Computing the Arrow Polynomial

Kumud Bhandari

May 5, 2009

McKendree University

Abstract

Determining if two knots are not equivalent in an efficient manner is important in the study of knots. The arrow polynomial, which is calculated from a virtual knot diagram and is invariant under the Reidemeister moves, can be used to determine if two knots are not equivalent and determine a lower bound on the virtual crossing number. In this paper, we present the necessary data structures and algorithms to represent a link diagram on a computer and calculate the arrow polynomial.

1 Introduction

We see knots in our daily life; for example, everyday we tie our shoelaces into a knot. Inspired by different types of knots that appear around us, mathematicians study an area of mathematics called *knot theory*, which provides the tools to represent, study and classify knots similar to ones found in our daily life.

Imagine that one day you decided to glue the two ends of your shoelace together after you tied a knot as shown in figure 1. This results in a piece of string with no loose ends and which cannot be untangled without using a pair of scissors to cut it. The resulting structure is known as a *trefoil*, the second simplest type of knot [1]. The simplest knot is the *unknot* or the *trivial knot*, which is simply a closed loop with no tangling present. Knot theorists are interested in studying knots that cannot be untangled (untied) into a circle without cutting it first. Formally, a *knot* is defined as a mapping of the circle into three dimensional space[1].

In the study and classification of knots, being able to determine efficiently if two knots are not equivalent is important. Hence, we present a method that can be used to differentiate between two virtual knots quickly using a computer, and avoiding tedious calculations by hand. First, we introduce oriented virtual knots. Second, we present data structures for representation and manipulation of oriented knots. Finally, we show how an arrow polynomial is calculated using a computer and how it is utilized to determine if two knots are equivalent due to its invariant property under Reidemeister moves.

1.1 Diagrammatic Representation of a Classical Knot

If we think of string as having no thickness (its cross-section being a single point), then a knot is a closed curve in space that does not intersect itself anywhere. To simplify the visualization

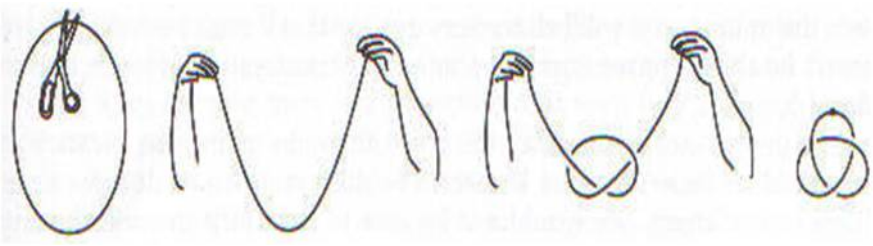


Figure 1: Forming a trefoil from a string [1]

and the manipulation of a knot, it is represented by its projection on a plane as shown in figure 2 (think of a knot as casting shadow on vertical plane when light is shone on it horizontally). The places where the projection intersects itself are called the *crossings* of the projection. The strand of the string passing underneath is represented by the interrupted line and the strand passing over by the continuous line. An uninterrupted section of the projection is called an *arc*; the endpoints of an arc occur at classical crossings [1]. The trefoil is a *three-crossing* knot because every projection of the trefoil contains at least three crossings.

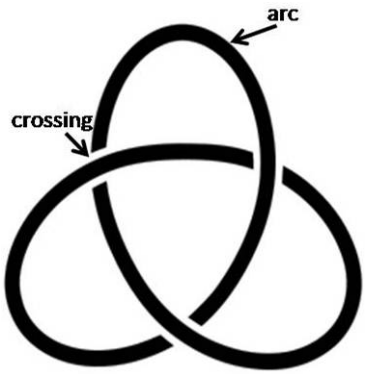


Figure 2: A projection of a trefoil

1.2 Virtual Knots

Virtual knots are a generalization of the classical knots. Virtual knots have an extra type of crossing called virtual crossings. This type of crossing appears because virtual diagrams are non-planar [2]. Louis H. Kauffman, a famous knot theorist, who introduced virtual knot theory, once made the following comment:

> *"... The idea is that the virtual crossing is not really there...If I draw a non-planar diagram in the plane, it necessarily acquires virtual crossing" – Louis H. Kauffman*

A virtual crossing is represented by a small circle around the crossing as shown in figure 3. The crossing itself is represented by two solid continuous lines.

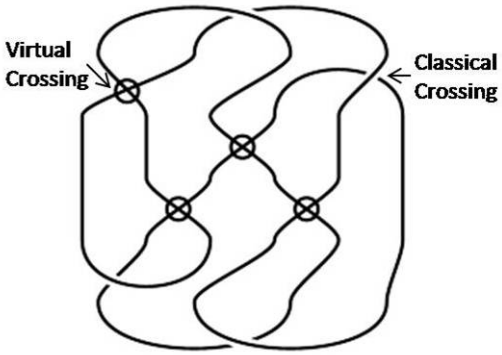


Figure 3: A representation of a virtual knot

1.3 Reidemeister Moves

Two different projections of the same knot are shown in figure 4. This occurs due to different arrangements of a knot in three dimensional space or different projections of the knot— consider how your shadow changes throughout the day. The Reidemeister moves form a planar calculus that accounts for the different projections of a virtual knot. Figure 5 shows all seven Reidemeister moves. The classical Reidemeister moves involve only the classical crossings. In the virtual Reidemeister moves, types I, II, and III involve only virtual crossings whereas type IV involves both virtual and classical crossings. Moreover, the moves are considered local, meaning that it is assumed that the projection remains unchanged except for the crossings on which we apply the Reidemeister moves. If two virtual knot diagrams differ by a sequence of Reidemeister moves, then the two diagrams represent the same virtual knot. Hence, the Reidemeister moves establish equivalence classes of virtual knot diagrams. Figure 6 shows how we obtain the knot diagram shown on the right hand side of figure 4 from the diagram shown on the left hand side using the Reidemeister moves.

