POLYNOMIALS, BINARY TREES, AND POSITIVE BRAIDS

CHAD WILEY AND JEFFREY GRAY

Abstract. In knot theory, a common task is to take a given knot diagram and generate from it a polynomial. One method for accomplishing this is to employ a skein relation to convert the knot into a type of labeled binary tree and from this tree derive a two-variable polynomial. The purpose of this paper is to determine, in a simplified setting, which polynomials can be generated from labeled binary trees. We give necessary and sufficient conditions for a polynomial to be constructible in this fashion and we will provide a method for reconstructing the generating tree from such a polynomial. We conclude with an application of this theorem to a class of knots and links given by closed positive braids.

1. Introduction

Binary trees are a commonly used object in mathematics and computer science, appearing in problems of graph theory, probability, sorting and elsewhere. The problem discussed in this paper was inspired by a problem in knot theory, specifically the use of skein trees to compute knot polynomials. In such problems, a skein relation is used to recursively transform a knot diagram into a weighted sum of two simpler diagrams. Sufficiently simplified diagrams can then be replaced with constants, resulting in a polynomial. Since each knot diagram is transformed into two diagrams, this process can be formulated in terms of generating a binary tree, called a *skein tree*. A good example of a knot polynomial computed via skein relations is the bracket polynomial, introduced in [4].

Following the example of skein trees, this paper will consider binary trees whose edges and vertices are labeled and use such trees to define polynomials in two variables. The main intent of the paper is to show that this is a reversible process; given a two-variable polynomial, it is possible to determine whether that polynomial could have been generated by a binary tree and, moreover, decide which tree or trees generate that polynomial. As an application, we note that polynomials which are generated in this fashion are invariants of a particular class of knots, so this result allows us to determine which polynomials could appear as such an invariant.

We begin by discussing definitions and terminology relevant to the problem.

1.1. Definitions and Terminology.

In this paper we are concerned with *binary trees*, which are trees in which one vertex (the *root*) has degree 2 and every other vertex has degree 1 or 3. A simpler way to visualize a binary tree is to use the following construction. Begin with a single point, which we will call the root. Add two more vertices to the diagram, and connect each of them to the root with an edge. This process can be repeated as often as one likes, each time choosing a leaf in the tree and connecting two new vertices to it. We'll refer to this process as *branching*. Beginning with the root and performing the branching operation repeatedly produces a binary tree.

The *level* of a vertex in a binary tree is the number of edges in the path from the vertex to the root. The root has level 0 by definition. The *height* of a binary tree is the maximum level among all the vertices of the tree. A binary tree that contains the maximum number of vertices for its height will be referred to as a *fully branched* tree.

In many applications involving trees, it is helpful to give labels to the edges and/or vertices of the tree. Such a tree is called a *labeled tree*.

We will also borrow a bit of notation from family trees and call vertex p a *descendant* of a non-root vertex q if p has a higher level than q and the path from p to q does not include the root. We also refer to q as an *ancestor* of p in this situation. We define every non-root vertex to be a descendant of the root, and the root is an ancestor of every other vertex in the tree. The set of descendants of a vertex p together with the edges connecting them is called the *subtree* of p. Removing the subtree of a vertex is called *pruning* the vertex, and it can be thought of as the inverse operation to branching.

2. Polynomials From Trees

The basic construction for generating a two-variable polynomial from a binary tree is as follows. Start with a labeled binary tree. Each edge in the tree can have one of two possible labels. The actual labels used are not important; for convenience we'll label our edges either L or R depending on whether the edge is angled down and left or down and right when the tree is read from top to bottom.

We will also label the vertices of the tree. The root is always given the label 1. Every other vertex is labeled by tracing the path from the vertex back to the root and multiplying together all the L's and R's (which are considered to be commutative). Thus every vertex in the tree receives a label of the forms L^aR^b, where a and b are nonnegative integers.

Definition. The polynomial generated by a labeled binary tree is defined to be the sum of the labels of the leaves. We denote the coefficient of the term L^aR^b in the polynomial by $C_{a,b}$. If a polynomial can be generated by some labeled binary tree, we refer to that polynomial as a *tree polynomial*.

For example, $L^2+2LR+R^2$ is a tree polynomial. It is generated by the tree given in Figure 1.

