Introduction

Knots are a pervasive element of our world. They are simple because they can be created from bare-bones materials, but also complex because they can easily present a human with a good challenge to untie one. We use knots for many basic applications, and they turn up in all sorts of situations, so it is no surprise that we should want to understand knots on a formal basis. Knots can be studied mathematically after establishing a suitable definition. Within the framework created by a definition, it is possible to develop Knot Theory, which studies the properties of knots.

Knot Theory is an interesting branch of mathematics. The methods for proving theorems are relatively open-ended, giving it a certain freedom that other branches don't experience. Additionally, Knot Theory at first seems to be insubstantial, but in fact it is quite rich with deep results and connections to other branches.

Knot Theory is predominantly concerned with distinguishing different types of knots. There are several different reasons for distinguishing knots. One is to enumerate all distinct knots, which requires that you know which ones are actually distinct. Another reason is to determine which knots are possible to untie in the mathematical sense. This paper will serve as an introduction to elementary knot theory and will present a new idea for attacking the problem of distinguishing knots.

Definition of a Knot

The formal definition of a knot is constructed so that it resembles the notion of a contorted rope as closely as possible, while being careful to prevent any undesirable results down the road. The **conventional picture** of a knot is a finite length of rope in three-dimensional space with some form of interlacement that does not disappear when the rope is pulled taught. This is not yet a mathematically useful statement, so we will begin a translation. First, we need to know what a rope is. Roughly, it is a cylindrical curve, but in order to avoid problems with overlaps, we model it as a curved line. A second issue is that we would like to have any configuration be called a knot, so even if the interlacements disappear when the rope is pulled taught, it is still called a knot. Therefore, we must drop this requirement in the conventional picture. A third issue is that all knots in this scheme can be untied without cutting the rope, since it is of finite length and one end can be fed backwards through the knot until it is untied. A good mathematical definition should preserve the topological character of a knot, so we choose to connect the two ends of the rope. In this way, there will be no way to perform that untying procedure. A fourth issue is that the conventional picture tacitly assumes that the rope is solid, so it cannot intersect itself. It is necessary to explicitly state this in order to incorporate the assumption.

The **revised picture** of a knot is a finite curve in three-dimensional space that starts and ends at the same point and never intersects itself and may have some form of interlacement. Some terminology can be used to neaten up this statement. A curve is **closed** iff it starts and ends at the same point. A curve is **simple** iff it does not intersect itself. Now, the **revised picture** of a knot is a finite simple closed curve in three-dimensional space that may have some form of interlacement. The most difficult part of the translation remains to be dealt with. The phrase "some form of interlacement" needs to be clarified. The approach is to specify a starting point and a way of transforming from that starting point to an arbitrary knot.

We will briefly consider one natural possibility before presenting the final definition. The starting point could be the interval on the real line, [0,1] and the transformation could be any continuous function φ: [0,1]$\rightarrow \mathbb{R}^3$ such that $\varphi(0) = \varphi(1)$ so that the curve is closed and whenever $\varphi(x) = \varphi(y)$ we have x = y so that the curve is simple. This handles the conditions of the revised picture, and some sources use this as their definition of a knot, but this is not the conventional definition of a knot.

The previous idea, although straightforward, will be dismissed in favor of a definition that will facilitate progress in proving theorems. It is preferable to put a lot of work into the definition and having theorems flow readily than to have a quick definition that requires a lot of work to use. In the definition of a knot, the starting point is a triangle in R^3 and the transformation will be any sequence of insertions of bends into edges. This type of transformation need not be made precise because the definition of a knot just uses the end product—specifying a set of such transformations on a triangle is equivalent to specifying a polygon. A **polygonal curve**[1] is the union of a finite set of line segments $[p_1,p_2],[p_2,p_3],\ldots,[p_{n-1},p_n],[p_n,p_1]$ where $(p_1,p_2,\ldots,p_n)$ is an ordered set of distinct points in R^3. If the ordered set $(p_1,p_2,\ldots,p_n)$ defines a knot, and no proper subset defines the same knot, then the elements of the set $\{p_i\}$ are called the **vertices**[1] of the knot.

Now the translation can be concluded with a concise definition of a knot. A **knot**[1] is a simple closed polygonal curve in $\mathbb{R}^3$. This is the complete mathematical definition of a knot, and with it the study of Knot Theory may begin.