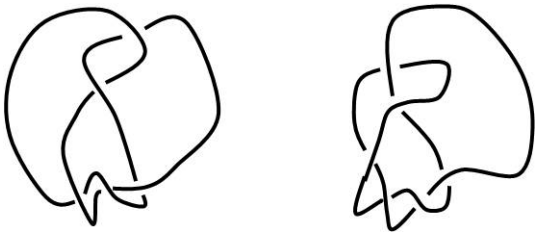


Figure 4: Two projections of the same knot.

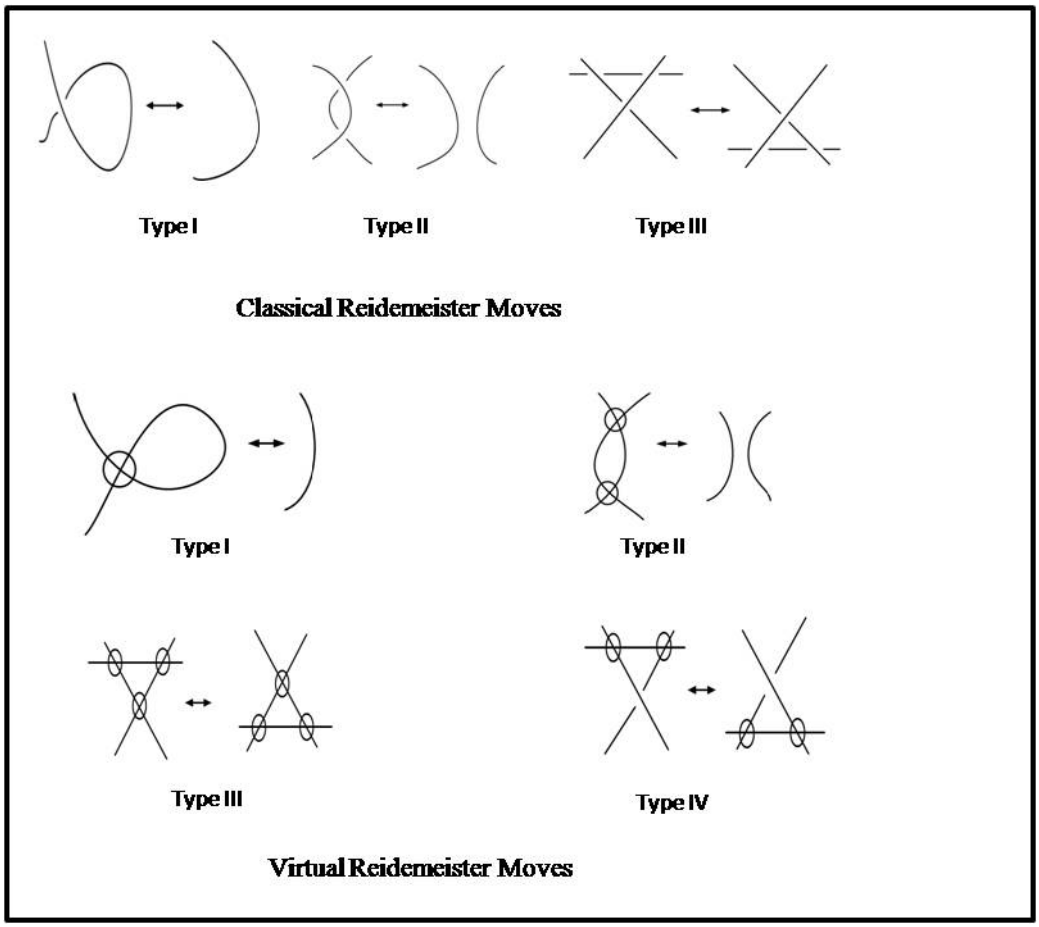


Figure 5: The Reidemeister moves

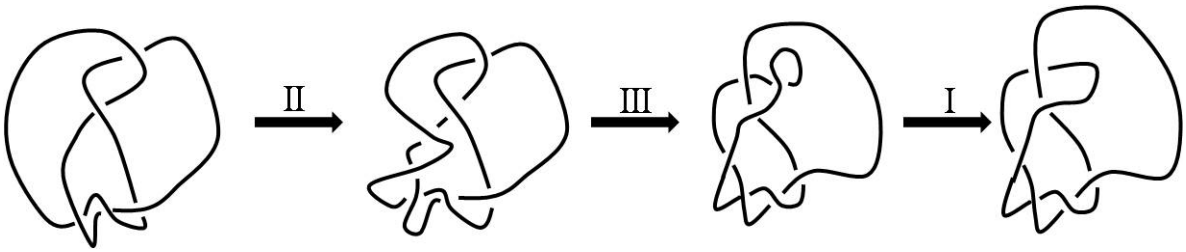


Figure 6: Demonstration of the use of the Reidemeister moves

2 Motivation

The early development of knot theory was motivated by its application in other areas of science, particularly chemistry. In the 1880s, when William Thompson (Lord Kelvin), Scottish mathematician and chemist, hypothesized the structure of matter as knots floating in ether, it aroused the interest of many mathematicians in this field. Today, knot theory is used in the study of DNA's double helical structures and DNA knottings, a phenomenon that destroys DNA's functionality. Additionally, it is used in statistical mechanics to represent mathematical