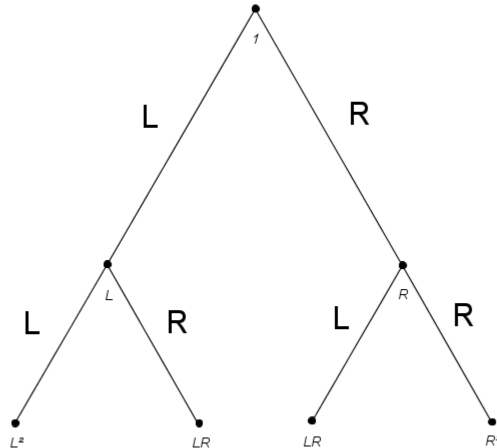

FIGURE 1. A labeled binary tree

3. The Main Theorem

The primary goal of this paper is to determine necessary and sufficient conditions for a two variable polynomial to be a tree polynomial. The most natural condition to begin with is based on the fact that any binary tree can be turned into a fully branched binary tree through the branching process described above. Since the polynomial generated by a fully branched binary tree of height n is $(L+R)^n$, we can phrase this property in polynomial terms as follows: if each term L^aR^b is multiplied by $(L+R)^{n-a-b}$, the resulting polynomial is $(L+R)^n$.

As a bookkeeping device, we rephrase this property in terms of a matrix equation.

Definitions. Let $P(L,R)$ be a two-variable polynomial of degree n. We define $\vec{v}$ to be the *coefficient vector* of P. That is, $\vec{v}$ is a column vector of length $\frac{(n+1)(n+2)}{2}$ whose entries are the coefficients of P, including zeros. Specifically, the entry of $\vec{v}$ that corresponds to L^aR^b is $C_{a,b}$.

We adopt the convention that in vector form the coefficients are listed in order of descending degree, and within each degree in descending powers of L. For clarity, we will often indicate which L^aR^b term each entry refers to next to the entry itself. For example, the polynomial $2L^2+5LR+R^2+3L+2$ would result in the following vector:

$$\begin{matrix} L^2 \\ LR \\ R^2 \\ L \\ R \\ 1 \end{matrix} \begin{bmatrix} 2 \\ 5 \\ 1 \\ 3 \\ 0 \\ 2 \end{bmatrix}$$

Definition. Define $\vec{b_n}$ to be the vector of length $n+1$ whose i^{th} entry is the binomial coefficient $\binom{n}{i-1}$.

Definition. Define A_n to be the following block matrix:

$$A_n = \left[\, A^{(n)} \mid A^{(n-1)} \mid \dots \mid A^{(1)} \mid A^{(0)} \,\right]$$

where $A^{(k)}$ is the $(n+1)\times(k+1)$ matrix whose ij entry is defined by

$$a_{i,j}^{(k)} = \begin{cases} \binom{n-k}{i-j}, & \text{if } j \le i \le j+n-k; \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

To help understand the matrix A_n, consider the matrix A_3 as an example.

$$A^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad A^{(1)} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 1 & 2 \\ 0 & 1 \end{bmatrix} \quad A^{(0)} = \begin{bmatrix} 1 \\ 3 \\ 3 \\ 1 \end{bmatrix}$$

$$A_3 = \begin{matrix} \\ L^3 \\ L^2R \\ LR^2 \\ R^3 \end{matrix} \begin{matrix} L^3 & L^2R & LR^2 & R^3 & L^2 & LR & R^2 & L & R & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 2 & 1 & 3 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{matrix}$$

The rows and columns of A_3 are labeled in a very specific way. Consider a fully branched labeled binary tree of height 3. Every vertex in that tree has a label that matches one of the columns of A_3. The entries in that column represent the leaves contained in the subtree whose root is the chosen vertex. For example, the vertex L^2 has a subtree of height one

with two leaves, L^3 and L^2R, and the column labeled L^2 encodes that information.

A simpler way of constructing the matrix A_n is to notice that the column labeled L^aR^b is just the $a+b^{\text{th}}$ row of Pascal's triangle with an extra b zeros at the top of the column and a zeros at the bottom.

Multiplying the coefficient vector of a polynomial on the left by the matrix A_n, where n is the degree of the polynomial, has the same effect as multiplying each L^aR^b term in the polynomial by $(L+R)^{n-a-b}$. By the discussion above, we conclude that the coefficient vector of a tree polynomial must satisfy $A_n\vec{x} = \vec{b_n}$.