Equivalence

The primary objective in Knot Theory is to be able to distinguish between knots. Before this can be done, a definition of knot equivalence must be established. The approach will be to specify a subset of the knot creating transformations that are knot preserving. The subset of transformations that preserve a knot correspond to a certain class of transformations called ambient isotopies. An **ambient isotopy**[2] is a transformation that moves the knot through three-dimensional space without letting it pass through itself. This is the desired way of distinguishing knots because ambient isotopies correspond exactly to the manipulations that can be applied to a knotted rope without cutting it.

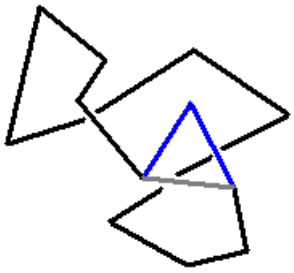


Figure 1 Gray segment is replaced by blue segment with an elementary deformation.

Ambient isotopies must now be formulated in terms of the definition of a knot. An ambient isotopy on a knot will be constructed from a sequence of elementary deformations. A knot J is an **elementary deformation**[1] of the knot K iff one of the two knots is determined by a sequence of points $(p_1,p_2,\ldots,p_n)$ and the other is determined by the sequence $(p_0,p_1,p_2,\ldots,p_n)$, where (1) p_0 is a point which is not collinear with p_1 and p_n, and (2) the triangle spanned by (p_0,p_1,p_n) intersects the knot determined by $(p_1,p_2,\ldots,p_n)$ only in the segment $[p_1,p_n]$. Criterion (1) of this definition ensures that the knots are actually geometrically different, and criterion (2) ensures that the insertion of a bend into an edge does not require the strand to pass through another strand. Such a passing is exactly what is restricted in an ambient isotopy, so elementary deformations correspond well to ambient isotopies.

Now it is possible to define the equivalence of knots. Knots K and J are **equivalent**[1] iff there is a sequence of knots $K=K_0,K_1,\ldots,K_n=J$, with each K_{i+1} an elementary deformation of K_i, for i greater than 0.
Only the notion of an elementary deformation is used in the definition, the notion of an ambient isotopy served only as a guideline, and has not been proved to be equivalent to an elementary deformation, so for the purposes of the mathematics, ambient isotopies are forgotten. We can prove that this definition is an equivalence class.

Theorem The definition of equivalent knots provides an equivalence class.
Proof: Reflexive: The sequence is trivial: $K=K_0=K$ and the definition holds, so K~K
Symmetric: If K~J, then there is a sequence of elementary deformations going from K to J. If this sequence is reversed, it starts on J and goes to K. Each pair is still linked by an elementary knot deformation because the definition of elementary deformation allows the two knots to be swapped. So J~K.
Transitive: If K~J and J~L, then there is a sequence of knot deformations from K to J and from J to L. By concatenating these sequences, and dropping the duplicate of J, a sequence is found that goes from K to L, so K~L. ⌘

So the definition of equivalence satisfies the usual notion of an equivalence class. The next step is to find ways of proving whether knots are equivalent according to this definition. The strategy is to find and utilize knot invariants.

A **knot invariant** is a function A from the set of knots such that if K~J, then A(K) =A(J). A **complete knot invariant** is a knot invariant such that if not(K~J) then A(K)≠A (J). Tools of this form are employed to prove various facts about where knots belong in the equivalence classes. One important example is the equivalence class of the trivial knot. The **trivial knot** (or unknot) is the knot represented by a triangle.
And so the equivalence class of the trivial knot is the set of all knots that can be untied into the trivial knot by a sequence of elementary deformations. Any knot in this equivalence class is referred to as trivial or unknotted. Oftentimes the trivial knot is pictured as a circle when not working with strict definitions.

It can be somewhat difficult to prove that a given function is a knot invariant, but there is a useful theorem from Kurt Reidemeister that acts as an intermediary.

Connected Sum

Two knots can be combined in a simple way to make a new knot. If a strand on each knot is cut and each of the freed ends glued to a freed end on the other knot, then the **connected sum** of the two knots is formed. This is a new knot, and it is called a **composite knot**, and it is denoted K#J if K and J were the two original knots, called **factors**. If a knot is not a connected sum of more than one nontrivial knot, then it is called a **prime knot**.

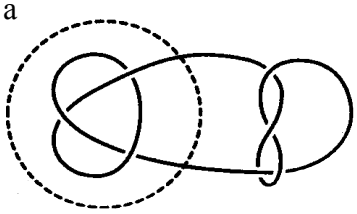


Figure 2 The connected sum of the trefoil and figure eight knots. Notice that a circle can be drawn around each factor.