models of molecules, atoms and other particles subjected to force. In these applications of knot theory, being able to determine efficiently whether two knotted structures are not equivalent is very important. As we saw in section 1.3, one can use Reidemeister moves to determine if two diagrams represent the same knot. However, this process becomes increasingly tedious as the number of crossings in the diagram grows. Therefore, we explore an invariant called the arrow polynomial to examine the equivalence of virtual knot diagrams. This invariant is calculated from the diagram of a virtual knot and is invariant under all Reidemeister moves except classical Reidemeister move I. If two virtual knot diagrams yield different arrow polynomials, then neither diagram can be obtained from the other by applying the classical Reidemeister moves II and III, and the virtual Reidemeister moves. This means that the two projections do not belong to the same equivalence class, and therefore, do not represent the same virtual knot.

3 Oriented Knots

An oriented virtual knot diagram is formed by arbitrarily decorating a virtual knot diagram with an arrow as shown in figure 7. This arrow assigns an orientation to the knot; there are two possible such orientations. Informally, we can consider this as a direction of travel along the knot [3]. To determine the local orientations of the arcs at any crossing, one can trace the arrow around the knot.

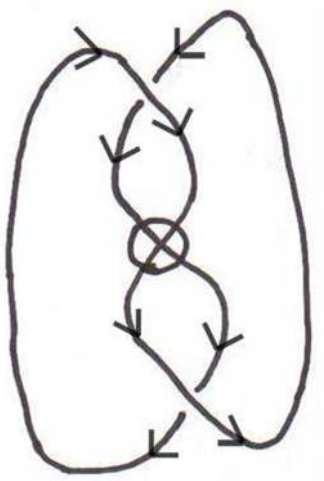


Figure 7: An oriented virtual knot diagram

3.1 Types of Crossings in an Oriented Virtual Knot Diagram

Due to the orientation of the virtual knot diagram, two different types of crossings arise depending on the local orientation of the arcs at the classical crossings as shown in figure 8 [3]. The crossings are called either positive or negative crossings based on this information. The type of crossing is distinguished by using the *right-hand* rule.

3.2 Data Structure to Represent an Oriented Virtual Knot:

We calculate the arrow polynomial of a virtual knot by manipulating (reducing) one of its oriented virtual knot diagrams to a collection of closed curves. Hence, to design a computer program that computes the arrow polynomial, we need a data structure that represents an

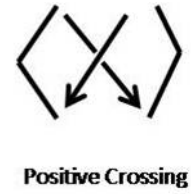


Figure 8: Types of crossings in an oriented knot diagram

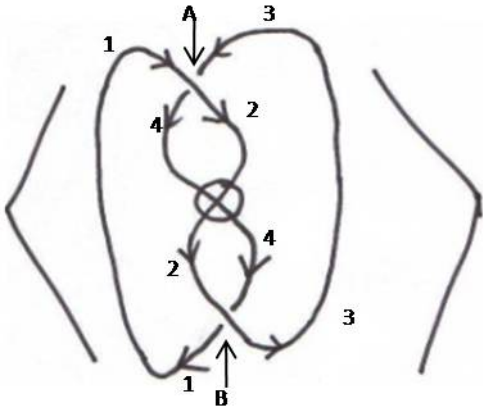


Figure 9: A labeled oriented virtual knot diagram

oriented virtual knot diagram. To construct this data structure, we first label an oriented virtual knot diagram as shown in figure 9 using the following rules:

1. Label the portion of an arc between two classical crossings with a positive integer.
2. Every portion of an arc between any two classical crossings will have a unique positive integer label associated with it. Arcs do not terminate at virtual crossings.

For the purpose of calculating an arrow polynomial of a virtual knot, we are only concerned with the manipulation of classical crossings. Therefore, virtual crossings are not represented explicitly. We represent each classical crossing by the following data structure:

crossing type (+/-)	*integer1*	*integer2*	*integer3*	*integer4*

where

$$crossing\ type = \begin{cases} 0 & if\ -\ crossing \\ 1 & if\ +\ crossing \end{cases},$$

and

integer1,...,integer4

is a sequence of integers formed by enumerating the labels that surround the crossing clockwise starting at the bottom end of the over passing arc. For example, the two classical crossings from the virtual knot in figure 9 are represented as follows:

- Crossing A is a positive crossing.

$$\boxed{0}\boxed{2}\boxed{4}\boxed{1}\boxed{3} \Longrightarrow C_A = [0 \mid 2, 4, 1, 3]$$

- Crossing B is a negative crossing.

$$\boxed{0}\boxed{3}\boxed{1}\boxed{2}\boxed{4} \Longrightarrow C_B = [0 \mid 3, 1, 2, 4]$$

We then represent a virtual knot with n classical crossings by the following data structure:

$$\boxed{C_0}\boxed{C_1}\boxed{.}\boxed{.}\boxed{.}\boxed{C_{n-1}}$$

where $C_0, C_1, ..., C_{n-1}$ are the classical crossings represented by the data structure described above. That is, a virtual knot is represented by a vector of classical crossings. For example, the virtual knot shown in figure 9 can be represented as follows:

$$K = \boxed{C_A}\boxed{C_B} \Longrightarrow \left\langle \begin{array}{c} C_A \\ C_B \end{array} \right\rangle.$$

Remark 3.1. *A given virtual knot can be represented in more than one way in this data structure because the orientation and labelling is chosen arbitarily.*

4 Arrow Polynomial

The arrow polynomial is calculated from an oriented virtual knot diagram by manipulating each classical crossing present in the diagram according to its type and then evaluating a collection of closed loops that result from such manipulation. This polynomial is invariant under all Reidemeister moves except classical Reidemeister move I. If two virtual knot diagrams yield two different arrow polynomials, then the two virtual knots are not equivalent under the classical Reidemeister moves II and III and all virtual Reidemeister moves. This polynomial also allows us to compute a lower bound on the virtual crossing number. In this section, we describe in detail how to compute this polynomial using a computer.