While it is not hard to see that this condition is necessary for a specified polynomial to be a tree polynomial, it is not sufficient. The simplest counterexample is the polynomial $L^3 + 3LR + R^3$, which satisfies the equation above but cannot be generated by a binary tree because no binary tree has more than two nodes labeled LR. In addition to satisfying the matrix equation, we must ensure that the coefficients of the polynomial are not too large. To do this, we define the following function.

Definition. Given a two-variable polynomial, we define a function f by

$$f(a,b) = \sum_{\substack{0 \le i+j < a+b \\ i \le a \\ j \le b}} C_{i,j} \binom{a+b-i-j}{a-i}$$

where a and b are nonnegative integers that are not both zero.

The purpose of the function f is to calculate, given a tree polynomial, the number of vertices with the label L^aR^b that are "missing" from the generating tree (compared to a fully branched tree). It does this by searching for leaves whose level is less than $a+b$ and then counting how many L^aR^b vertices would have been in the subtree of that leaf if the tree were fully branched. This is best shown by an example.

Consider the tree polynomial $L^3 + 2L^2R + LR^2 + R$. The number of missing LR vertices in the tree that generates this polynomial is

$$\begin{aligned} f(1,1) &= C_{0,0}\binom{1+1-0-0}{1-0} + C_{1,0}\binom{1+1-1-0}{1-1} \\ &\quad + C_{0,1}\binom{1+1-0-1}{1-0} \\ &= 0\binom{2}{1} + 0\binom{1}{0} + 1\binom{1}{1} \\ &= 1. \end{aligned}$$

This is easily verified, since the tree generating $L^3 + 2L^2R + LR^2 + R$ is fully branched except for a pruning at the vertex labeled R. This pruning removes a single LR vertex.

It is very important to notice that the values of f can also be calculated even for polynomials which are not tree polynomials since f does not reference any tree diagrams directly. In fact, its primary use will be in deciding whether the coefficients of a polynomial are too large for it to be a tree polynomial.

Note that if $a = b = 0$, the formula fails since $C_{i,j}$ is only defined for nonnegative integers i and j. So we must define $f(0,0) = 0$ since no pruning can remove the root vertex.

Given these definitions, we can now state the main theorem.

Theorem 1. *Let $P(L,R)$ be a two-variable polynomial of degree n. Let $\vec{v}$ be the coefficient vector of P. Then P is a tree polynomial if and only if*

(1) *$\vec{v}$ is a solution to the matrix equation $A_n\vec{x} = \vec{b_n}$, and*
(2) *$0 \le C_{a,b} \le \binom{a+b}{a} - f(a,b)$ for all $a, b \ge 0$.*

Theorem 1 tells us that a two-variable polynomial with nonnegative coefficients is a tree polynomial if its coefficient vector is a solution to the given matrix equation, provided that the coefficients themselves are not too large. Both of these conditions can be easily verified by computer.

4. The Proof of Theorem 1

We've seen already that condition (1) of the main theorem is necessary. It is not hard to see that condition (2) is necessary as well. Since a binary tree can have no more than $\binom{a+b}{a}$ nodes labeled L^aR^b, the L^aR^b coefficient of $P(L,R)$, which we've labeled $C_{a,b}$, can be no larger than $\binom{a+b}{a}$. In practice, however, the upper bound for $C_{a,b}$ could be smaller; some nodes may be ineligible to appear as leaves because an ancestor node was pruned. The function f defined above counts precisely how many L^aR^b nodes are missing because of pruning at a higher level of the tree, and thus $\binom{a+b}{a} - f(a,b)$ is an upper bound for $C_{a,b}$. Thus it remains only to show that conditions (1) and (2) together are sufficient for a polynomial $P(L,R)$ to be a tree polynomial.

To prove sufficiency, we must start with a two-variable polynomial $P(L,R)$ with degree n whose coefficient vector satisfies properties (1) and (2) of Theorem 1 and construct a labeled binary tree whose polynomial is P. We will give an algorithm for constructing such a tree. Begin with a fully branched binary tree. Arrange the terms of the polynomial in ascending order of degree. Then, beginning with the first term and proceeding one term at a time, use the coefficient of the term to decide how many vertices

with that label to prune. For example, if you encounter the term $3L^5R^4$ you should choose any three terms in your current tree with labels L^5R^4 (the actual choice of vertices does not affect the outcome of the algorithm) and prune them before moving on to the next term.