Connected sums of knots are similar to products of integers, and in fact, an involved proof by Schubert showed that any knot can be uniquely decomposed into a connected sum of prime knots, analogous to the fundamental theorem of arithmetic. It follows that no knot has an inverse under connected

sum, which means that a connected sum of two nontrivial knots is never equivalent to the trivial knot. This fact is a very elementary concept, but it is far from intuitively obvious.

Another similarity between connected sum and integer multiplication is commutativity. The commutativity property of connected sums says that for any set of knots, $K_1,K_2,\ldots,K_n$, the connected sum $K_1\#K_2\#\ldots\#K_n$ is equivalent to the connected sum of any rearrangement of the same set of knots. This property can be illustrated with a technique called swallow-follow[2]. Suppose a thin torus is wrapped around the curve of one of the factor knots, but gets thicker around all the other factors and engulfs them whole. In this way, the torus swallows all but one factor, which it follows. The knot can then be slid through this shell until the follow-region is between any two other factors that were adjacent in the original connected sum. When the shell is removed, the one factor that was followed will be left in the new position in the connected sum. By repeating this procedure for each factor knot, it is possible to obtain any rearrangement of the original factors.

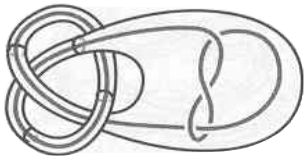


Figure 3 A swallow-follow torus that swallows figure eight knot and follows a trefoil knot.

Integer multiplication is not an exact parallel. One notable difference is that there can be two different connected sums of a pair of knots. This fact is a result of the orientation of knots. An **orientation**[2] is defined by choosing a direction to travel along the curve of a knot. Some knots can be deformed into the same knot with the orientation reversed. These knots are called **invertible**[2]. If either of the two original knots is invertible, then the connected sum will always yield the same knot. However, if both knots are non-invertible, then flipping one upside down with respect to the other will change the connected sum.

Projections

It is not always convenient to work with three-dimensional objects. Two-dimensional diagrams are often easier to think about and communicate with. According to their definition, knots are inherently three-dimensional, but there is a straightforward way to create flat diagrams. The diagrams are called projections, and they are made by drawing the shadow of a knot, but leaving gaps wherever there is an underpass. Usually, the polygon is converted into a smooth curve to improve the appearance as in Figure 2. With no extra requirements, there is opportunity for confusion. At certain angles, features of the knot may overlap. To eliminate this problem, projections are required to be regular projections. A **regular projection** is a projection of a knot where no three point on the knot project to the same point, and no vertex projects to the same point as any other point on the knot.

Any projections of any nontrivial knot will have points where two strands cross each other. These points are called crossing points. It turns out that there is an important knot invariant called crossing number. The **crossing number** of a knot is the least number of crossings that occur in any projection of the knot. This quantity is a very fundamental invariant, but it is difficult to compute for many knots. It is still not known whether the minimal crossing number is additive under connected sum.[2] This remains a big question in Knot Theory.

Reidemeister's Theorem

As mentioned earlier, it is not always easy to prove propositions about equivalence of knots using the bare definition. Working directly with the definition of equivalence requires an understanding of the behavior of three-dimensional polygons, which is not entirely straightforward. Reidemeister's Theorem serves as a layer of abstraction that converts between the definition of equivalence and some simple operations on knot projections. In 1926, Reidemeister introduced a set of three invertible moves, now called Reidemeister moves, which can be applied to any knot projection. The moves are the twist, the poke, and the slide, along with their inverse moves.

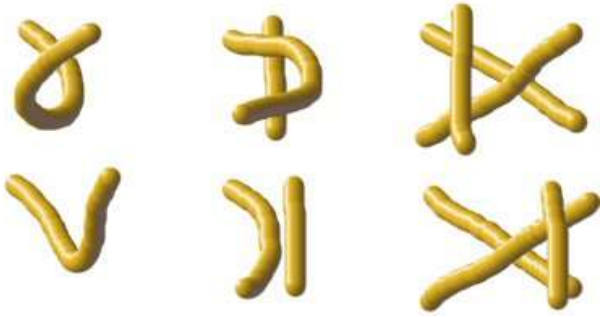


Figure 4 The Reidemeister Moves - Twist, Poke, and Slide

Theorem (Reidemeister's Theorem) If two knots are equivalent, then their projections can be obtained from one another by a sequence of Reidemeister moves.

Sketch of Proof: Since the two knots are equivalent, there is a sequence of elementary deformations that goes from one to the other. Along this sequence, each stage will have a projection, and if necessary, a small rotation will assure that the projection is regular. So, it suffices to show that the projection of a knot after an arbitrary elementary deformation is related to the original projection through a sequence of Reidemeister moves.