4.1 Oriented Expansion

In order to obtain a state of the diagram (a collection of closed loops), we expand each classical crossing in the oriented virtual knot diagram as shown in figure 10 [3]. In this figure,

each term on the right hand side of the expansion is called a *horizontal smoothing* if it contains a pair of nodal arrows and a *vertical smoothing* otherwise. Because the smoothed arcs contain orientation information and possibly nodal arrows, we require a new data structure to represent this information.

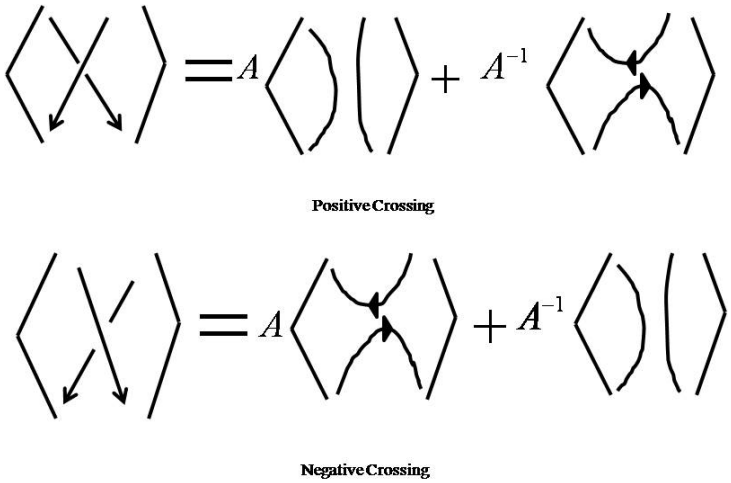


Figure 10: The oriented state expansion for calculating an arrow polynomial

4.2 Data Structure to Represent Oriented Expansion

The arcs in both horizontal and vertical smoothings are represented by the following data structure:

isOriented	*Integer1*	*Integer2*

where

$$isOriented = \begin{cases} True & if\ a\ nodal\ arrow\ present(oriented) \\ False & if\ a\ nodal\ arrow\ is\ absent \end{cases}$$

$$Integer1, Integer2 = \begin{cases} sequence\ of\ labels\ taken\ in\ the\ direction\ of\ nodal\ arrow\ if\ present \\ list\ of\ labels\ representing\ the\ endpoints\ if\ nodal\ arrow\ not\ present \end{cases}$$

Using this data structure, we represent the expansion of positive and negative crossings as follows:

- Positive Crossing:

$$[1|Integer1, Integer2, Integer3, Integer4] =$$
$$A \left\langle \begin{array}{c} [False|Integer2, Integer1] \\ [False|Integer3, Integer4] \end{array} \right\rangle + A^{-1} \left\langle \begin{array}{c} [True|Integer3, Integer2] \\ [True|Integer1, Integer4] \end{array} \right\rangle$$

- Negative Crossing:

$$[0|Integer1, Integer2, Integer3, Integer4]$$
$$= A \left\langle \begin{matrix} [True|Integer4, Integer3] \\ [True|Integer2, Integer1] \end{matrix} \right\rangle + A^{-1} \left\langle \begin{matrix} [False|Integer3, Integer2] \\ [False|Integer4, Integer1] \end{matrix} \right\rangle$$

Figure 11 shows the expansion of individual classical crossings in the virtual knot diagram shown in figure 9. This expansion is represented as follows:

$$C_A = [0 \mid 2, 4, 1, 3] = A \left\langle \begin{matrix} [True|3, 1] \\ [True|4, 2] \end{matrix} \right\rangle + A^{-1} \left\langle \begin{matrix} [False|1, 4] \\ [False|3, 2] \end{matrix} \right\rangle$$

$$C_B = [0 \mid 3, 1, 2, 4] = A \left\langle \begin{matrix} [True|4, 2] \\ [True|1, 3] \end{matrix} \right\rangle + A^{-1} \left\langle \begin{matrix} [False|2, 1] \\ [False|4, 3] \end{matrix} \right\rangle$$