By design, this algorithm produces a tree whose polynomial P' matches P except possibly for the coefficients of terms with maximal degree. A discrepancy could occur if at the nth level of the tree, the number of leaves labeled L^aR^b is greater than $C_{a,b}$. But the fact that both P and P' satisfy condition (1) and they agree in all the terms of degree less than n guarantees that $P = P'$, so this discrepancy will not occur. It remains only to show that the algorithm will not terminate too early by requiring more prunings than there are available nodes.

To accomplish this, we will use a pair of lemmas. Both of these lemmas refer to the branching operation on a coefficient vector, so recall that if the vector $\vec{v}$ shown below is a coefficient vector satisfying $p \geq 0$, $q \geq 0$, $r > 0$, and $a + b < n$, then the result of branching on the L^aR^b term is the vector $\vec{w}$.

$$\vec{v} = \begin{matrix} \\ L^{a+1}R^b \\ \\ L^aR^{b+1} \\ \\ L^aR^b \\ \\ \end{matrix} \begin{bmatrix} \vdots \\ p \\ \vdots \\ q \\ \vdots \\ r \\ \vdots \end{bmatrix} \qquad \vec{w} = \begin{matrix} \\ L^{a+1}R^b \\ \\ L^aR^{b+1} \\ \\ L^aR^b \\ \\ \end{matrix} \begin{bmatrix} \vdots \\ p+1 \\ \vdots \\ q+1 \\ \vdots \\ r-1 \\ \vdots \end{bmatrix}$$

Lemma 1. *Property (1) of Theorem 1 is invariant under branching. That is, if $\vec{v}$ satisfies property (1), then so does $\vec{w}$.*

Proof. Consider the matrix equation in condition (1) in terms of a weighted sum of the columns of A_n, where the weights are the entries of $\vec{v}$. Then the branching operation on an entry of $\vec{v}$ corresponding to L^aR^b amounts to replacing one copy of the L^aR^b column of A_n with the sum of one copy each of the $L^{a+1}R^b$ and L^aR^{b+1} columns. That this replacement is always valid is a simple consequence of the well-known recursive equation for the binomial coefficients:

$$\binom{c}{d} = \binom{c-1}{d-1} + \binom{c-1}{d}. \tag{2}$$

□

Lemma 2. *Property (2) of Theorem 1 is invariant under branching. That is, if $\vec{v}$ satisfies property (2), then so does $\vec{w}$.*

Proof. Suppose that the entries of $\vec{v}$ satisfy the bounds given in property (2). Since branching can only be performed on a term with a positive coefficient and since that term is the only one which decreases during branching, every entry of $\vec{w}$ will be nonnegative.

Now we show that for any entry in $\vec{w}$, the upper bound is never smaller than the upper bound for the corresponding entry in $\vec{v}$. Notice that in order for the upper bound of the entry $C_{s,t}$ to decrease, the value of $f(s,t)$ must increase. In the passage from $\vec{v}$ to $\vec{w}$, there are only two entries that increase: $C_{a+1,b}$ and $C_{a,b+1}$ (where $C_{a,b}$ is the entry where the branching operation was performed as in the example above). In order for these changes to affect the value of $f(s,t)$, we must have $s+t > a+b+1$ and either $a+1 \leq s$, $b \leq t$ or $a \leq s$, $b+1 \leq t$.

We'll consider the worst case scenario: both $a+1 \leq s$ and $b+1 \leq t$. In other words, both $C_{a+1,b}$ and $C_{a,b+1}$ appear as coefficients in the defining sum of f. Since these values go up when we go from $\vec{v}$ to $\vec{w}$, they will cause the value of $f(s,t)$ to increase. We can show that there is a corresponding decrease elsewhere in the sum that cancels out this increase.

Note that by the above inequalities, we have $s+t > a+b$, $a \leq s$, and $b \leq t$. So by definition $C_{a,b}$ will appear as a coefficient in the defining sum of f. The value of $C_{a,b}$ decreases as we go from $\vec{v}$ to $\vec{w}$, reducing the overall sum. Note that an increase of 1 in $C_{a+1,b}$ results in an increase of $\binom{s+t-(a+1)-b}{s-(a+1)}$ in the value of f and an increase of 1 in $C_{a,b+1}$ results in an increase of $\binom{s+t-a-(b+1)}{s-a}$.