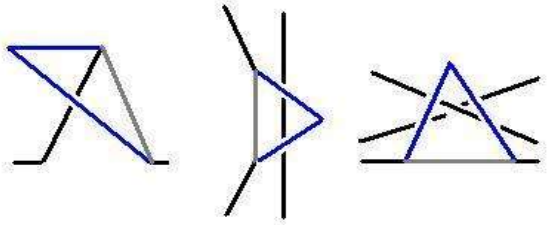


Figure 5 Subcases in the proof of Reidemeister's Theorem

Case: The elementary deformation adds a point to the polygonal curve.

Subcase: Projection of new point does not cross any lines.

The projection does not change, so no Reidemeister move is needed.

Subcase: Projection of new point crosses a single adjacent line. (Figure 5a)

A twist will mimic this elementary deformation.

Subcase: Projection of new point crosses a single nonadjacent line. (Figure 5b)

A poke will mimic this elementary deformation.

Subcase: Projection of new point crosses a single crossing point. (Figure 5c)

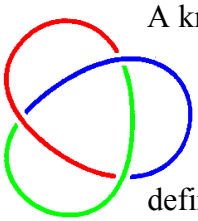
This elementary deformation can be duplicated in Reidemeister moves by two pokes followed by a slide.

All other subcases are combinations of the previous three subcases, so an appropriate sequence of Reidemeister moves can be found by concatenating the subsequences of moves. (A full proof would have to prove this claim.)

Case: The elementary deformation removes a point from the polygonal curve.

The Reidemeister moves are invertible, so similar subcases apply. ⌘

This theorem is extremely useful in Knot Theory. It could be called the Fundamental Theorem of Knot Theory because many important knot invariants are proved by showing that they are invariant under the Reidemeister moves. One simple example of an invariant that is proved in this way is tricolorability.



A knot projection is **tricolorable**[1] iff each continuous arc (from underpass to underpass) can be labeled with a color from a set of three colors, say red, green, and blue, such that at least two colors are used and at any crossing where two colors appear, all three colors appear. A knot is **tricolorable** iff its regular projections are tricolorable. This is well defined due to the following theorem.

Figure 6 The trefoil knot is tricolorable

Theorem Tricolorability is a knot invariant.

Sketch of Proof: We must show that if two knots K and J are equivalent, then either they are both tricolorable or they are both not tricolorable. By Reidemeister's Theorem, this is equivalent to showing that tricolorability is invariant under each Reidemeister move. So, it is sufficient to show that if a given knot is tricolorable, then it will also be tricolorable after any given Reidemeister move is applied. (Note that it follows that if a knot is not tricolorable, then after a Reidemeister move, it will still be not tricolorable, or else going in the reverse direction would violate this statement.) Consider only the isolated portion of the knot that is being modified by the move. The arcs that are connected to the rest of the knot will be given the same color before and after the move.

Case: Twist/Untwist

One arc becomes two, and each can be labeled with the same color as the original was colored. For the inverse move, each of the two must be labeled with the same color so as not to violate the second criterion for tricolorability, so the arc can be labeled with the same color as the original two.

Case: Poke/Unpoke

Two arcs become 4. If the original two were the same color, then all for can be labeled with this common color. If the original two were different colors, then the ends can be given their original colors while the middle arc gets labeled with the third color. For the inverse move, the two ends of the understrand must be the same color so as not to violate the second criterion for tricolorability, so it is possible to label the understrand with this color, and the overstrand with its original color.

Case: Slide

This case is tedious, so it will be omitted. ⌘

The trefoil is tricolorable as seen in Figure 6, but the trivial knot is not tricolorable. Since the previous theorem proved that tricolorability is a knot invariant, it follows that the trefoil is not equivalent to the trivial knot. This is significant because it proves that nontrivial knots exist under the current definition of equivalence.

The basic structure of this proof is used to prove many more advanced knot invariants.

