EVALUATING KNOT THEORY ALGORITHMS FOR THE JONES POLYNOMIAL

EVAN ASHOORI, PETER BAI, AND HANSEN SHIEH

ABSTRACT. The Jones polynomial is a widely used knot invariant that was first discovered by Vaughan Jones in the 1980s during his investigation of von Neumann algebras. The computation of the Jones polynomial is known to be #P-hard and NP-hard, and the most efficient current classical implementations have sub-exponential time complexities. In this paper, we introduce knots, links, knot invariants, and the Jones polynomial as a knot invariant. After this, we explore and present implementations of various algorithms, including several that are provably or likely sub-exponential, for calculating the Jones polynomial. Finally, we implement and benchmark several algorithms for computing the Jones polynomial, and we partially implement an approach using the Hecke algebra, identifying normalization/substitution issues that remain unresolved. This has immediate applications in tabulation efforts for 21 and higher crossing knots, which use invariants such as the Jones polynomial to remove duplicate entries. Improving the speed of this computation allows knot theorists to tabulate higher crossing knots much faster. By making implementations of these algorithms open source, we hope to accelerate tabulation efforts.

1. INTRODUCTION

A *knot* is a smooth embedding of the circle in $\mathbb{R}^3$. More plainly, it is a closed loop of rope. A knot diagram is a regular projection of a knot in the plane in the sense that no three points on the knot project to a single point on the plane, and any two points on the knot which project to the same point are labeled to identify which point lies above the other. The *Reidemeister moves* are changes to knot diagrams that emulate all deformations made to three-dimensional knots. Hence, using Reidemeister moves, one can turn one projection of a knot into any other. One of the main areas of research in knot theory is *knot invariants*, which are mathematical structures (numbers, booleans, polynomials, groups, etc.) associated to knot diagrams which remain unchanged after Reidemeister moves. As a result, they are the same for all projections of a given knot. Different invariants allow us to prove that two knot diagrams are distinct under ambient isotopy. Thus, combining many different invariants allows us to distinguish different knots from one another, aiding in knot tabulation. Currently, knot tabulation has been done up to 20 crossing prime knots (knots which cannot be represented as the connected sum of two smaller knots) [Thi25], and computationally expensive knot invariants present a challenge when extending this tabulation to the many billions of 21 crossing prime knots. A wide variety of invariants have been used in knot theory including numerical values, booleans, and polynomials [Ber23]. However, the distinguishing power varies from invariant to invariant. There are, in fact, perfect invariants, that can distinguish every single knot; in particular, the fundamental quandle is a perfect invariant up to orientation reversal, but its computation is far too expensive to have practical applications to tabulation efforts [Mil22]. Like the fundamental quandle, many other invariants are also slow to compute, and have exponential time or even slower algorithms. It thus remains to find a knot invariant which is both fast to compute and able to distinguish between a large number of knots.

The Jones polynomial is one of the most well-known knot invariants, and a naïve recursive algorithm is simple enough to construct, but with a time complexity that is exponential in the number of crossings of the knot [Jon05]. Its computation is known to be #P-Hard [Kup14] and NP-Hard [PP93], but there are known quantum algorithms [AJL06] that additively approximate the Jones polynomial, doing so in polynomial time. Lacking a quantum computer, we restrict

Date: June 18th, 2025.

ourselves to classical algorithms, and in recent decades many alternative methods to compute the polynomial have been explored. In 1996, El-Misery and El-Horbaty provided an alternative algorithm for computing the Jones polynomial using the Hecke algebra [EMEH96]. In 1999, a skein-template algorithm was developed by Gouesbet, Meunier-Guttin-Cluzel, and Letellier [GMGCL99]. Additionally, one of the more recent algorithms, presented by Ellenberg et al. in 2014, uses tangles to efficiently calculate the Jones polynomial [ENSS14]. Though the algorithm by Ellenberg et al. is asymptotically the fastest, as it requires exponential time in the square root of the number of crossings of a knot, it fails with virtual knots. The relative size of implied constant factors were also unknown prior to the open-source implementation of all algorithms. All three algorithms had their own potential drawbacks and advantages. Some of these algorithms apply to virtual knots (knots in 3-manifolds other than $\mathbb{R}^3$), and others do not. Namely, [Mag24]'s approach extends to virtual knots because it makes no assumptions that the knot is in $\mathbb{R}^3$. [GMGCL99]'s approach also extends to virtual knots. The program for [GMGCL99] found at [MABS25] can be used for virtual knots. We are also exploring whether [EMEH96]'s approach can be extended to handle virtual knots via a modification of the algebra to include virtual generators.

Moreover, the language and architecture in which an algorithm is implemented can have substantial limitations on the size of its inputs and runtime. Many libraries such as SageMath and Python use arbitrary precision integers, which results in slower calculations than fixed-width arithmetic in programming languages such as C or Java. For our implementations, we picked the C language due to its portability, history, and speed. Moreover, several C compilers such as GCC and Clang are some of the most highly optimized compilers among all languages. We have elected to use fixed-width arithmetic in our implementations to speed up the computation, with either 32-bit or 64-bit integer and double-precision floating point data types. Thus, for exponential algorithms, we may be limited to knots with up to around 32 or 64 crossings. This is not a serious limitation for our purposes, though, since we are aiming to tabulate knots with around 21 crossings.

In this paper, we first define the Jones polynomial in full in Section 2. In Section 3, Section 4, Section 5, and Section 6 we present C implementations of the tangle algorithm, skein-template algorithm, and Hecke algebra algorithm, respectively. The code for the C implementations can be found at [MABS25]. By making our implementations and benchmarks open source, we hope to aid in tabulation efforts for 21-crossing and higher knots.

2. Jones Polynomial Definition

Definition 2.1. A *knot* is a smooth embedding of the circle S^1 into $\mathbb{R}^3$. A *link* is a non-empty collection of disjoint circles embedded in $\mathbb{R}^3$. A *knot diagram* is a projection of a knot in $\mathbb{R}^3$ onto $\mathbb{R}^2$ such that no three points on the knot map to the same point in $\mathbb{R}^2$, and portions of the diagram which lie directly above other portions are made apparent through gaps left in the diagram. Analogously, a *link diagram* is a projection of a link in $\mathbb{R}^3$ onto $\mathbb{R}^2$ with no three points on the link projecting to the same point in $\mathbb{R}^2$, and gaps are left in the diagram in the same manner.

Example 2.2. Figure 1 shows a diagram of the unknot, the simplest knot, while Figure 2 shows the trefoil knot.

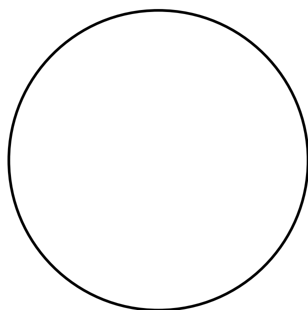

Figure 1: The unknot

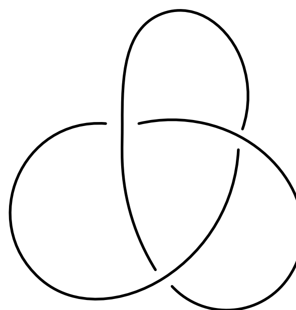

Figure 2: The trefoil knot

Definition 2.3. The three *Reidemeister moves* represent the elementary deformations which can be applied to diagrams of knots and links as outlined in [Rei27].

Type 1. Reidemeister move 1A twists part of an arc in the diagram, creating an extra arc. Reidemeister move 1B does the opposite by removing a twist in the diagram, reducing the number of arcs by one. Figure 3 shows moves 1A and 1B applied to a section of a knot diagram.

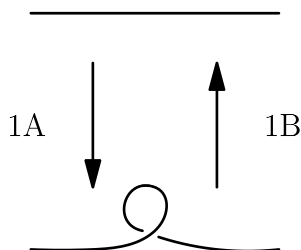

Figure 3: The first type of Reidemeister move

Type 2. Reidemeister move 2A begins with two disjoint arcs, and then moves one arc over the other. Reidemeister move 2B does move 2A in reverse, by moving the arc which passes over another arc twice to be disjoint from the second arc. Both moves 2A and 2B are depicted in Figure 4.

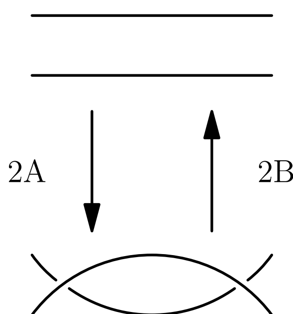

Figure 4: The second type of Reidemeister move

Type 3. Reidemeister move 3A takes a strand which lies above the overpass of a nearby crossing, and moves the strand over the crossing. Reidemeister move 3B moves the strand in the opposite direction with respect to the crossing. The two type 3 moves are shown in Figure 5.