By equation 2 we can see that the total increase in $f(s,t)$ will be $\binom{s+t-a-b}{s-a}$. But this is precisely the amount that $f(s,t)$ will decrease due to the decrease of 1 in the value of $C_{a,b}$. So there is no net change in f for this entry, and thus the upper bound will not change.

If only $C_{a+1,b}$ appears in the sum and not $C_{a,b+1}$ (i.e. if $a+1 \leq s$ and $t = b$), then the changes in $C_{a,b}$ and $C_{a+1,b}$ cause an increase of $\binom{s+t-(a+1)-b}{s-(a+1)}$ and a decrease of $\binom{s+t-a-b}{s-a}$, a net decrease in the value of $f(s,t)$ and thus a net increase in the upper bound for $C_{s,t}$. Similarly, if only $C_{a,b+1}$ appears in the sum and not $C_{a+1,b}$ there is a net increase in the upper bound. Thus the upper bounds described in property (1) can never decrease as a result of branching. Since most of the entries in $\vec{v}$ stay the same when branching, they will still satisfy the new upper bounds.

There are only three terms left to consider to complete the proof: $C_{a,b}$, $C_{a+1,b}$, and $C_{a,b+1}$. The entry $C_{a,b}$ decreases in the passage from $\vec{v}$ to $\vec{w}$, so its upper bound is still satisfied. The entries $C_{a+1,b}$ and $C_{a,b+1}$ both

increase by one. But as we have seen above, the decrease in $C_{a,b}$ will cause $f(a+1,b)$ and $f(a,b+1)$ to strictly decrease and thus the upper bounds for these entries will strictly increase. Thus these entries will satisfy the new upper bounds as well. Since all the entries of $\vec{v}$ and $\vec{w}$ are accounted for, the proof is complete. □

Now we can show that the algorithm will not terminate too early. Suppose that we have a polynomial $P(L,R)$ with degree n and with coefficient vector $\vec{v}$. Suppose that we are applying the algorithm to P so we have a tree T that is being pruned so that the leaves match the terms of P. For contradiction, assume we are now being asked to choose more vertices labeled L^aR^b than exist in T. Recall that terms with the lowest degree are handled first by the algorithm, so we can assume without loss of generality that all choices involving terms with degree lower than $a+b$ have been made. In tree terms, this means that all the pruning at levels less than $a+b$ has been done.

Suppose that m vertices with the label L^aR^b have been removed from T by previous pruning operations and that k such vertices remain in the tree. Then $k+m=\binom{a+b}{a}$, since $\binom{a+b}{a}$ is the number of L^aR^b vertices in a fully branched tree. Now say that the coefficient of L^aR^b in P is l, with $l>k$.

By Lemma 1, we can apply the branching operation to the coefficient vector $\vec{v}$ of P to create a new vector $\vec{v'}$ which is still a solution to the matrix equation but in which all the entries with degree less than $a+b$ have been reduced to zero. In doing so, the entries of the degree $a+b$ terms will go up. In tree terms, we are constructing a new tree T' from T which is fully branched up to level $a+b$. So we are restoring all of the missing L^aR^b vertices from T in T'. Since there were m such vertices missing in T, the entry labeled L^aR^b in $\vec{v'}$ will be $l+m$. But $l+m>k+m=\binom{a+b}{a}$. However, this contradicts Lemma 2 since $f(a,b)$ is always nonnegative.

Thus we can be sure that we are never asked to choose more vertices than exist in the current tree and so the algorithm will run to completion. Since the algorithm produces a binary tree whose polynomial matches the given polynomial, we conclude that conditions (1) and (2) are sufficient and Theorem 1 is proved.

5. Applications to Knot Theory

The original motivation for studying polynomials generated by labeled binary trees was to better understand the process of generating knot polynomials via skein relations. In particular, is it possible to reconstruct from a particular polynomial at least one knot with the given polynomial invariant? Theorem 1, while far from a complete solution to this problem, can at least help us address a limited version.