The Trace Encoding

An **encoding** is a concise way of describing a knot that satisfies the requirement that no two inequivalent knots have the same encoding. An encoding is similar to a complete knot invariant except that two equivalent knots can have the different encodings. Also, an encoding tends to retain more information about a knot, so an encoding is similar to the inverse of an invariant. The trace encoding is an extension of the idea of projection. A projection reduces the knot from $\mathbb{R}^3$ to $\mathbb{R}^2 \times \mathbb{Z}_2$. This is because a projection is basically two-dimensional, but there is an extra piece of information coming from the crossing points that is stored in $\mathbb{Z}_2$. The heights jump from 0 to 1 somewhere between the crossing points so that at a crossing point, one is at 0 and the other is at 1. The trace encoding then acts on the projection to take the knot in $\mathbb{R}^2 \times \mathbb{Z}_2$ to an ordered set of elements in $\mathbb{Z}_n \times \mathbb{Z}_2$. The idea is that the actual locations of the crossing points in the plane are not important. The only information that is needed is the order in which crossings are made with respect to other crossings and which go over or under. A trace encoding can be created from a projection using the following algorithm.

Algorithm (Trace Encoding)

1) Label each crossing point in the projection with a unique positive integer.
2) Pick a starting point the knot in the projection that is not a crossing point and choose an orientation.
3) Trace along the knot in the given orientation until a crossing is reached. Say N is the label assigned to this crossing point. Add the pair (N,0) to the encoding in the event of an underpass or (N,1) in the event of an overpass.
4) Repeat step 3 until the starting point is reached. ⌘

After this procedure is finished, a sequence of pairs called **passings** will have been generated (see Appendix II). The integer that comes first in the passing is called the **point** and the Boolean value that comes second in the passing is called the **height**. Each crossing point in the projection will have two passings associated with it, and these two passings are called **mates** of each other. This sequence is easy to enter into a computer, and can also be easily manipulated by a computer.

The Reduction Algorithm

The trace encoding makes it possible for a computer to systematically apply Reidemeister moves to a knot. Given an arbitrary projection of a knot, this method could be used to find an encoding of a projection with minimal crossing number. If this were accomplished, it would be possible to use the program to determine whether the arbitrary knot is equivalent to the trivial knot. This procedure would mimic the process of a human untangling a loop of rope. Although it seems as if humans have the ability to do this with efficacy, in a complex case a human may be unable to finish the untangling at all. If the tangled rope is very convoluted, the human manipulator may spend a very long time and not succeed in untangling it. But the worst part is that since a human uses heuristic methods, it will never be known whether it is even possible to untangle the knot. If an algorithmic approach by a computer could find the minimal crossing number, this would prove whether a knot can be untangled, and even provide a partial classification of the knot in the event that it cannot be untangled. The classification is not complete because

there may be multiple encodings for the same knot, so this method is not necessarily useful for distinguishing two knots of the same crossing number. The goal of the reduction algorithm is to determine the crossing number of all prime factors of a knot from its trace encoding.

The approach using a systematic application of Reidemeister moves is guaranteed to work if you try all possible moves, but this does not produce a finite algorithm, as will soon be shown. A **finite algorithm** is an algorithm that can be assigned an upper bound for running time for each given input. If an algorithm is not finite, then it is possible for it to run forever without producing a result.

The first two Reidemeister moves can be implemented simply under the trace encoding. A routine for untwist just looks for two adjacent passings that have the same point, and deletes both passings. Applying untwist reduces the number of crossings by 1. The implementation for unpoke looks for two adjacent passings with the same height and checks that the mates of the two points are also adjacent, and then removes all four passings. Applying unpoke reduces the number of crossings by 2. The slide Reidemeister move produces some problems. Not only is it more difficult to implement, it is also does not reduce the number of crossings. Therefore, an algorithm would have to somehow prevent a slide and its inverse from being applied infinitely. There are ways of doing this, but they require memory. For a large knot, the memory required could be enormous. If this were the only problem, a finite algorithm would still be theoretically possible. However, there is an additional problem. In 1934,Goeritz proved by example that some projections of the trivial knot cannot be reduced by Reidemeister moves without going through a projection with a greater number of crossings.[3] Since there is no bound known on how high the number of crossings must go, the algorithm would have to be allowed to apply an arbitrarily long string of moves that increase crossing number. As a result, the algorithm does not have to terminate, and so is not finite.

The issues that arise from the straight Reidemeister move approach can be handled by requiring that the algorithm reduce the number of crossings strictly monotonically so that each move will definitely take the knot closer to its minimal crossing number projection. This means that the slide move needs to be removed, and also that more powerful moves have to be introduced. The slide move is undesirable in an algorithm because it is symmetrical. A move is **symmetric** iff it does not change the number of crossings. Slide is a very powerful move because it is symmetric, but the more powerful a move is, the harder it is to control in an algorithm. To fix the issues, we seek a set of asymmetric moves that is able to reduce any prime knot into its minimal crossing number projection with a strictly monotonic decrease in number of crossings.