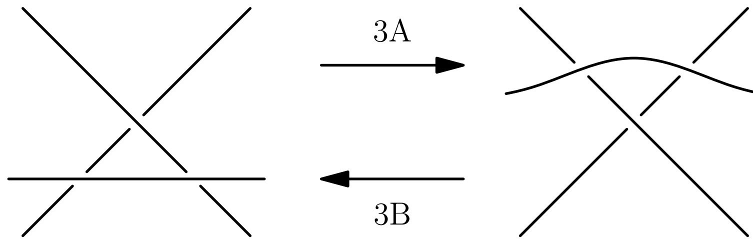

Figure 5: The third type of Reidemeister move

Definition 2.4. The *Kauffman bracket polynomial*, is a polynomial assigned to every link diagram D_L, denoted by $\langle D_L \rangle$. The Kauffman bracket polynomial is recursively defined by sending the unknot to 1, and using the following rules for link diagrams which concur outside of the dotted circle:

(1) $\langle \quad \rangle = 1$

(2) $\langle \quad \rangle = (-A^2 - A^{-2})\langle \quad \rangle$

(3) $\langle \quad \rangle = A\langle \quad \rangle + A^{-1}\langle \quad \rangle$

As defined, the Kauffman bracket polynomial is invariant under the second and third types of Reidemeister moves, but changes by $-A^{\pm 3}$ if a move of type 1 is applied. Thus, in order to define an invariant which is unchanged by all three types of Reidemeister moves, we must modify the Kauffman bracket polynomial.

Definition 2.5. The *writhe* of a link diagram D_L, denoted by $w(D_L)$, is the number of right-handed crossings minus the number of left-handed crossings. The *normalized bracket polynomial*, defined as $X(L) = (-A^{-3})^{w(D_L)}\langle D_L \rangle$, is then an invariant under all three Reidemeister moves. After substituting $A = t^{-\frac{1}{4}}$, we obtain the *Jones polynomial* of L, denoted by $V(L)$. As shown in [Jon87], the resulting polynomial is well-defined.

3. Tangles

3.1. **Tangle Definitions.** We will present a practical implementation of the algorithm discussed in [ENSS14] by Ellenberg et al. First, we must define tangles, which are the main structures used in the algorithm. Informally speaking, tangle diagrams may be thought of as placing a disk over a portion of a link diagram, and only considering the part of the link contained within or on the boundary of the disk. More formally, we have the following definitions.

Definition 3.1. Let D be the closed disk. A *tangle* is an embedding of a closed 1-manifold into $D \times (0, 1)$.

Analogous to how we represent knots with knot diagrams, we define tangle diagrams to be the two-dimensional representations of tangles.

Definition 3.2. [ENSS14] A *tangle diagram* is a finite multigraph embedded in a disk which satisfies the below properties:

(1) Every vertex has degree 1, 2, or 4.
(2) All univalent vertices lie on the boundary of the disk, with their only edge coming out of the boundary. The graph does not intersect the boundary in any other places.
(3) The two edges attached to every bivalent vertex form a self-loop.
(4) Every quadravalent vertex, which represents a crossing, has two opposite edges as the "over" crossing, and two opposite edges as the "under" crossing. In a diagram, the two edges forming the "over" crossing are connected, whereas the two edges forming the "under" crossing are disjoint.

Tangle diagrams are only to be considered up to planar isotopy.

Example 3.3. Figure 6 shows a tangle with a trefoil knot embedded in a disk as well as 4 strands forming an additional crossing.

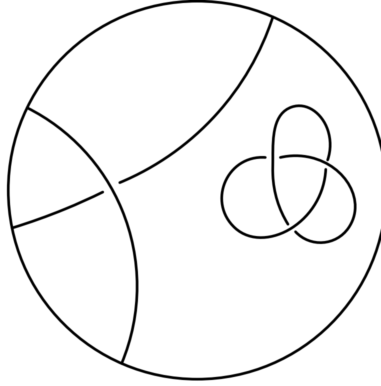

Figure 6: A tangle with 4 vertices on the boundary of the disk

Definition 3.4. [ENSS14] For a tangle T, the *type* of T is defined as the boundary circle with the univalent vertices on the circle. The univalent vertices are called the *boundary points* of T. A tangle is a *base tangle* if all of its vertices are univalent.

Example 3.5. Figure 7 is an example of a base tangle with 6 boundary points.

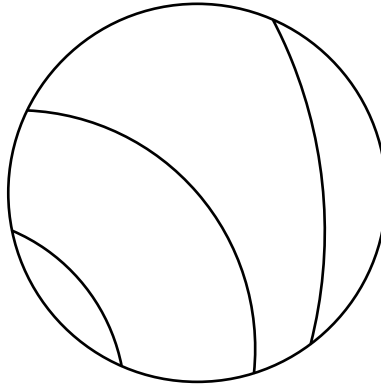

Figure 7: A base tangle with 6 boundary points

We note that naturally, we may extend the notion of the Kauffman bracket polynomial to tangles. Using equations 2 and 3 provided by Theorem 2.4, the Kauffman bracket polynomial of a tangle T may be expressed as a linear combination of base tangles with the same type as T, with elements of $\mathbb{Z}[A, A^{-1}]$ as coefficients.

Definition 3.6. Let B be any type. The *Kauffman skein module* is the free module over $\mathbb{Z}[A, A^{-1}]$ whose basis is the set of all base tangles with type B, considered up to isotopy equivalence.

Example 3.7. We can find the Kauffman Bracket polynomial of a tangle with one crossing in terms of two base tangles: $\langle \quad \rangle = A\langle \quad \rangle + A^{-1}\langle \quad \rangle$.

In the algorithm to compute the Jones polynomial, we will take advantage of the Kauffman skein module's *locality.* More specifically, suppose that T is a tangle with type B, and we cut out a disc within T that forms a tangle T' with type B'. To compute $\langle T\rangle$, we may first compute $\langle T'\rangle$ as an element of $\mathcal{M}_{B'}$. Then, we may paste each of the base tangle diagrams with type B'

back into T, which gives a linear combination of tangles equal to $\langle T\rangle$. By smoothing out all of the crossings with the rules in Theorem 2.4, we obtain $\langle T\rangle$ expressed as an element of $\mathcal{M}_B$. For links, we can use the Kauffman skein module's locality by starting with a tangle which consists of only one crossing, and then add each of the other crossings one by one, while simplifying the Kauffman bracket polynomial after every crossing is added.

Example 3.8. In Figure 8, to compute the Kauffman bracket polynomial of the outer tangle, we can first compute the Kauffman bracket polynomial of the inner tangles in terms of its respective base tangles.

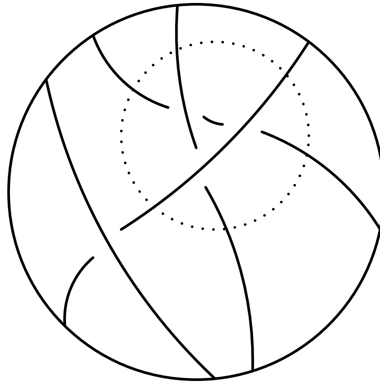

Figure 8: A smaller tangle with 3 crossings contained in a larger one

Definition 3.9. Let L be a link with n crossings. A *cutting* of L is a sequence of $n+1$ tangles denoted by T_0, T_1, $\ldots$, T_n such that T_0 is empty, T_n consists of L embedded in a disk, and for $0 \leq i < n$, the graph of $T_{i+1} - T_i$ consists solely of edges connecting the boundary vertices of T_{i+1} to T_i, in addition to one crossing. The *girth of a cutting* is the maximum number of boundary vertices of all tangles in the cutting. The *girth of a link* is the minimum girth across all cuttings of the link.

Example 3.10. In Figure 9, we have a cutting of the trefoil knot with girth 4 since the tangle intersects T_2 at 4 points.

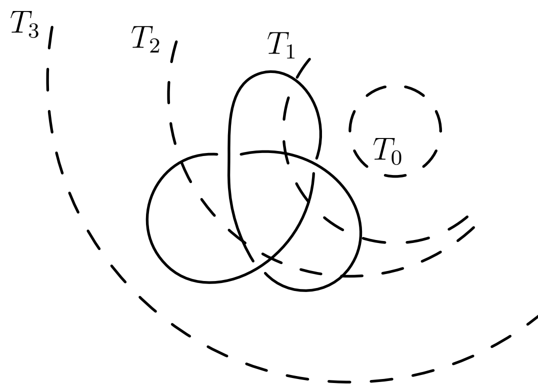

Figure 9: A cutting of the trefoil knot. The cutting's girth is 4 since the knot intersects T_2 at four points.

In the tangle algorithm, links will be given in *PD code* (planar diagram code). For a link with n crossings, the PD code will consist of n quadruples $[a, b, c, d]$, one for every crossing, where a is the label of the under crossing which is oriented towards the crossing, and the rest of the labels at the crossing are given in clockwise order.

After we go to every new tangle in the cutting and add a new crossing, we will smooth out the crossing which we add. Thus, all tangles in the algorithm will have at most one crossing, and at most one vertex with degree greater than one. We will use the following representation of tangles in the algorithm.

Definition 3.11. Suppose T is a tangle with n boundary points. The *algorithm representation* of T consists of two ordered lists of n integers, as well as an ordered quadruple. The first ordered list, or L_1, is used to represent the type of the tangle. First, arbitrarily select a boundary point to be labeled as point 1. Starting from point 1 and going in a counterclockwise fashion, label the points consecutively as $2, 3, \dots, n$. The list L_1 consists n positive integers $(b_1,\ b_2,\ \dots,\ b_n)$ such that for $1 \leq i \leq n$, it holds that b_i is the label of the edge of boundary point i. The second ordered list in the algorithm representation of T, or L_2, represents which boundary points have strands drawn between them. The list L_2 consists of n positive integers $(e_1, e_2, \dots, e_n)$ such that the boundary point which lies on edge e_i is on the same strand boundary point i. Finally, the ordered quadruple of the algorithm representation, or C, records the crossing in T, if one is present. If T has a crossing, we will set $C = (a, b, c, d)$, where a is one of the two edges which is an under crossing, and the other edges are labeled in clockwise order. If T does not have a crossing, we simply set $C = (0, 0, 0, 0)$.