In order to generate a polynomial from a knot, we use an algorithm described by Bigelow in [2]. We have modified it here to highlight the relationship to binary trees. Given an oriented knot or link, express it as a closed braid (Alexander's Theorem [1] guarantees that this can always be done). We will consider this diagram to be the root vertex of a binary tree. Choose a basepoint as close as possible to the center of the closed braid. Starting at the basepoint, travel around the diagram following the given orientation. Each time a crossing is encountered for the first time, we label it "good" if we are traveling on the overcrossing strand and "bad" if we are on the undercrossing strand. If the crossing is good, we continue moving along the diagram. If the crossing is bad, we perform a branching operation on the current vertex of the tree. We add two edges and two child vertices to the current vertex, where the child vertices are represented by a pair of closed braid diagrams related to the current one by the relations in Figure 2. While these equations are just two forms of the same equation, it helps to see both since the first is applied to *positive* crossings (those in which the overcrossing strand runs bottom-left to top-right) and the second is applied to *negative* crossings (in which the overcrossing strand runs bottom-right to top-left).

$$\times = L \times + R \uparrow\uparrow$$

$$\times = L^{-1} \times - RL^{-1} \uparrow\uparrow$$

Figure 2. Two forms of the skein relation

The skein relation should be interpreted as a local operation. In one child vertex, the bad crossing has been replaced by the opposite crossing, leaving the rest of the diagram fixed. The edge connecting this child to the current vertex is labeled L if the bad crossing was positive, and L^{-1} if it was negative. In the other child vertex, the bad crossing has been replaced by a pair of uncrossed segments (removing the crossing entirely), again leaving the rest of the diagram fixed. The edge leading to this child vertex is labeled R if the bad crossing was positive or $-RL^{-1}$ if the bad crossing was negative. Each time the skein relation is applied, the algorithm is recursively applied to the new diagrams using the same basepoint.

We continue in this fashion until we arrive at the basepoint again. If all the crossings in the diagram have been accounted for, and thus all the crossings in the diagram are good, we can stop. If there are still crossings unaccounted for, then we have a link in which one component has been

traversed. Since all the crossings involving that component are good, we can move that component over the other components. Dilate that component so that it is farther away from the center than any other part of the braid. Then choose a new basepoint as close to the center as possible and begin the algorithm again.

Once the recursive portion algorithm is finished, we have a labeled binary tree whose vertices are closed braid diagrams and whose leaves are diagrams with only good crossings. As a final step of our braid algorithm, we give each leaf diagram a coefficient which is the product of the labels on each edge in the path from that leaf to the root, and then sum over all the leaves. This is the same process described in Section 2.

In this way, we can turn a given closed braid diagram into a formal sum of closed braid digrams whose crossings are all good. The coefficients in this sum are products of the coefficients that appear in the skein relation. In [2], it is shown that a trace function can be applied to each of these diagrams, resulting in the HOMFLY polynomial, a well-known polynomial invariant. To better match the setup of Theorem 1, we instead map each diagram in the formal sum to 1. We also restrict our attention to closed *positive braids*, those whose crossings are all positive (so that only the top equation in Figure 2 is ever used during the algorithm). Given these restrictions, the output of the algorithm will always be a tree polynomial.

The final result of the algorithm is the tree polynomial generated by this tree: $L + LR + R^2$.

The interesting connection between this algorithm and Theorem 1 is that tree polynomials are invariants of a limited class of knot and link diagrams.

Theorem 2. *Let K be the class of closed positive braid 3-braid diagrams which represent knots or links of braid index 3. Then the tree polynomial produced by the algorithm above is an invariant of K.*

Proof. In [2], it is shown that the output of the algorithm is invariant under braid isotopy. By the Markov Theorem Without Stabilization [3], it suffices to show that the output of the algorithm is invariant under 3-braid flype moves. This is verified in [5]. □

Taken together, Theorems 1 and 2 give us a way of determining precisely which polynomials appear as invariants of knots and links in K.