Theorem A set of asymmetric moves exists that are able to reduce any prime knot into its minimal crossing number projection with a strictly monotonic decrease in number of crossings.

Proof: Given a prime knot, we know that a sequence of Reidemeister moves exists that reduces the knot to its minimal crossing number projection. Proceed through this sequence one move at a time until the number of crossings is decreased. Define a new move by the change from original knot to the current knot, and add this move to the set. Continue through the sequence of Reidemeister moves in this manner until all the moves have been used. Then repeat this process for every projection of every knot. In the end, the composite set will satisfy the stated requirements. ⌘

Unfortunately, this theorem does not state that the set of moves is finite. In the case that the set does happen to be finite, a finite algorithm can be programmed from the set. The success of a set of two powermoves is good evidence for the finiteness of the set of required moves. A powertwist is similar to the Reidemeister twist, except that there can be a knot inside of the loop. A powerpoke is similar to a Reidemeister poke, except that there can be crossings between the two crossings of the poke on either strand. These powermoves always decrease the number of crossings in a knot projection, and Goeritz's example can be reduced without increasing the number of crossings. There is no proof that these moves are sufficient to reduce an arbitrary knot to its minimal crossing number projection, but they do suffice for many notoriously difficult cases, such as the projection known as "The Monster" in Appendix II.

Figure 7 Powertwist

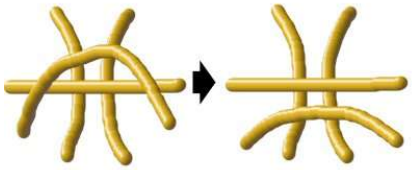


Figure 8 Powerpoke

The powertwist move looks for a region of the projection that is isolated from the rest of the knot by a single crossing and deletes both passings at that crossing. The implementation of the algorithm in Appendix I never actually executes a powertwist because whenever it finds such a situation it simply factors the knot around that point. Any situation that requires a powertwist actually contains a factor knot in the loop that is twisted, so the implementation factors it at this point so that it does not have to factor it later. To recognize a factor from a trace, the algorithm must find a subset of adjacent passings such that every passing in the subset has its mate in the subset.

The powerpoke move looks for a sequence of adjacent passings that have the same height with a passing before and after that have the same height as each other, though the height may be different from that of the middle passings. When this situation is found, the two outer passings are removed, and all the inner passings are moved to the other side of the strand that was poked. Moving passings is very difficult in a trace encoding because it is hard to tell which way the trace is traveling for another strand. For this reason, powerpoke is not yet implemented in the algorithm in Appendix I.

An algorithm that could perform both of these powermoves, would still not be able to untangle all projections of the trivial knot, although it would provide a quick way of analyzing many relatively simple cases. The fundamental problem arises from the inability to effectively manipulate strands or regions that are overlaid onto other regions of the knot. For example, consider a large composite knot with the trefoil as one of its factors. If the trefoil is overlaid onto the rest of the knot in the projection, it may be difficult to determine if it can be pulled off given just the trace encoding, even though it is obvious from the projection.

Conclusion

The experiment was not successful in finding an algorithm to determine the minimal crossing number of a knot from its trace encoding. However, the possibility for such an algorithm may still be open. In 1961, Wolfgang Haken proved that there is an algorithm for determining if a given projection is a projection of the trivial knot. Unfortunately, the algorithm is so difficult that it has never been implemented on a computer.[2] The result is nonetheless encouraging, and with some more work, new algorithms may avail themselves.

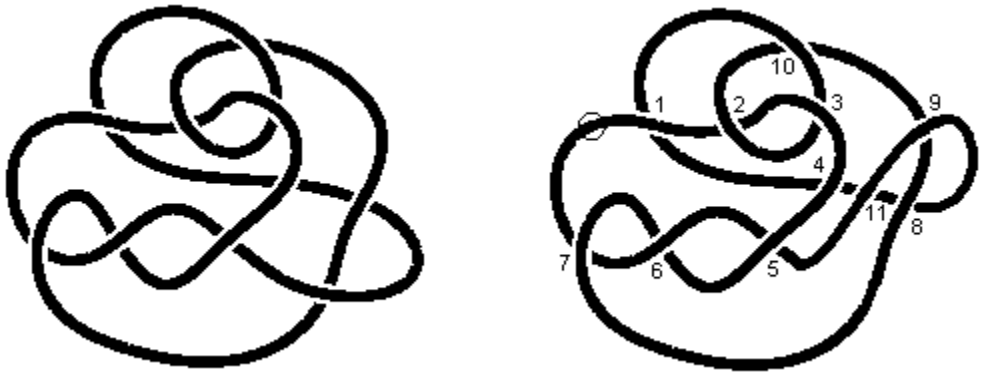
Bibliography

1. Livingston, Charles. Knot Theory. Washington, DC: The Mathematical Association of America, 1993.
2. Adams, Colin. The Knot Book. New York, NY: Henry Holt and Company, 2001.
3. Curtis, Fred. *Unknot Equivalence*. 25 Apr. 2004 <http://f2.org/maths/kt/unknoteq.html>