Example 3.12. Let T be the tangle shown in Figure 10. Suppose we label points $1, 2, \dots, 6$ as shown. In the algorithm representation of T, the type is specified by $(1, 2, 3, 4, 5, 6)$. The second list in the algorithm representation of T, which represents the strands between the boundary points, is $(5, 3, 2, 6, 1, 4)$. The ordered quadruple, which represents the crossing in T, can be either $(6, 5, 4, 1)$ or $(4, 1, 6, 5)$.

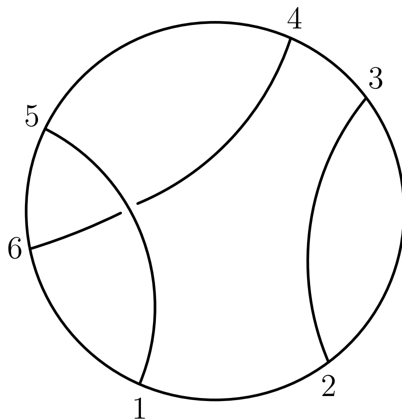

Figure 10: A tangle T

Fix a type B with n boundary points. Since every boundary point must be connected with exactly one other boundary point by a direct strand, n must be even. Up to planar isotopy, the number of base tangles which may be formed with type B is the same as the number of ways to draw $n/2$ non-intersecting line segments such that the endpoints of each line segment are boundary points on B, and every boundary point of B is the endpoint of a line segment. Thus, the number of such base tangles is the $(n/2)$th Catalan number, or $C_{n/2}$. In order to be able to differentiate between the base tangles quickly, we will need to index them, which we perform as follows.

Definition 3.13. Let T be a tangle with type B. Suppose that T has n boundary points, in the algorithm representation of T, its type is denoted by $(b_1, b_2, \dots, b_n)$. Let A be the list of $n/2$ distinct ordered pairs (b_i, b_j) with $b_i < b_j$ such that the edges b_i and b_j are part of the same strand on T. Sort the ordered pairs in A by only considering the first element of every ordered pair, and sort from least to greatest. Suppose that the ith ordered pair in A is now

(x_i, y_i). Then, the *Catalan list* of T is the list of $n/2$ nonnegative integers $(c_1, c_2, \ldots, c_{n/2})$, where $c_i = \frac{y_i - x_i - 1}{2}$. The *tangle index* of T is the number of possible Catalan lists for base tangles with type B and the same first boundary point which are lexicographically smaller than the Catalan list of T.

Example 3.14. For base tangle T_1, shown in Figure 11, the sorted list A is $\{(1,2),(3,4)\}$, and the Catalan list of T_1 is $(0,0)$. For base tangle T_2, shown in Figure 12, the sorted list A is $\{(1,4),(2,3)\}$ and the Catalan list is $(1,0)$. We thus find that T_1 has tangle index 0, and T_2 has tangle index 1.

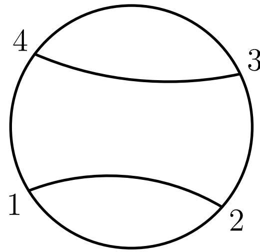

Figure 11: Base tangle T_1

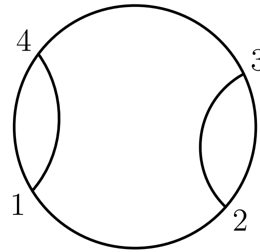

Figure 12: Base tangle T_2

Definition 3.15. A *Kauffman tangle summand* $(P(A), T)$ consists of a base tangle T along with a polynomial coefficient $P(A) \in \mathbb{Z}[A, A^{-1}]$.

3.2. **The Tangle Algorithm.** Before we present the tangle algorithm, we will also note that occasionally, adding a crossing to a tangle in the cutting will produce a crossing at which the same edge occurs multiple times. Since algorithm representations are not able to be used for such tangles, we will immediately smooth out the crossing according to the rules in Theorem 2.4. The calculations proceed as follows:

$$\begin{aligned} \langle \quad \rangle &= A\langle \quad \rangle + A^{-1}\langle \quad \rangle \\ &= (-A^3 - A^{-1})\langle \quad \rangle + A^{-1}\langle \quad \rangle \\ &= -A^3\langle \quad \rangle \end{aligned}$$

Likewise, we find that $\langle \quad \rangle = -A^{-3}\langle \quad \rangle$.

We now have all of the tools to show the entire tangle algorithm for computing the Jones polynomial. An implementation of the algorithm may be found at [MABS25].

Algorithm 1 Tangle Algorithm [ENSS14]

Require: A knot K in PD Code with n crossings and a cutting $T_0, T_1, \ldots, T_n$ for K

1: Set **current_kauffman_summands** to be an array of Kauffman tangle summands of length 1, with the only element being T_0 with a coefficient of 1
2: ▷ ***current_kauffman_summands****[i] is the base tangle with tangle index i* ◁
3: Set **previous_kauffman_summands** to a null array of Kauffman tangle summands
4: Set **crossings_visited** to 0
5: **while crossings_visited** $< n$ **do**
6: Increment **crossings_visited** by 1
7: **current_width** := the number of boundary points of $T_{\textbf{crossings_visited}}$
8: Set **previous_kauffman_summands** to **current_kauffman_summands**
9: Set **current_kauffman_summands** to an empty array of Kauffman tangle summands with length $C_{\textbf{current_width}/2}$
10: **for all** Kauffman tangle summands $(P(A), T)$ in **previous_kauffman_summands do**
11: Introduce the crossing in $T_{\textbf{crossings_visited}} - T_{\textbf{crossings_visited}-1}$ to T
12: **if** The crossing in T has an edge which appears more than once **then**
13: Remove the crossing in T
14: Add the new Kauffman summand to **current_kauffman_summands** at the corresponding tangle index
15: **else**
16: Smooth the crossing in T as done in Theorem 2.4
17: Add the two new Kauffman summands to **current_kauffman_summands** at the corresponding tangle indices
18: ▷ *At this point,* ***current_kauffman_summands*** *will only contain one Kauffman tangle summand* ◁
19: $P(A)$:= the polynomial coefficient of **current_kauffman_summands**[0]
20: w := the writhe of K
21: $X(A) := (-A^{-3})^w P(A)$
22: $t := A^{-4}$
23: **return** X as a polynomial in t

3.3. **Algorithm Performance.** To determine the time and memory complexity of the tangle algorithm, as cited by Ellenberg et al. [ENSS14], an upper bound for the girth of a knot with n crossings is $c\sqrt{n}$, where $c = 6\sqrt{2} + 5\sqrt{3}$ [DV03]. Additionally, from [ENSS14], we find that for any Kauffman tangle summand, the polynomial coefficient $P(A)$ will have a span of at most $4n$, with all powers of A having the same exponent modulo 4.

Theorem 3.16. *The time complexity for the tangle algorithm for a knot with n crossings is $O(n^{5/4}2^{c\sqrt{n}})$. The amount of memory required for the algorithm is $O(n^{1/4}2^{c\sqrt{n}})$.*

Proof. Every time we introduce a new crossing, we must take all of the Kauffman tangle summands from **previous_kauffman_summands**, apply the rules in Theorem 2.4, and then add the new Kauffman tangle summands to **current_kauffman_summands**. For each Kauffman tangle summand, we must keep track of the boundary points, and also the polynomial coefficients. Thus, each Kauffman tangle summand takes up $O(n)$ memory.

At every tangle in the cutting, we only need to keep track of the Kauffman tangle summands in **previous_kauffman_summands** and **current_kauffman_summands**, and we can discard the summands in **previous_kauffman_summands** after they are used. For any tangle in the cutting there can be no more than $C_{\textbf{current_width}/2}$ Kauffman tangle summands in total. Because the mth Catalan number C_m is $O(m^{-3/2}4^m)$ and current width $\leq c\sqrt{n}$, there must be no more than $O(n^{-3/4}2^{c\sqrt{n}})$ Kauffman tangle summands for each cutting. Thus, since there are $n + 1$ tangles in the cutting, we require $O(n^{1/4}2^{c\sqrt{n}})$ memory for the algorithm.

The most time-costly step in the algorithm is when we must add a new Kauffman tangle summand to **current_kauffman_summands**, since it requires determining the tangle index

of a base tangle, and then possibly adding the polynomial coefficients of two tangles. It takes $O(\sqrt{n})$ time to determine the tangle index of a base tangle, and up to $O(n)$ steps to add the polynomial coefficients of two tangles, so the more costly step is the addition of polynomial coefficients. Over all n crossings, we must use at most $O(n^{1/4}2^{c\sqrt{n}})$ Kauffman tangle summands, meaning the time necessary for the algorithm is at most $O(n^{5/4}2^{c\sqrt{n}})$. □

Despite the appearance of $c\sqrt{n}$ in the exponent for the time complexity of the algorithm, the tangle algorithm runs considerably faster in the majority of knots, as most knots have a much smaller girth. In fact, as we find below, for certain classes of knots, the tangle algorithm is able to run in quadratic time.