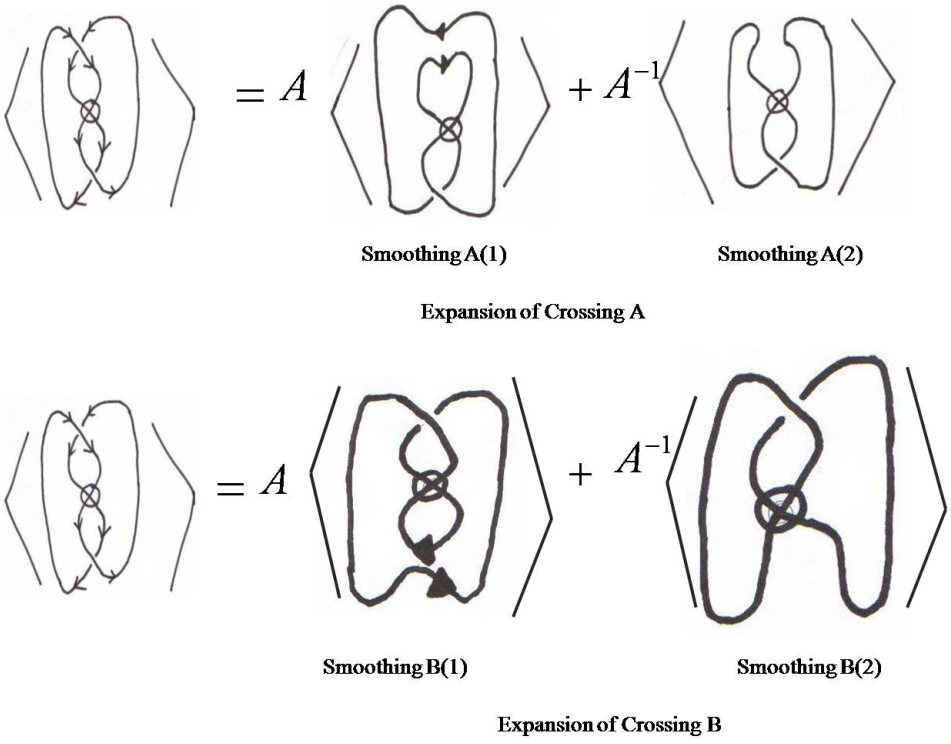


Figure 11: Expansion of individual crossings in the oriented virtual knot diagram of figure 9

4.3 Derivation of States

Once we apply the expansion discussed in section 4.1 recursively to every classical crossing, we obtain a weighted sum consisting of collections of closed loops. A collection of closed loops is referred to as a state. For a virtual knot with n classical crossings, we obtain 2^n states. Figure 12 shows all the possible states and their weights for the virtual knot shown in figure 7. Table 1 shows these states represented in pseudocode.

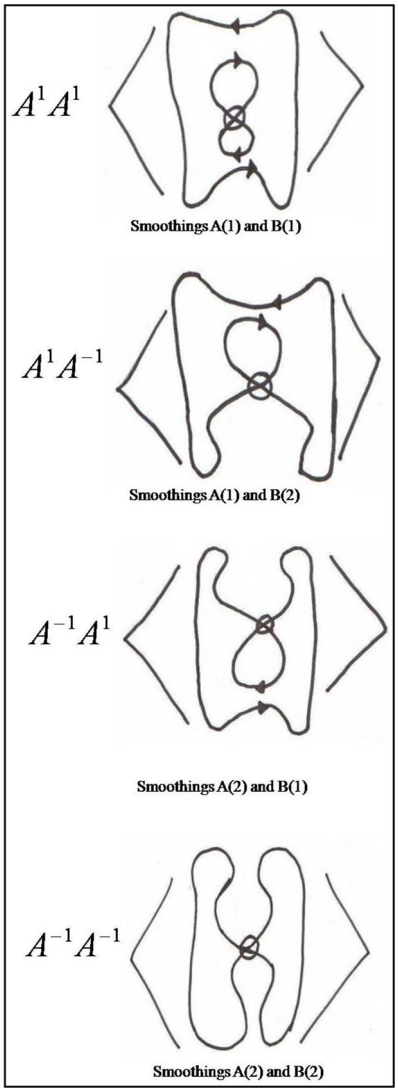


Figure 12: All possible combinations of smoothings for the crossings in figure 11

Smoothings Applied	State Representation
A(1) and B(1)	$A^1A^1 \left\langle \begin{matrix}[True\|3,1]\\[True\|4,2]\\[True\|4,2]\\[True\|1,3]\end{matrix} \right\rangle$
A(1) and B(2)	$A^1A^{-1} \left\langle \begin{matrix}[True\|3,1]\\[True\|4,2]\\[False\|2,1]\\[False\|4,3]\end{matrix} \right\rangle$
A(2) and B(1)	$A^{-1}A^1 \left\langle \begin{matrix}[False\|1,4]\\[False\|3,2]\\[True\|4,2]\\[True\|1,3]\end{matrix} \right\rangle$
A(2) and B(2)	$A^{-1}A^{-1} \left\langle \begin{matrix}[False\|1,4]\\[False\|3,2]\\[False\|2,1]\\[False\|4,3]\end{matrix} \right\rangle$

Table 1: Representations of states in pseudocode

4.4 Reduction of States

In figure 12, we see the virtual knot diagram reduced to all possible states. To determine the arrow polynomial, we will simplify and evaluate the states, adding the result to form the arrow polynomial. These states are a collection of loops and each loop may contain nodal arrows. Using the following rule, we reduce the number of nodal arrows to obtain invariance under Reidemeister moves:

Two adjacent closed arrows cancel if they are both oriented in the same direction [3].

For the data structure described above, we use algorithm 1 to reduce the states of the closed loops. Depending on the location and orientation of the nodal arrows, two adjacent arcs may be reduced to one arc. This possible reduction can be categorized by five cases, of which only four cases (case I, II, III, IV) are reducible as shown in table 2. The sequence(s) of *Integers* in an arc without the nodal arrow(s) can be altered to suit the need of the reduction.This algorithm exhaustively searches for the four reducible cases among the adjacent arcs and reduces them accordingly. The algorithm stops when each state is either a single arc with no nodal arrow present or only contains one or more pairs of irreducible adjacent arcs as shown in Table 2, Case V.