Figure Credits

Figures 2,3, and 6 are from The Knot Book.
Figure of "The Monster" in Appendix II is from
<http://www.cs.ubc.ca/nest/imager/contributions/scharein/thesis/fav.html>

Appendix II – Trace Encoding and Reduction of "The Monster"



The diagram on the left is a projection of the unknot called "The Monster". The program cannot reduce this trivial knot, but with a minor modification the program can handle it. The diagram on the right shows the modified projection that gives the loop between points 9 and 10 adjacent crossings on the underlying strand. With the given numbering of crossing points and starting from the circle, taking the orientation that goes right, the trace encoding yields
{{1,1},{2,0},{3,1},{4,1},{5,1},{6,0},{7,1},{8,1},{9,0},{10,0},{2,1},{3,0},{10,1},{1,0},
{4,0},{11,0},{8,0},{9,1},{11,1},{5,0},{6,1},{7,0}}
The first line of input to the program is the number of crossings, which is 11 for this case. The following lines are the trace encoding formatted as shown on the left.

Input:

```
11
1 1
2 0
3 1
4 1
5 1
6 0
7 1
8 1
9 0
10 0
2 1
3 0
10 1
1 0
4 0
11 0
8 0
9 1
11 1
5 0
6 1
7 0
```

Output:

```
Extended Poke found at points 10 9
Poke found at points 3 5
Poke found at points 2 6
Twist found at point 7
Factorization: Trivial
```