Proposition 3.17. *For all twist knots, as well as torus knots of the form $(p, 2)$ for all odd integers $p > 2$, the tangle algorithm is able to compute the Jones polynomial in quadratic time.*

Proof. Let K be a twist knot or a $(p, 2)$ torus knot for an odd integer $p > 3$, with n crossings. Then, the girth of K is 4, so in the tangle algorithm, the array **current_kauffman_summands** can contain up to $C_2 = 2$ Kauffman tangle summands. The addition of the polynomial coefficients of two Kauffman tangle summands takes up to $O(n)$ time, so across all n crossings, the tangle algorithm takes at most $O(n^2)$ time. □

Using the C implementation of the algorithm in [MABS25], the Jones polynomial of the $(25, 2)$ torus knot can be computed in 0.001 seconds, which is orders of magnitude faster than the computation from several other exponential algorithms. However, we do also note that for torus and twist knots, the Jones polynomial has a closed form solution. The (p, q) torus knot has the Jones polynomial

$$t^{(p-1)(q-1)/2}\frac{1-t^{p+1}-t^{q+1}+t^{p+q}}{1-t^2}.$$

On the other hand, a twist knot with n half-twists has the Jones polynomial

$$\frac{1+t^{-2}+t^{-n}-t^{-n-3}}{t+1}$$

if n is odd and

$$\frac{t^3+t-t^{3-n}+t^{-n}}{t+1}$$

if n is even. Thus, for torus and twist knots, we find that the speed of the tangle algorithm is unnecessary.

While the tangle algorithm runs in sub-exponential time, there are many limitations. The input must be a classical knot in order for the algorithm to work. The algorithm fails for virtual knots because virtual crossings are not handled by the planar tangle skein module used in this algorithm. Moreover, for the C implementation, we must define an entirely new data structure to represent tangles, resulting in part of the runtime resulting from conversion between PD code and tangles. For families of knots which take polynomial time for the tangle algorithm, the conversion makes up a larger portion of the total runtime, and hence provides an area of the algorithm which could be improved in the future.

4. Skein-Template Algorithm

We will present an implementation of the algorithm outlined in [GMGCL99].

Program 4.1. `skein_template_algorithm` *We start by labeling the edges of the link. Then, we begin at edge 1. We move until we encounter a crossing. If it is an overcrossing, we decorate it with ◯ and proceed, but if it is an undercrossing, we generate two new links. The first results from splicing the crossing so that ⊗ becomes ◯ and crossing orientations are preserved. The second results from reversing the crossing (bringing the edge going under to the top) and once again decorating it with ◯ . We then proceed to the next edge and repeat this process for each generated link. Once all crossings have been decorated on all links, we sum $(-1)^{t_-(S_i)}(\sqrt{t}-\frac{1}{\sqrt{t}})^{t(S_i)}(t)^{-w(S_i)}(-\sqrt{t}-\frac{1}{\sqrt{t}})^{||S_i||}$ over all links generated by the recursion, where $t_-(S_i)$ denotes*

the number of splices of negative crossings, $t(S_i)$ denotes the number of spliced crossings, and $||S_i||$ is one less than the number of link components of S_i.

Definition 4.2 (Sign of Crossing)**.** We define a crossing to be positive and negative as shown in the figures below

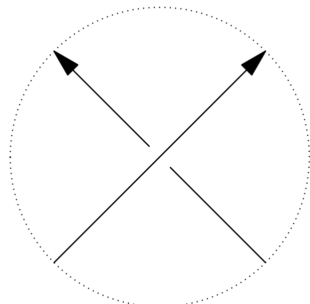

Figure 13: A Positive Crossing

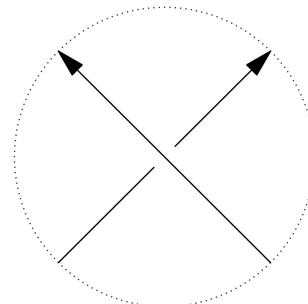

Figure 14: A Negative Crossing

Definition 4.3 (Gauss Code)**.** Gauss code uniquely encodes a knot or link by indicating the order of traversal of the crossings, the sign of each crossing, and whether they are encountered as undercrossings or overcrossings.

Example 4.4 (Gauss Code)**.**

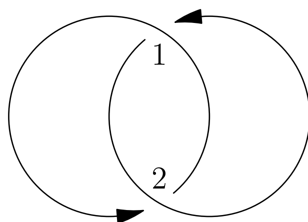

Figure 15: Hopf Link

The Gauss code for the Hopf link, pictured above, reads $\{1\text{O}+2\text{U}+, 1\text{U}+2\text{O}+\}$ since both crossings are positive, in the first component, 1 is encountered as an overcrossing and 2 is encountered as an undercrossing, and vice versa for the second component.

In pseudocode, the Skein-Template Algorithm reads

Algorithm 2 Skein_Template_ Algorithm

Input: Gauss Code

Output: Laurent Polynomial

Move along the edge until a crossing is encountered.
while crossings undecorated **do**
  for all links in set **do**
    if we encounter an undecorated overcrossing **then** we decorate it with ◯ and proceed. Any crossing decorated with ◯ should not be spliced or reversed.
    if we encounter an undecorated undercrossing **then** we branch into two cases. Note that this will generate two distinct links.
      Splice: We `splice` the crossing so that ⊗ becomes ◯ and crossing orientations are preserved.
      Reverse and Decorate: We change the sign of the crossing by bringing the edge passing over the crossing behind the one passing under and decorate it with ◯ .
      set = splice(set) $\cup$ reverse(set)
    Choose next crossing for the `splice` link and reverse link; if all crossing in link component have been taken, proceed to the next lowest crossing.
Sum $(-1)^{t_-(S_i)}(\sqrt{t}-\frac{1}{\sqrt{t}})^{t(S_i)}(t)^{-w(S_i)}(-\sqrt{t}-\frac{1}{\sqrt{t}})^{||S_i||}$ over all links generated in the set generated by the above loop. $t_-(S_i)$ denotes the number of splices of negative crossings, $t(S_i)$ denotes the number of spliced crossings, and $||S_i||$ is one less than the number of link components of S_i.

Example 4.5. The Twisted Unknot

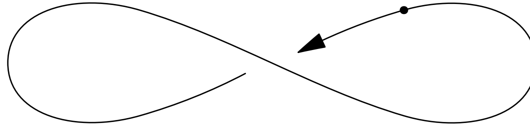

Figure 16: Twisted Unknot

Our first crossing is an undercrossing, so we split into two cases. Splicing produces the unlink.

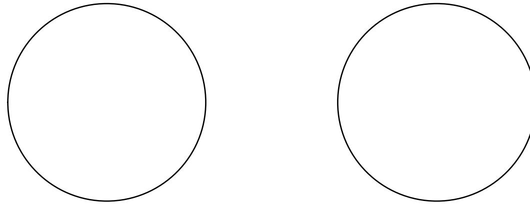

Figure 17: Unlink

Reversing produces the unknot.

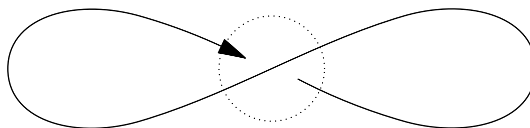

Figure 18: Decorated Twisted Unknot

Summing the terms in the recurrence relation corresponding to the links depicted in figures 17 and 18, we find that the Jones polynomial of the twisted unknot is 1. This example also demonstrates that depending on where the base point is, the algorithm can make different numbers of links and thus take different amounts of time.

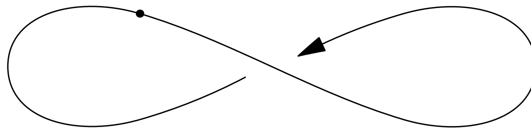

Figure 19: Twisted Unknot

If the base point were as is shown above, the crossing would be decorated and there would only be one resulting link instead of two. Thus, it may be possible to reduce the runtime of the algorithm by choosing a better starting point.

Theorem 4.6. *The Skein-Template Algorithm is correct and terminates. (Gouesbet et al., 1999).*

Remark 4.7. This is not explicitly proven in the referenced paper, although it follows quickly from some well-known results about ascending knots.

Proof. Sketch: First, note that with each iteration, the number of undecorated crossings decreases by 1, so the process terminates by descent. I also claim that all resulting links are unlinks. First, note that unlinks without crossings in their diagrams cannot be spliced or reversed, so if a 0-crossing link is inputted, an unlink will result. Note that splicing results in an $(n-1)$-crossing link and that reversing and decorating results L_1, the first link component, being on top of all others. Thus, this reduces to the case of $m-1$ link components. Hence, using induction, this reduces the case to that of disconnected knots of smaller size. Finally, we must show that if a knot results from this algorithm, it must be an unknot. Since `splice` reduces the number of crossings, we can use induction to show that this is equivalent to the statement "reversing every undecorated undercrossing and decorating it results in an unknot." We start at our first crossing, which must be an overcrossing. Note that the first undercrossing must result from passing under an already-decorated crossing.

