1, 0], 0] end if;
F := Listt(A[1]);
for k from 2 to nops(A) do
F := interstree(F, Listt(A[k]))
end do
end proc;
> Av := proc (T::list, k::posint)
local a, x, c, p, B, S, t, i, m, n, g, U, V, L, Y, E, z;
option remember;

```

```

E := {}; U := {}; V := {};
z[0] := a[1, 0] = x+a[1, [1, 0], [1, 0], [1, 0], 0];
z[1] := a[1, [1, 0], [1, 0], [1, 0], 0] = a[1, 0]*a[1, 0]*a[1, 0]-a[op(interstree([1,
0], T[2]))]*a[op(interstree([1, 0], T[3]))]*a[op(interstree([1, 0], T[4]))];
U := U union {a[1, 0], a[1, [1, 0], [1, 0], [1, 0], 0]};
V := V union {a[op(interstree([1, 0], T[2]))], a[op(interstree([1, 0], T[3]))],
a[op(interstree([1, 0], T[4]))], a[1, 0], a[1, [1, 0], [1, 0], [1, 0], 0]};
E := E union {z[0], z[1]};
c := 2;
while V minus U ≠ {} do
B := V minus U;
for i to nops(V minus U) do
z[c] := B[i] = a[op(op(2, B[i]))]*a[op(op(3, B[i]))]*a[op(op(4, B[i]))]-a[op(interstree(op
B[i]), T[2]))]*a[op(interstree(op(3, B[i]), T[3]))]*a[op(interstree(op(4, B[i]),
T[4]))];
U := U union {B[i]};
V := V union {a[op(op(2, B[i]))], a[op(op(3, B[i]))], a[op(op(4, B[i]))], a[op(interstree
B[i]), T[2]))], a[op(interstree(op(3, B[i]), T[3]))], a[op(interstree(op(4, B[i]),
T[4]))]};
E := E union {z[c]};
c := c+1
end do;
end do;
m := nops(E);
g := eliminate(E, {seq(lhs[i](E[i]), i = 2 .. m)});
L := op(g[2]);
L := subs(a[1, 0] = sum(q[n]*x^n, n = 0 .. k), L);
Y := expand(L);
for i from 0 to degree(Y,x) do
p[i] := coeff(Y, x, i)
end do;
S := solve({seq(p[t] = 0, t = 0 .. k)}, {seq(q[t], t = 0 .. k)});
subs(S, [seq(q[t], t = 0 .. k)])
end proc;

```

## References

- [1] Rowland, Eric S. “Pattern Avoidance in Binary Trees,” *Journal of Combinatorial Theory*, Series A **117** (2010) 741-758.
- [2] Stanley, Richard P. *Enumerative Combinatorics*, New York: Cambridge University Press, 1999.
- [3] On-Line Encyclopedia of Integer Sequences. Published electronically, <http://oeis.org>, 2010.