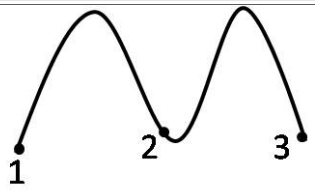
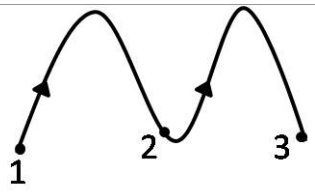
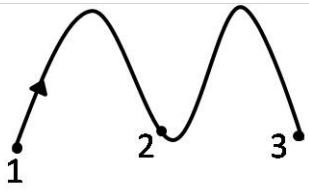
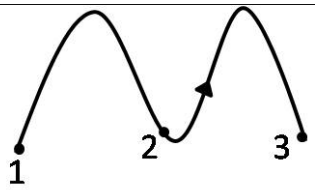
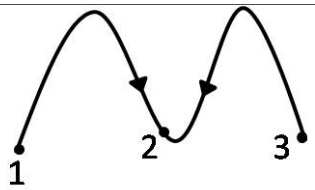
Case	Diagrammatic Representation	Pseudocode Representation	Reduced Pseudocode Representation
I		$[False\|1,2]$ $[False\|2,3]$	$[False\|1,3]$
II		$[True\|1,2]$ $[True\|2,3]$	$[False\|1,3]$
III		$[True\|1,2]$ $[False\|2,3]$	$[True\|1,3]$
IV		$[False\|1,2]$ $[True\|2,3]$	$[True\|1,3]$
V		$[True\|1,2]$ $[True\|3,2]$	*Irreducible*

Table 2: Different arrangements of nodal arrows within two adjacent arcs

Algorithm 1 The algorithm to reduce states

ALGORITHM *ReduceState* ($arcs[0...n]$)

//Input: An array $arcs[0...n]$ containing n arcs representing states obtained from applying expansion and represented using the data structure described in section 4.2.

//Output: An array $reduced[0...m]$ containing m arcs representing reduced states.

repeat

$further\,\mathrm{Re}ductionPossible \leftarrow false$

for $i, j \in \mathrm{Z}$ **and** $0 \le i, j < n$,

if $\exists arcs[i], arcs[j]$ **such that**

Case 1: $arcs[i].oriented = false, arcs[j].oriented = false$ **and** $arcs[i].label[1] = arcs[j].label[0]$, **then**

$arcs[i].label[1] \leftarrow arcs[j].label[1]$

$arcs[j] \leftarrow null$

$further\,\mathrm{Re}ductionPossible \leftarrow true$

Case 2: $arcs[i].orientation = true, arcs[j].orientation = true$ **and** $arcs[i].label[1] = arcs[j].label[0]$, **then**

$arcs[i].label[1] \leftarrow arcs[j].label[1]$

$arcs[j] \leftarrow null$

$further\,\mathrm{Re}ductionPossible \leftarrow true$

Case 3: $arcs[i].orientation = true, arcs[j].orientation = false$ **and** $arcs[i].label[1] = arcs[j].label[0]$, **then**

$arcs[i].label[1] \leftarrow arcs[j].label[1]$

$arcs[j] \leftarrow null$

$further\,\mathrm{Re}ductionPossible \leftarrow true$

Case 4: $arcs[i].orientation = false, arcs[j].orientation = true$ **and** $arcs[i].label[1] = arcs[j].label[0]$, **then**

$arcs[j].label[0] \leftarrow arcs[i].label[0]$

$arcs[i] \leftarrow null$

$further\,\mathrm{Re}ductionPossible \leftarrow true$

until ($further\,\mathrm{Re}ductionPossible = false$)

copy remaining strands from $arcs[0...n]$ **to** $reduced[0...m]$

return reduced

Using algorithm 1, we reduce the states from table 1 obtained in section 4.3 as shown in table 3.

States	Reduction	Reduced States
$A^1A^1\left\langle \begin{matrix}[True\|3,1]\\ [True\|4,2]\\ [True\|4,2]\\ [True\|1,3]\end{matrix}\right\rangle$	$\begin{cases}3\to 1\to 1\to 3\\ 4\to 2\\ 4\to 2\end{cases}$	$A^2\left\langle \begin{matrix}[True\|3,3]\\ [True\|4,2]\\ [True\|4,2]\end{matrix}\right\rangle$
$A^1A^{-1}\left\langle \begin{matrix}[True\|3,1]\\ [True\|4,2]\\ [False\|2,1]\\ [False\|4,3]\end{matrix}\right\rangle$	$\begin{cases}4\to 3\to 3\to 1\\ 4\to 2\to 2\to 1\end{cases}$	$A^0\left\langle \begin{matrix}[True\|4,1]\\ [True\|4,1]\end{matrix}\right\rangle$
$A^{-1}A^1\left\langle \begin{matrix}[False\|1,4]\\ [False\|3,2]\\ [True\|4,2]\\ [True\|1,3]\end{matrix}\right\rangle$	$\begin{cases}1\to 4\to 4\to 2\to 3\\ 1\to 3\end{cases}$	$A^0\left\langle \begin{matrix}[True\|1,3]\\ [True\|1,3]\end{matrix}\right\rangle$
$A^{-1}A^{-1}\left\langle \begin{matrix}[False\|1,4]\\ [False\|3,2]\\ [False\|2,1]\\ [False\|4,3]\end{matrix}\right\rangle$	$\{1\to 4\to 4\to 3\to 2\to 2\to 1$	$A^{-2}\langle[False\|1,1]\rangle$