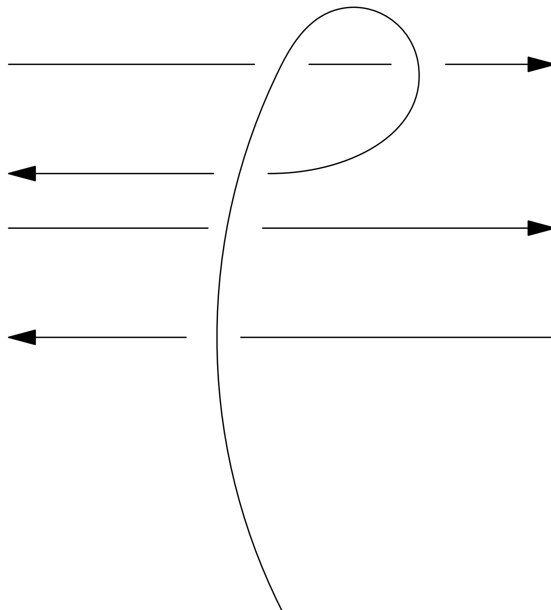

Figure 20: Part of an Unknot

Note that the twist shown in 20 may be undone. This results in a knot still satisfying the property that the first undercrossing must result from passing under an already-decorated crossing and with one fewer crossing. By induction, any such knot may be reduced to a knot with 0 crossings and must be an unknot. Thus, at the end of the algorithm, we are left with an unlink with some number of components, which has Jones polynomial $(-\sqrt{t} - \frac{1}{\sqrt{t}})^{||\text{unlink}||}$, which is consistent with the algorithm. Note that decorating a crossing does not change the link and thus preserves its Jones Polynomial; note also that no terms in the sum change so that the decorating has no impact on the Jones polynomial of the link and the result of the algorithm. Finally, noting that the recursion for undercrossings is equivalent to the skein relation $(\sqrt{t} - \frac{1}{\sqrt{t}})V(L_0) = \frac{1}{t}V(L_+) - tV(L_-)$, 4.6 follows. □

Remark 4.8. Knot/link diagrams satisfying the above property are called "descending," and it is well-known that they are unknots/unlinks.

Program 4.9. `is_it_smaller`. *This function takes in as input a Gauss code struct, which has a current crossing and subsequent crossing specified and outputs* 0 *if splicing the crossing would create a new link component and* 1 *if it would eliminate a link component. This code simply checks whether both instances of the crossing in the Gauss code are from the same link component.*

Program 4.10. `splice`. *We present pseudocode for the* `splice` *function working on Gauss code. On a crossing with edges $A \to B$ and $C \to D$, the* `splice` *function sends $A \to D$ and $C \to B$. In particular, if both edges are part of the same link component, then $xAByCDz \to (DzxA, ByC)$ and the link component splits into two different components. If they are part of different link components, then $(wABx, yCDz) \to xwADzyCB$, and the two merge. Thus, the* `splice` *function can be thought of as moving sections of the Gauss code and deleting the original crossing. The* `splice` *function handles each of these two cases separately, acts according to the piecewise definition above, and returns the spliced link. Written in pseudocode,*

Algorithm 3 Splice.

```
Input: struct Gauss g1
Output: struct Gauss output with current crossing spliced updated
if g1 splits into 2 components after splice then
    output.number_of_components = input.components + 1
    output.components[split_component] = take between 1st and 2nd instances of crossing
    output.components[final_component] = take everything else
else → g1 grafts into 1 component after splice
    output.number_of_components = input.components - 1
    output.components[grafted_component] = 1st before, 2nd after, 2nd before, 1st after
output.start = next crossing not labeled -1.
return output;
```

Program 4.11. `virtue`. *This function receives a link's Gauss code and outputs the Gauss code for a modified link. It finds all instances of the current crossing and "decorates" them by setting their crossing numbers to -1. Then, it sets the new current crossing to the next crossing not labeled -1. One can then quickly check whether a crossing has been decorated.*

Program 4.12. `jones_partial`. *Using polynomial structures, the two functions above, and the recursion outlined in the algorithm, this function computes the Jones polynomial of a knot or link up to some factor.*

Algorithm 4 *Jones_Partial.*

Input: *Link in Gauss code*

Output: *Laurent Polynomial in t*

if *number of undecorated crossings = 0* ***then***
  return $(-\sqrt{t} - \frac{1}{\sqrt{t}})^{\text{number of components}-1}$;
if *current crossing = overcrossing* ***then***
  return `jones_partial`*(virtue(link))*$\cdot t^{-\text{sign_of_crossing}}$
if *current crossing = undercrossing* ***then***
  return `jones_partial`*(virtue(link))*$\cdot t^{\text{sign_of_crossing}} + \text{sign_of_crossing} \cdot$ $\text{jones_partial}(\text{splice}(\text{link})) \cdot (\sqrt{t} - \frac{1}{\sqrt{t}})$;

Program 4.13. `writhe`. *This function computes the writhe by summing the signs in the Gauss code of the link and dividing by two (noting that each crossing is counted exactly twice).*

Program 4.14. `jones`. *This function scales the result from Jones_Partial to produce the Jones polynomial of the knot. The factor we scale it by is* $t^{-\text{writhe(link)}}$. *One can check that this definition is consistent with 2.*

Remark 4.15. The algorithm takes $2^{n+O(\log(n))}$ time in the worst case, where n is the number of crossings; in this case, the algorithm encounters no undecorated overcrossings. Thus, this algorithm is much slower than the one presented in 3, which runs in $2^{O(\sqrt{n})}$. The space complexity of the Skein-Template Algorithm is also $2^{n+O(\log(n))}$ in the worst case, since 2^n links are stored.

On the other hand, other knots, in which the Skein-Template Algorithm encounters few undercrossings, can have linear expansions. For example, all diagrams of the unknot with the property that there exists some base point such that as one travels along the knot diagram, one encounters all crossings first as overcrossings run in polynomial time. This is because there are no branches, so `virtue` is called $O(n)$ times, resulting in polynomial complexity. For other knot diagrams for which we are guaranteed to have a constant number of undecorated undercrossings encountered, we also get polynomial runtime.

Unlike the algorithms in 5 and 3, the Skein-Template Algorithm does not run faster for twist knots and torus knots because it still encounters many undecorated undercrossings, resulting in branching. Note that the Jones Polynomial has a closed form for twist knots and that it is much faster to use this than an algorithm taking exponential time. Implementations of all functions can be found in [MABS25].

5. Hecke Algebra Approach and Implementation Obstacles

We will finally present an implementation of the algorithm described by El-Misiery and El-Horbaty in [EMEH96]. We must first introduce the idea of a braid representation, which is used exclusively by this algorithm to work with knots.

Definition 5.1. A *closed braid representation* of a knot is a projection of that knot that takes the form of a braid diagram with the ends of corresponding strands connected to each other. By Alexander's Theorem, every classical knot or link has a closed braid representation [Ale23]. An example of a closed braid representation of a trefoil is given below:

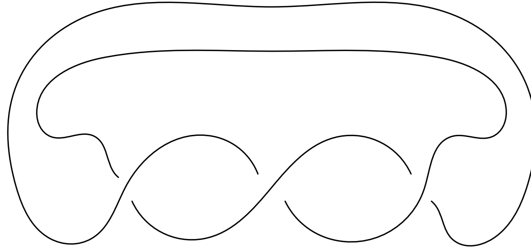

Figure 21: A closed braid representation of a trefoil

We can represent closed braid representations as elements in a braid group where the generators are single twists between adjacent strands in the braid.

Definition 5.2. [EMEH96] The *braid group on n strands* is a group generated by the $n-1$ generators $\sigma_1, \sigma_2, \ldots, \sigma_{n-1}$ under the following relations:

$$\sigma_i\sigma_j = \sigma_j\sigma_i \qquad |i-j| \neq 2$$
$$\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1} \qquad 1 \leq i \leq n-2$$

At any point in a braid, if we number the n strands in a braid with the integers from 1 to n inclusive, these relations arise from considering σ_i to represent a right-handed twist between strands i and $i+1$.

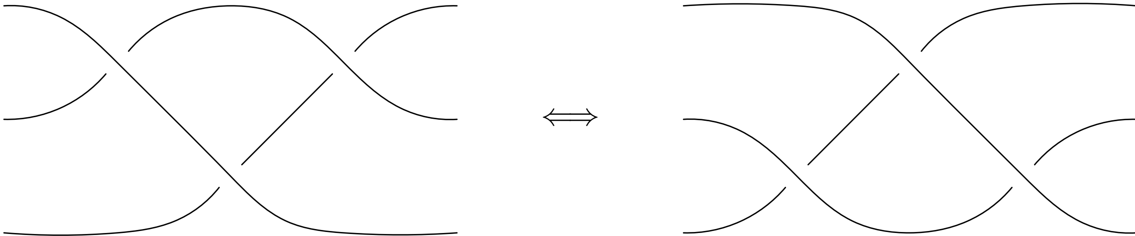

Figure 22: $\sigma_1\sigma_2\sigma_1 = \sigma_2\sigma_1\sigma_2$ in the braid group on 3 strands

To begin calculating the Jones polynomial of a knot, we consider the image of its closed braid group representation in a Hecke algebra, a finite-dimensional quotient in which the Jones polynomial can be computed.

Definition 5.3. [EMEH96] The Hecke algebra $H(t,n)$ of type A_{n-1} has the generators of $\delta_1, \delta_2, \ldots, \delta_{n-1}$ and a variable t satisfying the following relations:

$$\delta_i\delta_j = \delta_j\delta_i \qquad |i-j| \neq 2 \tag{1}$$
$$\delta_i\delta_{i+1}\delta_i = \delta_{i+1}\delta_i\delta_{i+1} \qquad 1 \leq i \leq n-2 \tag{2}$$
$$\delta_i^2 = (1-t)\delta_i + t \qquad 1 \leq i \leq n-1. \tag{3}$$

Remark 5.4. Notice that the first two relations are identical to those of braid groups, and the third relation corresponds to one of the Skein relations satisfied by the Jones polynomial. This is in fact the construction for the Jones Polynomial used in [Jon87].