A long-term goal is to find a way to reconstruct from a polynomial a knot or link with that polynomial invariant. While we cannot do that at this time, we can use properties of binary trees to determine some properties of knots or links which generate a given polynomial. For example, suppose B is a closed positive 3-braid in K which generates the tree polynomial $P(L, R)$. Precisely one term of P has the form $L^m R^0$. We know this because there is only one path in the binary tree which generates P that

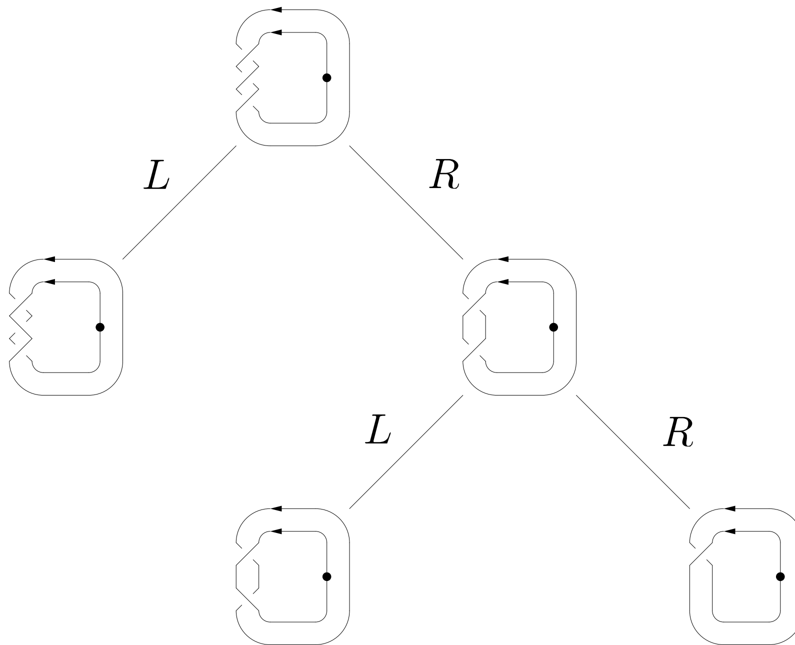

FIGURE 3. The skein tree produced by applying the braid algorithm to the trefoil knot.

contains no edges labeled R. In terms of the closed braid algorithm, this path represents replacing each bad crossing with a negative crossing. So the total number of bad crossings in B is m.

Label the strings in the original (unclosed) braid from left to right as 1, 2, and 3. Then in the closed braid, since all the crossings are positive a bad crossing occurs during the algorithm if and only if we are traveling from string i to $i+1$ if the orientation is counter-clockwise, or i to $i-1$ if the orientation is clockwise. Since we have only three strings available, in order to have m bad crossings there must also be at least $m-2$ good crossings in the diagram (provided m is at least 3) or else we run out of strings. Similarly, if the diagram had more than $m+2$ good crossings we would run out of strings. Thus we have proved the following result.

Theorem 3. *Suppose B is a closed positive 3-braid in K with c crossings, and suppose the braid algorithm applied to B results in the polynomial $P(L,R)$. If P contains the term L^m, then*

$$\max(m-2,0) \leq c \leq m+2.$$

In terms of future results, a long term goal would be to develop a method for reconstructing the original knot or link from a skein tree. Theorem 1 is a small step in that direction. A reasonable next step would be to incorporate the second version of the skein relation into Theorem 1 by including edges labeled L^{-1} and $-RL^{-1}$.

References

[1] J. W. Alexander, *A lemma on systems of knotted curves*, Proceedings of the National Academy of Sciences, **9.3** (1923), 93–95.

[2] S. Bigelow, *Braid groups and Iwahori-Hecke algebras*, Problems on maping class groups and related topics, Vol. 74, Proc. Sympos. Pure Math., 285–299, Amer. Math. Soc., Providence, RI, 2006.

[3] J. S. Birman and W. W. Menasco, *Stabilization in the braid groups*, I. MTWS. Geom. Topol., 10:413–540 (electronic), 2006.

[4] L. H. Kauffman, *State models and the Jones polynomial*, Topology, **26.3** (1987), 395–407.

[5] C. Wiley, *Nugatory Crossings in Closed 3-Braid Diagrams*, Ph.D. thesis, University of California, Santa Barbara, 2008.

MSC2010: 05C05, 05C31

Key words and phrases: Binary trees, Skein relations, Knot polynomials, Positive braids

Department of Mathematics, Computer Science, and Economics, Emporia State University, Emporia, KS 66801
E-mail address: cwiley1@emporia.edu

Department of Mathematics, Santa Barbara City College, Santa Barbara, CA 93101
E-mail address: jpgray4@pipeline.sbcc.edu