Table 3: Reduction of States

4.5 Evaluation of States:

Once the states are reduced as described in section 4.4, each state consists of either a single arc without a nodal arrow or one or more pairs of irreducible adjacent arcs with nodal arrows (Table 2, Case V). Suppose the number of arrows remaining in the loop, C, is m. Note that m will always be even. Then, the value of the loop is represented by $< C >$, and

$$< C >= k_n \text{where } n = \tfrac{m}{2}. \text{ If } n = 0 \text{ then } < C >= 1[3].$$

Now, we evaluate the state by multiplying together the values of the loops in the state. Suppose evaluation of the state is represented by $< S >$. Then,

$$< S >= \prod_{all\ loops} < C > [3].$$

For the reduced states represented in section 4.4, we use algorithm 2 to determine the number of loops and the value of the states. This algorithm first detects all the loops with no nodal arrows present, represented by $[False|Integer1, Integer1]$. Next, it detects loops with more than one arrow and counts the number of arcs with arrows that it comes across while traversing the loop one full cycle. We apply this algorithm to the reduced states in table 3 to obtain the following results:

$$A^2\left\langle \begin{matrix}[True|3,3]\\ [True|4,2]\\ [True|4,2]\end{matrix}\right\rangle \Rightarrow \#\ of\ loops = 2,\ < S >= k_0k_1 = k_1$$

$$A^0\left\langle \begin{matrix}[True|4,1]\\ [True|4,1]\end{matrix}\right\rangle \Rightarrow \#\ of\ loops = 1,\ < S >= k_1 = k_1$$

$$A^0\left\langle \begin{matrix}[True|1,3]\\ [True|1,3]\end{matrix}\right\rangle \Rightarrow \#\ of\ loops = 1,\ < S >= k_1 = k_1$$

$$A^{-2}\langle [False|1,1]\rangle \Rightarrow \#\, of\ loops = 1,\ < S >= k_0 = 1$$

Algorithm 2 The algorithm to determine number of loops and evaluate states

```
ALGORITHM determineLoopsAndStates (arcs[0...n-1])
//Input: An array arcs[0...n-1] of n arcs representing reduced states obtained by using
        algorithm 1.
//Output: An integer array result[0...m] where m is the number of classical crossings in a knot,
         result[0] contains the number of loops and result[1]...result[m] contains the power
         of the variables k_1, k_2,...k_m respectively resulting from the state evaluation.
    for i ← 0 to n-1 do
        if arcs[i].label[0] = arcs[i].label[1], then
            result[0] ← result[0]+1
            arcs[i] ← null
    for i ← 0 to n-1 do
        first ← arcs[i].label[0]
        second ← arcs[i].label[1]
        arrowCount ← 1
        do
            for j ← i+1 to n-1 do
                if first = arcs[j].label[1] then
                    first ← arcs[j].label[0]
                    arrowCount ← arrowCount+1
                    arcs[j] ← null
                else if first = arcs[j].label[0] then
                    first ← arcs[j].label[1]
                    arrowCount ← arrowCount+1
                    arcs[j] ← null
                else if last = arcs[j].label[0] then
                    last ← arcs[j].label[1]
                    arrowCount ← arrowCount+1
                    arcs[j] ← null
                else if last = arcs[j].label[1] then
                    last ← arcs[j].label[0]
                    arrowCount ← arrowCount+1
                    arcs[j] ← null
        until (first = last)
        result[0] ← result[0]+1
        if 1 ≤ arrowCount/2 ≤ m then
            result[arrowCount/2] ← result[arrowCount/2]+1
    return result
```

4.6 Terms of the Arrow Polynomial:

The arrow polynomial of the oriented virtual knot with diagram K is given by [3]:

$$\langle K\rangle_A = \sum_S A^{\alpha-\beta} d^{|S|-1} < S >$$

where α and β is the number of smoothings with coefficient A and A^{-1}respectively in the formation of the state, d is equal to the quantity $(-A^2 - A^{-2})$, $|S|$ is the number of loops in the state, and $< S >$ is the state evaluation described in section 4.5.

For the oriented virtual knot diagram in figure 9 whose state evaluation is presented in 4.5, the arrow polynomial is

$$< K >_A = A^2 d^{2-1} k_1 + A^0 d^{1-1} k_1 + A^0 d^{1-1} k_1 + A^{-2} d^{1-1}.$$

Simplifying, we obtain

$$< K >_A = A^2(d)(k_1) + k_1 + k_1 + A^{-2}.$$

Replacing $d = -A^2 - A^{-2}$,we obtain

$$< K >_A = A^2(-A^2 - A^{-2})k_1 + k_1 + k_1 + A^{-2}$$

$$= (-A^4 - 1)k_1 + 2k_1 + A^{-2}.$$

Finally, we conclude

$$< K >_A = (-A^4 + 1)k_1 + A^{-2}. \quad (4.6.1)$$

Figure 13 shows an oriented knot diagram obtained by changing every classical crossing present in the virtual oriented knot diagram of figure 7. The two positive classical crossings replaces the two negative classical crossings in figure 7 . The arrow polynomial for the resulting diagram shown in figure 13 is

$$< K >_A = A^2 + (1 - A^{-4})k_1. \quad (4.6.2)$$

By comparing equations (4.6.1) and (4.6.2), we conclude that the arrow polynomial also succeeds in distinguishing two knot diagrams where one is obtained from another by simply changing the classical crossings.