Using (3), we can hope to write $\delta_i^k = c_1(t,k) \cdot \delta_i + c_2(t,k)$ for $k \in \mathbb{N}$ and some polynomials $c_1(t,k)$ and $c_2(t,k)$. Additionally, we would like to find two more polynomials $c_3(t,k)$ and $c_4(t,k)$ such that $\delta_i^{-k} = \Big(c_3(t,k) \cdot \delta_i + c_4(t,k)\Big)/(t^k)$ for $k \in \mathbb{N}$. [EMEH96] states recursive conditions that these polynomials must satisfy, but do not provide explicit definitions for these polynomials. We provide these definitions below.

Lemma 5.5. *For an integer $k \neq 0$, define Laurent polynomials $\Omega_1(t,k)$ and $\Omega_2(t,k)$ with*

$$\Omega_1(t,k) = \frac{1-(-t)^k}{1+t}, \qquad \text{and} \qquad \Omega_2(t,k) = \frac{t+(-t)^k}{1+t}.$$

Then, in the Hecke algebra $H(t,n)$ of type A_{n-1},

$$\delta_i^k = \Omega_1(t,k) \cdot \delta_i + \Omega_2(t,k). \tag{4}$$

Lemma 5.5 can be verified using a standard inductive argument.

Lemma 5.6. *Consider the braid group on n strands. For $s = \pm 1$, any integer k, and any integer i with $1 \leq i \leq n-2$, the following identities hold:*

$$\sigma_i^s \sigma_{i+1}^s \sigma_i^k = \sigma_{i+1}^k \sigma_i^s \sigma_{i+1}^s$$
$$\sigma_i^k \sigma_{i+1}^s \sigma_i^s = \sigma_{i+1}^s \sigma_i^s \sigma_{i+1}^k.$$

Proof. Since relation (2) is symmetric, as long as we only use it to prove the first identity, the second identity must hold as well. Additionally, notice that relation (2) must hold for inverses as well, because

$$\begin{aligned} &\sigma_i \sigma_{i+1} \sigma_i = \sigma_{i+1} \sigma_i \sigma_{i+1} \\ \implies &\big(\sigma_i \sigma_{i+1} \sigma_i\big)^{-1} = \big(\sigma_{i+1} \sigma_i \sigma_{i+1}\big)^{-1} \\ \implies &\sigma_i^{-1} \sigma_{i+1}^{-1} \sigma_i^{-1} = \sigma_{i+1}^{-1} \sigma_i^{-1} \sigma_{i+1}^{-1}. \end{aligned}$$

Thus, the first identity for $s = -1$ will hold as long as $s = 1$ holds. To tackle the first identity in the case where $s = 1$, we will do casework on k.

If $k = 0$, then the first identity becomes trivial.

If $k > 0$, then we must have

$$\sigma_i \sigma_{i+1} \sigma_i^k = \sigma_{i+1} \sigma_i \sigma_{i+1} \sigma_i^{k-1} = \sigma_{i+1}^2 \sigma_i \sigma_{i+1} \sigma_i^{k-2} = \cdots = \sigma_{i+1}^k \sigma_i \sigma_{i+1}.$$

If $k < 0$, then first notice that $\sigma_{i+1}\sigma_i\sigma_{i+1} = \sigma_i\sigma_{i+1}\sigma_i \implies \sigma_i\sigma_{i+1} = \sigma_{i+1}^{-1}\sigma_i\sigma_{i+1}\sigma_i$. Then,

$$\sigma_i \sigma_{i+1} \sigma_i^k = \sigma_{i+1}^{-1} \sigma_i \sigma_{i+1} \sigma_i^{k+1} = \sigma_{i+1}^{-2} \sigma_i \sigma_{i+1} \sigma_i^{k+2} = \cdots = \sigma_{i+1}^k \sigma_i \sigma_{i+1}. \qquad \square$$

Corollary 5.7. *Consider the Hecke algebra $H(t,n)$ of type A_{n-1}. For $s = \pm 1$, any integer k, and any integer i with $1 \leq i \leq n-2$, the following identities hold:*

$$\delta_i^s \delta_{i+1}^s \delta_i^k = \delta_{i+1}^k \delta_i^s \delta_{i+1}^s$$
$$\delta_i^k \delta_{i+1}^s \delta_i^s = \delta_{i+1}^s \delta_i^s \delta_{i+1}^k.$$

Let a knot K have the closed braid group representation $\sigma_{i_1}^{j_1}\sigma_{i_2}^{j_2}\ldots\sigma_{i_m}^{j_m}$, and let it be represented as the product of Hecke algebra generators $A = \delta_{i_1}^{j_1}\delta_{i_2}^{j_2}\cdots\delta_{i_m}^{j_m}$. Let I represent the array $\{i_1, i_2, \ldots, i_m\}$, and let J represent the array $\{j_1, j_2, \ldots, j_m\}$. Finally, let $N = \max(i_1, i_2, \ldots, i_m)$ be the largest subscript found in I before any computations are performed.

For the computation of the actual Jones polynomial, a formula is provided by [EMEH96] to calculate a two-variable Jones polynomial in w and t:

$$\left(\left(\frac{w(1-t)^2}{(1-wt)^2}\right)^{\left(-\frac{N}{2}+\frac{1}{2}\sum_{k=1}^{m} j_k\right)}\right)\cdot \operatorname{tr}(A). \tag{5}$$

Most of the algorithm will focus on calculating this trace function since the normalization factor (everything except $\operatorname{tr}(A)$, which is the Ocneanu trace [Jon87]) in the front is comparatively straightforward to handle.

Program 5.8. `collect`. *Read through I and J and merge adjacent terms $\delta_{i_k}^{j_k}\delta_{i_{k+1}}^{j_{k+1}} \to \delta_{i_k}^{j_k+j_{k+1}}$ whenever $i_k = i_{k+1}$. For example, $\delta_2\delta_1^2\delta_1\delta_3^3$ becomes $\delta_2\delta_1^3\delta_3^3$.*

If the last strand in the closed braid representation is only involved in one sequence of consecutive twists, then we can remove the last strand and adjust the trace by a multiplicative factor.

Program 5.9. `drop`. *Let $i_{\max} = \max(i_1, i_2, \ldots, i_m)$ be the current largest subscript in I. We require that $i_{\max}$ only appears in I once. Find the integer a such that $i_a = i_{\max}$. Then, there exists the term $\delta_{i_a}^{j_a}$ in A. Remove i_a from I (and remove the corresponding exponent j_a from J), and return the multiplicative factor*

$$\Omega_1(t, j_a)\cdot\frac{1-t}{1-wt} + \Omega_2(t, j_a).$$

We will multiply the trace by this.

Otherwise, if there are multiple occurrences of $i_{\max}$ in I, then we will seek to get rid of them one by one. This will be accomplished by the combined efforts of several functions.

Program 5.10. `flush`. *Locate the first and second occurrences of $i_{\max}$ in I, and let us denote them as i_a and i_b in order of appearance. Apply relation (1) to A in order to move as many of these terms out from between the two occurrences as possible, and run* `collect` *afterwards. For example,* `flush` *on $\delta_1\delta_3^2\delta_1^2\delta_2\delta_3$ becomes $\delta_1^3\delta_3^2\delta_2\delta_3$ because δ_1 and δ_3 commute.*

Program 5.11. `squeeze`. *After running* `flush`*, we are left with a subarray of terms that relation (1) cannot be used to flush any further. For $i_k = 4$, the product $\delta_4\delta_3\delta_2\delta_3\delta_4$ is an example. Since the only generators that do not commute are those with subscripts that differ by 1 (such as δ_4 and δ_3), in order for any such subarray to exist, it must begin with a sequence of terms whose subscripts decrease by 1. For some integer $k \geq 1$, we can express these first few terms as follows:*

$$\delta_{i_a}^{j_a}\cdot\delta_{(i_a)-1}^{j_{a+1}}\cdot\delta_{(i_a)-2}^{j_{a+2}}\cdots\delta_{(i_a)-k}^{j_{a+k}}.$$

If there existed a term whose subscript decreased by at least 2 with respect to the previous term, then we would have been able to flush it out by commuting it with all previous terms in the subarray. If we pick the largest possible value of k, this also means that the first subscript i_{a+k+1} in the subarray that fails to decrease by 1 must be larger than the previous subscript. An example of this where $i_a = 9$ and $k = 3$ is provided below:

$$\begin{array}{cccccccccc} \cdots & \delta_9 & \delta_8 & \delta_7 & \delta_6 & \delta_8 & \cdots & \delta_9 & \cdots \\ \cdots & \delta_{i_a} & \delta_{i_{a+1}} & \delta_{i_{a+2}} & \delta_{i_{a+3}} & \delta_{i_{a+k+1}} & \cdots & \delta_{i_b} & \cdots . \end{array}$$

We can then use relation (1) to squeeze (move) the term $\delta_{i_{a+k+1}}^{j_{a+k+1}}$ *as far left as possible. Let* $x := i_{a+k+1}$ *prior to the movement. We can do this squeezing until we get to a term whose subscript is* $x-1$*. Since the subarray must begin with a sequence of terms whose subscripts decrease by 1, where we are forced to stop squeezing, we must end up with the sequence of subscripts* $\{x, x-1, x\}$*. In the previous example, this would mean moving the second* δ_8 *leftwards until we see the subscript sequence* $\{8,7,8\}$*:*

$$\begin{array}{ccccccccc} \cdots & \delta_9 & \delta_8 & \delta_7 & \delta_8 & \delta_6 & \cdots & \delta_9 & \cdots \\ \cdots & \delta_{i_a} & \delta_{i_{a+1}} & \delta_{i_{a+2}} & \delta_{i_{a+3}} & \delta_{i_{a+4}} & \cdots & \delta_{i_b} & \cdots . \end{array}$$

Once the desired subscript sequence $\{x, x-1, x\}$ *is achieved somewhere in the subarray, this function is done.*

Program 5.12. `bgam`. *After* `squeeze` *has modified* A *and put it into a form where there exists 3 consecutive terms of the form* $\delta_x^p \delta_{x-1}^q \delta_x^r$*, we can apply Equation 4 to these terms with the goal of obtaining something of the form* $\delta_x^s \delta_{x-1}^s \delta_x^t$ *or* $\delta_x^t \delta_{x-1}^s \delta_x^s$ *for* $s = \pm 1$ *and some integer* t *in order for Corollary 5.7 to be applicable. For example, if we are given* $\delta_x^p \delta_{x-1}^q \delta_x^r = \delta_2^3 \delta_1^3 \delta_2^3$*, we would have:*

$$\begin{aligned} &\cdots \delta_2^3 \delta_1^3 \delta_2^3 \cdots \\ =&\cdots \Big(\Omega_1(t,3)\cdot\delta_2 + \Omega_2(t,3)\Big)\Big(\Omega_1(t,3)\cdot\delta_1 + \Omega_2(t,3)\Big)\delta_2^3 \cdots \\ =&\cdots \Big(\Omega_1(t,3)^2\cdot\delta_2\delta_1\delta_2^3 + \Omega_1(t,3)\Omega_2(t,3)\cdot\delta_1\delta_2^3 + \Big(\Omega_1(t,3)\cdot\delta_2 + \Omega_2(t,3)\Big)\cdot\Omega_2(t,3)\cdot\delta_2^3\Big) \cdots \\ =&\cdots \Big(\Omega_1(t,3)^2\cdot\delta_2\delta_1\delta_2^3 + \Omega_1(t,3)\Omega_2(t,3)\cdot\delta_1\delta_2^3 + \Omega_2(t,3)\cdot\delta_2^4\Big) \cdots . \end{aligned}$$

We have obtained the desired form with the $\delta_2\delta_1\delta_2^3$*. The other two items in the resulting sum result in two new Hecke algebra products* A_2 *and* A_3*, but they can still be plugged back into* `flush` *and have the entire process thus far repeated on them. Finally, using Corollary 5.7 on either one of our desired forms results in our subscript sequence* $\{x, x-1, x\}$ *being changed to* $\{x-1, x, x-1\}$*. This function should return a list of tuples of all resulting coefficient and exponent arrays as well as their Laurent polynomial multipliers. In our previous example, the tuples that are returned would thus be*

$$\Big(\{\cdots 2,1,2\cdots\}, \{\cdots 1,1,3\cdots\}, \Omega_1(t,3)^2\Big),$$

$$\Big(\{\cdots 1,2\cdots\}, \{\cdots 1,3\cdots\}, \Omega_1(t,3)\Omega_2(t,3)\Big),$$

$$\Big(\{\cdots 2\cdots\}, \{\cdots 4\cdots\}, \Omega_2(t,3)\Big).$$

Notice that all subscripts in the subarray before the subscript sequence must have been at least $x+1$*, and we have just been able to modify our subscript sequence to begin with* $x-1$*. This means that we must be able to flush out this first term with subscript* $x-1$ *and reduce the length of the subarray by 1 term.*

Algorithm 5 Hecke Algebra Algorithm [EMEH96]

Require: A knot K given in a closed braid representation

1: Set **queue_i** to be a queue of arrays that is initialized with a single element: the subscripts I of the closed braid representation of K
2: Set **queue_j** to be a queue of arrays that is initialized with a single element: the exponents J of the closed braid representation of K
3: Set **queue_p** to be a queue of Laurent polynomials initialized with the single element 1
4: Set **trace** to 0
5: Set **norm_factor** to be the normalization factor described in Expression 5
6: **while queue_i** is not empty **do**
7: Pop the first element from **queue_i** and set the popped data to **current_i**
8: Pop the first element from **queue_j** and set the popped data to **current_j**
9: Pop the first element from **queue_p** and set the popped data to **current_p**
10: **if current_i** is empty **then**
11: **trace** ← **trace** + **current_p**
12: Drop the first elements of **queue_i**, **queue_j**, and **queue_p**
13: **jump** to line 6
14: **Run** `collect` on **current_i** and **current_j**
15: Set **max_i** to the largest element in **current_i**
16: **if max_i** appears in **current_i** exactly once **then**
17: **Run** `drop` on **current_i** and **current_j**, and set **multi** to the return value
18: **current_p** ← **current_p** · **multi**
19: **else**
20: **Run** `flush` on **current_i** and **current_j**
21: **Run** `squeeze` on **current_i** and **current_j**
22: **Run** `bgam` on **current_i** and **current_j**
23: **for** each returned tuple (**bgam_i**, **bgam_j**, **bgam_p**) **do**
24: Push **bgam_i** onto **queue_i**
25: Push **bgam_j** onto **queue_j**
26: Push **current_p** · **bgam_p** onto **queue_p**
27: **return norm_factor** · **trace**

Remark 5.13. While the full logic behind this algorithm is presented above, we were unable to make this algorithm successfully calculate the Jones polynomial because we could not find substitutions for w and t that would turn the rational functions that result from this algorithm into polynomial expressions.

Theorem 5.14 ([EMEH96])**.** *The Hecke algebra algorithm terminates.*

Remark 5.15. The time complexity of this algorithm appears difficult to precisely analyze because it is very highly dependent on how easily the terms can be shuffled around in order to be merged with each other and subsequently simplified. In particular, it is very fast at handling closed braid representations where one twist is repeated many times in a row (since k such twists just gives us a single term of the form δ_i^k), but does take a lot of computation to deal with situations where `bgam` must be called many times.

In fact, the function `bgam` is the cause of most of the algorithm's runtime as it may return up to 3 Hecke algebra products that must be fed through the algorithm all over again. This leads to an approximately exponential runtime with respect to the number of crossings in the input closed braid representation. It is easy to see that all of the other functions defined in this section may be performed in linear time with respect to the number of terms in the Hecke algebra product (which is in turn bounded by the number of crossings in the closed braid representation).

6. Load-Balanced Algorithm

We will present an implementation of the algorithm outlined in [EM91]. Recall the definition of the Kauffman bracket polynomial:

$$\langle \quad \rangle = A\langle \quad \rangle + A^{-1}\langle \quad \rangle \tag{6}$$

$$\langle \quad \rangle = (-A^2 - A^{-2})\langle \quad \rangle \tag{7}$$

Note that the recursion usually has $2^{\text{crossing}\#}$ branches because each recursion smooths a single crossing. The Load-Balanced algorithm seeks to also reduce the number of crossings without employing the recursion by using Reidemeister moves of type I and II (R1 and R2). Further, it looks to smooth crossings that produce R1 and R2 moves in the resulting knots. Suppose that a knot has n crossings. The paper assumes that the time complexity of the algorithm is of the approximate form $\tilde{O}(\alpha^n)$ for some constant $\alpha < 2$, and then uses this as an estimate for the amount of computational work left to do after smoothing any given crossing. By comparing how much work is left to do, the algorithm can select which crossing to smooth at each step in the recursion.

Such crossings are found by searching for specific patterns in the knot diagram. Examples of patterns outlined in the paper are provided below. Note that, since we have already removed all R1 and R2 moves at a previous step, the signs of the crossings are not considered when searching for these patterns.

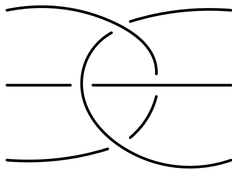

Figure 23: Triple

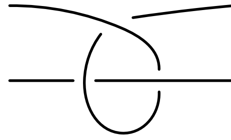

Figure 24: Gamma

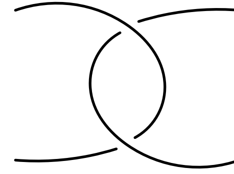

Figure 25: Bigon

Using our $\tilde{O}(\alpha^n)$ approximation for algorithm runtime, we can generate expressions to describe the estimated amount of time after smoothing a particular crossing in a pattern. For a bigon, smoothing either crossing results in one knot diagram where the crossing is removed and another diagram with a loop that can be immediately untwisted with a type I Reidemeister move. Thus, our estimate for the amount of time needed after such an operation (as a proportion of our estimate beforehand) is $\alpha^{-1} + \alpha^{-2}$, with the exponents corresponding to the change in the number of crossings in both of the knot diagrams that result from the crossing smoothing. Similarly, checking smoothings of all three crossings of the gamma pattern shows that smoothing the circled crossing is optimal,

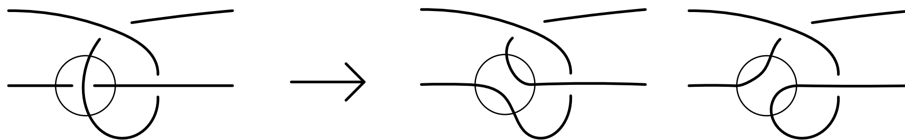

Figure 26: Smoothing a crossing in a gamma

and we once again end up with a new estimated runtime of $\alpha^{-1} + \alpha^{-2}$. Since this is identical to the time savings offered by smoothing a crossing in a bigon, and because a bigon can always be found inside of a gamma pattern, we can deduce that there is no need to specifically search for gamma patterns. It is possible to further refine our analysis of the gamma pattern by noting that the first resulting knot diagram has a bigon diagram inside of it, which (if one of its crossings

is subsequently smoothed) would result in an estimated runtime of $\alpha^{-1}(\alpha^{-1} + \alpha^{-2}) + \alpha^{-2} = 2\alpha^{-2} + \alpha^{-3}$. However, this is still worse than bigons for all $\alpha < (1 + \sqrt{5})/2$. Consequently, our algorithm does not search for gamma patterns at all.