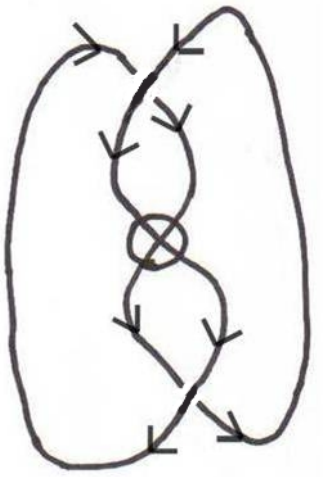


Figure 13: An oriented knot diagram resulting from changing each classical crossing of the virtual oriented knot diagram in figure 7

5 Conclusion

In this paper, we showed how to compute the arrow polynomial using a computer. This polynomial helps us to determine when a pair of virtual knots are not equivalent without manipulating the knot diagrams via the Reidemeister moves. We present the some results generated by a computer program written in Java (a high level programming language) in Appendix A.

References

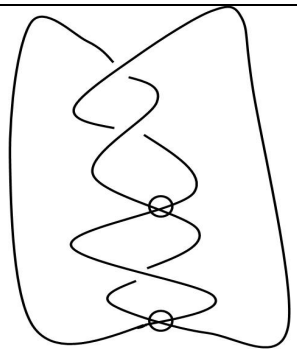
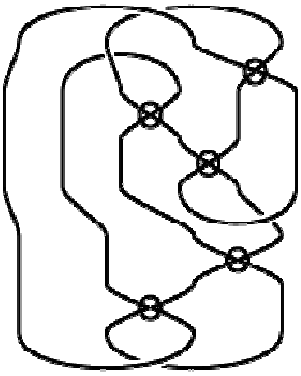
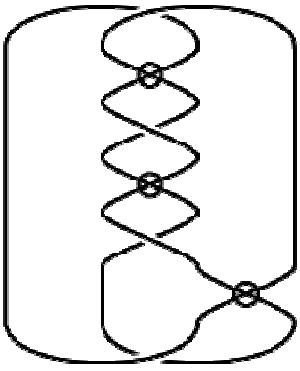
[1] Colin C. Adams, *The Knot Book: An Elementary Introduction to the Mathematical Theory of Knots.* Providence, RI: American Mathematical Society, 2004, pp. 1-95.

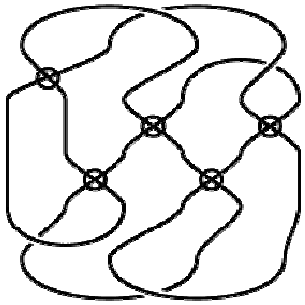
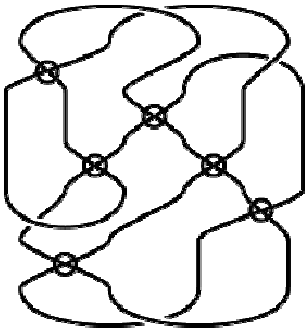
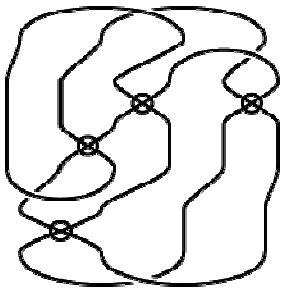
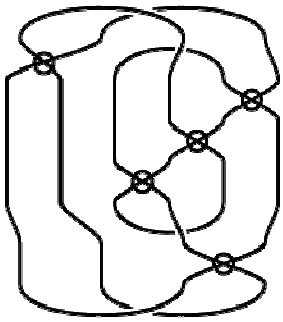
[2] L.H. Kauffman, "Virtual Knot Theory," *European Journal of Combinatorics,* vol. 20, pp. 663-691, 1999.

[3] H.A. Dye and L.H. Kauffman, "Virtual Crossing Number and the Arrow Polynomial," Preprint. [Online]. Available: arXiv:0810.3858v1, http://www.arxiv.org. [Accessed November 5, 2008]

Appendix A

This appendix contains the arrow polynomials for some four and five crossing knot computed using the program implemented in Java.

 **Figure 1: Virtualized Trefoil**	$< Virtual\ Trefoil >_A = A^{-5}k_1^2 - A^3k_1^2 - A^{-5}$
 **Figure 2: Virtual Knot 4.19**	$< Knot\ \ 4.19 >_A = A^4 + k_2 + 1 - A^4k_1^2 - k_1^2$
 **Figure 3: Virtual Knot 4.26**	$< Knot\ \ 4.26 >_A = A^2k_1 - A^{-2}k_1k_2 - A^2k_1k_2 + 1 + A^{-2}k_3$

 **Figure 4: Virtual Knot 4.33**	$< Knot \ \ 4.33 >_A = k_1 - A^4 k_1 + A^{-2}$
 **Figure 5: Virtual Knot 4.52**	$< Knot\ 4.52 >_A = -A^6 + 2A^2 k_1 + 3k_1 - 2A^{-2} + A^{-2} k_1 - A^2 + A^{-2} k_2 - 2A^{-4} - 1 + A^{-4} k_2$
 **Figure 6: Virtual Knot 4.66**	$< Knot\ 4.66 >_A = A^6 k_1 + A^{-2} k_1^3 + 1 + A^6 k_1^3 + 2A^2 k_1^3 - A^{-4} k_1^2 + k_2 - A^2 k_1 - A^2 k_1 k_2 - A^{-2} k_1 k_2 - k_1^2 + A^{-4}$
 **Figure 7: Virtual Knot 4.95**	$< Knot \ \ 4.95 >_A = -A^4 k_3 + A^{-2} + k_3$