By experimenting with different patterns, we have found an additional pattern that exhibits good performance: the canonical diagram for a type III Reidemeister move (R3).

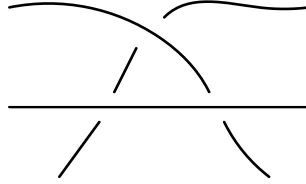

Figure 27: Reidemeister Type III Diagram

In particular, smoothing the circled crossing creates the opportunity to perform an R2,

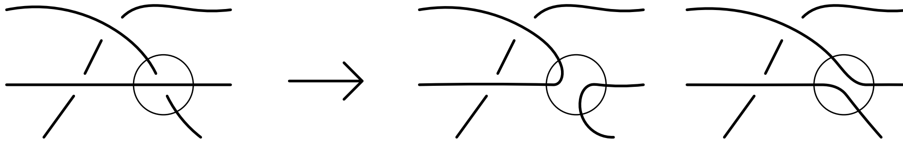

Figure 28: Smoothing a crossing in an R3 diagram

which results in an estimated runtime of $\alpha^{-3} + \alpha^{-1}$. This performance is strictly better than the bigon and gamma patterns for all α between 1 and 2. Furthermore, R3 diagrams can always be found within triple diagrams that admit smoothings with subsequent R1 and R2 simplifications, so we need not consider triples at all. The only triple that does not contain an R3 diagram, and also the only triple whose smoothings do not lead to immediate crossing removals via R1 and R2 moves, is provided below.

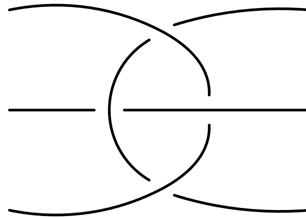

Figure 29: Triple of null type 4

Note that, since the number of crossings decreases by one with each recursion, the Load-Balanced Algorithm eventually terminates by descent.

Program 6.1. `struct crossing`. *The crossing structure we used is a variant of PD-code that is similar to a linked-list. Crossings contain*

(1) *Pointers to adjacent crossings as specified by PD code*
(2) *"Ports" arrays, which specify what part of the adjacent crossing they connect to (back of understrand, to the right of that, etc.)*

Ports are made necessary by edge cases, such as bigons, in which two crossings connect to one another in multiple ways. The motivation behind the structure was to be able to quickly traverse the knot and find crossings to the right/left in order to detect bigons, gammas, etc. Gauss code, for example, does not allow one to do this easily.

Program 6.2. `smooth_crossing`. *We reconnect crossings locally (switching ports arrays and pointers as needed), reorganize components, and ensure that understrand directions are consistent with one another. We branch into four cases depending on whether the understrand and overstrand belong to the same component and whether the smoothing is direction-preserving (K_0):*

1. *Case 1: K_0-smoothing with a single component → splits. Orientation consistency check unneeded.*
2. *Case 2: Non-K_0 smoothing with a single component → does not split. Orientation consistency check needed.*
3. *Case 3: K_0 with two components → merges. Orientation consistency check unneeded.*
4. *Case 4: Non-K_0 smoothing with two components → merges. Orientation consistency check needed.*

Algorithm 6 Load-Balanced Algorithm for the Kauffman Bracket Polynomial

Require: A link L, formatted using our `crossing` structures
1: **writhe_change** := 0
2: ▷ *Since R1 moves change writhe, we must maintain a variable **writhe_change** to be incremented or decremented as they occur* ◁
3: **while** there exists R1 or R2 simplifications in the knot **do**
4: Perform R1 and R2 moves in order to reduce the number of crossings in L, and update **writhe_change** accordingly
5: Set **number_of_unknots** to be the number of components of L that are canonical unknot diagrams
6: **if** the number of components of L is equal to **number_of_unknots then**
7: **return** $(-A^3)^{\textbf{writhe_change}} \cdot (-A^2 - A^{-2})^{\textbf{number_of_unknots}-1}$
8: **else**
9: Set **multiplier** to $(-A^3)^{\textbf{writhe_change}} \cdot (-A^2 - A^{-2})^{\textbf{number_of_unknots}}$
10: **if** an R3 diagram is found **then**
11: Set L_1 and L_2 to the resulting links after the correct crossing in the R3 diagram is smoothed
12: **else if** a bigon diagram is found **then**
13: Set L_1 and L_2 to the resulting links after a crossing in the bigon is smoothed
14: **else**
15: Set L_1 and L_2 to the resulting links after random crossing in L is smoothed
16: Set **polynomial_1** to the result of calling Algorithm 6 on L_1
17: Set **polynomial_2** to the result of calling Algorithm 6 on L_2
18: **return** $(AL_1 + A^{-1}L_2) \cdot$ **multiplier**

Algorithm 7 Load-Balanced Algorithm for the Jones Polynomial

Require: A knot K, formatted using our `crossing` structures
1: Set **kauffman_bracket** to the result of calling Algorithm 6 on K
2: $w :=$ the writhe of K
3: $X(A) := (-A^{-3})^w \cdot$ **kauffman_bracket**
4: $t := A^{-4}$
5: **return** X as a polynomial in t

7. Benchmarking the Jones Polynomial for Classical Knots

On a random sample of twelve 24-crossing knots the algorithms performed as follows:

Algorithm	Time (seconds)
Naive	33.819
Circle Counting	13.394
Skein Template Algorithm	1.611
Symbolic Calculus	0.121
Tangle Algorithm	0.052

Figure 30: Benchmarks on a random sample of 24-crossing knots

Hence, the Tangle Algorithm outperformed all other algorithms tested by a significant margin. We hope that this is helpful in extending Thistlethwaite's work to 21 and higher-crossing knot tabulation.

8. Acknowledgments

We thank Ryan Maguire for his help as we worked through the various algorithms and his guidance in writing this paper. We also thank PRIMES for affording us this opportunity.

References

[AJL06] D. Aharonov, V. Jones, and Z. Landau, *A Polynomial Quantum Algorithm for Approximating the Jones Polynomial*, 2006. ↑1

[Ale23] J W Alexander, *A lemma on systems of knotted curves*, Proceedings of the National Academy of Sciences of the United States of America **9** (1923), no. 3, 93–95. ↑15

[Ber23] C. Berkich, *An Introduction to Knot Polynomials* (2023). ↑1

[DV03] H. Djidjev and I. Vrťo, *Crossing Numbers and Cutwidths*, Journal of Graph Algorithms and Applications **7** (2003), no. 3, 245–251. ↑9

[EM91] B. Ewing and K. C. Millett, *A Load Balanced Algorithm for the Calculation of the Polynomial Knot and Link Invariants*, World Scientific Publ. Co. Singapore, 1991. ↑21

[EMEH96] A. E. M. El-Misiery and E.-S. M. El-Horbaty, *An Algorithm for Calculating Jones Polynomials*, Applied Mathematics and Computation **74** (1996), no. 2-3, 249–259. ↑2, 15, 16, 17, 18, 20

[ENSS14] L. Ellenberg, G. Newman, S. Sawin, and J. Shi, *Efficient Computation of the Kauffman Bracket*, Journal of Knot Theory and Its Ramifications **23** (2014), no. 05, 1450026. ↑2, 4, 5, 9

[GMGCL99] G. Gouesbet, S. Meunier-Guttin-Cluzel, and C. Letellier, *Computer Evaluation of Homfly Polynomials by Using Gauss Codes, With a Skein-template Algorithm*, Applied mathematics and computation **105** (1999), no. 2-3, 271–289. ↑2, 10

[Jon05] V. F. R. Jones, *The Jones Polynomial*, Discrete Math **294** (2005), 275–277. ↑1

[Jon87] V. F. R. Jones, *Hecke algebra representations of braid groups and link polynomials*, Annals of Mathematics **126** (1987), 335–388. ↑4, 16, 18

[Kup14] G. Kuperberg, *How hard is it to approximate the Jones polynomial?*, 2014. ↑1

[MABS25] R. Maguire, E. Ashoori, P. Bai, and H. Shieh, *Various algorithms for computing the jones polynomial*, 2025. ↑2, 8, 10, 15

[Mag24] R. Maguire, *Khovanov Homology and Legendrian Simple Knots* (2024). ↑2

[Mil22] J. Miller, *Quandle Invariants of Knots and Links* (2022). ↑1

[PP93] T. M. Przytycka and J. H. Przytycka, *Subexponentially Computable Truncations of Jones-type Polynomials*, Contemporary Mathematics **147** (1993), 63–108. ↑1

[Rei27] K Reidemeister, *Elementare begründung der knotentheorie*, Abh.Math.Semin.Univ.Hambg. (1927). ↑3

[Thi25] M. B Thistlethwaite, *The enumeration and classification of prime 20-crossing knots*, Algebraic and Geometric Topology **25** (2025), no. 1, 329–344. ↑1

E. Ashoori, Winchester High School, Winchester, MA, 01890
Email address: evanashoori@gmail.com

P. Bai, Lexington High School, Lexington, MA, 02421
Email address: aoepotato@gmail.com

H. Shieh, Westford Academy, Westford, MA, 01886
Email address: cyborgpenguinto@gmail.com