Progress:

```
#include <stdio.h>
#define MAX_CROSSINGS 256

void Abort(char *msg)
{
        fprintf(stderr,"%s\n",msg);
        exit(1);
}

typedef struct {
        int passings;
        int *point;
        int *height;
} knot;

int Point(knot* K, int index)
{
        int mod = K->passings;
        if(index < 0) return K->point[mod-(-index)%mod];
        return K->point[index%mod];
}
int Height(knot* K, int index)
{
        int mod = K->passings;
        if(index < 0) return K->height[mod-(-index)%mod];
        return K->height[index%mod];
}
int Passings(knot* K) {return K->passings;}
int Crossings(knot* K) {return (K->passings)/2;}
int abs(int x) {if(x>=0) return x; else return -x;}

knot* NewKnot(int passings)
{
        knot* K = (knot*)malloc(sizeof(knot));
        K->passings = passings;
        K->point = (int*)malloc(sizeof(int)*(passings+1));
        K->height = (int*)malloc(sizeof(int)*(passings+1));
        K->point[passings]=0;
        K->height[passings]=0;
        return K;
}

knot* InputKnot()
{
        int c,temp;
        scanf("%i",&temp);
        if(temp>MAX_CROSSINGS) Abort("Bad Knot: Too many crossings");
        knot* K = NewKnot(2*temp);
        for(c=0;c<Passings(K);++c)
        {
                scanf("%d",&temp);
                if(temp<1) Abort("Bad Knot: Can't use 0 as point");
                K->point[c] = temp;
                scanf("%d",&temp);
                K->height[c] = temp;
        }
        return K;
}

int Mate(knot* K, int index)
{       //returns index of other knot with the same Point
        int i;
        for(i=1;i<Passings(K);++i)
                if(Point(K,index+i)==Point(K,index))
                        return (index+i)%Passings(K);
        printf("Error: Point = %i\n",Point(K,index));
        Abort("Bad Knot: No mate found!");
}

int MatePoint(knot* K, int index)
        {return Point(K,Mate(K,index));}
int MateHeight(knot* K, int index)
        {return Height(K,Mate(K,index));}

void EraseKnot(knot* K)
{
```

```
        //K->passings = 0;
        free(K->point);
        free(K->height);
        free(K);
}

void RemovePassing(knot* K, int index)
{
        int i;
        if(index >= Passings(K)) index = 0; //modular trick used for SubKnot()
        for(i=index;K->point[i]!=0;++i)
                K->point[i] = K->point[i+1];
        for(i=index;K->point[i]!=0;++i)
                K->height[i] = K->height[i+1];
        K->passings -= 1;
}

void RemoveCrossing(knot* K, int point)
{
        int i,c=0;
        for(i=0;i<Passings(K);++i)
        if(Point(K,i)==point)
        {
                RemovePassing(K,i); //note: decrements Passings(K)
                i--; //check same index again after shifting
        }
}

void DisplayKnot(knot* K)
{
        int i;
        for(i=0;i<Passings(K);++i)
                printf("%i %i\n",Point(K,i),Height(K,i));
        printf("\n\n");
}

void RecordKnot(knot* K, int cmd)
{
        static knot* list[MAX_CROSSINGS];
        static int num = 0;
        int i;
        if(cmd==2) //display
        {
                printf("Factorization: ");
                for(i=0;i<num;++i)
                        printf("%i ",Crossings(list[i]));
                if(num==0) printf("Trivial");
                printf("\n\n");
        }
        if(cmd==9) //cleanup
        {
                for(i=0;i<num;++i)
                        free(list[i]);
                num = 0;
        }
        if(cmd==0) //record a knot
        {
                if(K==0)
                {
                        fprintf(stderr,"Null Knot!\n");
                        return;
                }
                if(Crossings(K)==0)
                        EraseKnot(K);
                else
                        list[num++]=K;
        }
}

int FindTwist(knot* K)
{               //Reidemeister Move I
        int i;
        for(i=0;i<Passings(K);++i)
        if(Point(K,i)==Point(K,i+1))
        {
                printf("Twist found at point %i\n",Point(K,i));
```

```
                RemovePassing(K,i);
                RemovePassing(K,i); //index i+1 gets shifted after first remove
                return 1;
        }
        return 0;
}

int FindPoke(knot* K)
{               //Reidemeister Move II
        int i,d;
        for(i=0;i<Passings(K);++i)
        if(Height(K,i)==Height(K,i+1))
        {
                d=Mate(K,i+1)-Mate(K,i);
                if(d==1 || d==-1)
                {
                        printf("Poke found at points %i %i\n",
                                        Point(K,i),Point(K,i+1));
                        d = Point(K,i+1);
                        RemoveCrossing(K,Point(K,i));
                        RemoveCrossing(K,d);
                        return 1;
                }
        }
        return 0;
}

void swap(int *a, int *b)
{
        (*a)=(*a)^(*b);
        (*b)=(*a)^(*b);
        (*a)=(*a)^(*b);
}

int CheckExtPoke(knot *K, int i)
{
        int start,end,j,d,h;
        int Removes[MAX_CROSSINGS];
        start=Mate(K,i);
        end=Mate(K,i+1);
        d=end-start;
        //Notice: Between the ends, there are N passings of the same
        //height. Each of these must have a corresponding passing that
        //is not between the ends. If they match up exactly, then
        //anything inside the loop is a standard poke, so that case
        //will be dealt with before coming to this function. So
        //we can assume that the subset of passings between the mates is
        //the direction that has the least number of passings.
        if((abs(d)>Passings(K)/2)^(d<0)) swap(&start,&end);
        //this just makes it so we can go forward through the passings
        h=Height(K,start+1);
        for(j=start+2;j<start+abs(d);++j)
        {
                if(Height(K,j)!=h) return 0;
        }
        printf("Extended Poke found at points %i %i\n",
                Point(K,start),Point(K,end));
        //store all the points that are between the ends
        for(j=start;j<=start+abs(d);++j)
        {
                Removes[j-start] = Point(K,j);
        }
        Removes[start+abs(d)+1] = 0;
        for(j=0;Removes[j]!=0;++j)
        {
                RemoveCrossing(K,Removes[j]);
        }
        return 1;
}


int FindExtPoke(knot* K)
{               //One of the strands can have passings between ends
                //as long as all have same height
        int i;
        for(i=0;i<Passings(K);++i)
```

```
        if(Height(K,i)==Height(K,i+1))
                if(CheckExtPoke(K,i)) return 1;
        return 0;
}

int AllZeroes(char *table)
{
        int i;
        for(i=0;i<MAX_CROSSINGS+1;++i)
                if(table[i]!=0) return 0;
        return 1;
}

int ClosedSubset(knot* K, int start)
{       //returns end of closed subset if one starts at start
        //or zero if one doesn't start here
        char *table = (char*)malloc(sizeof(char)*(MAX_CROSSINGS+1));
        int i;
        memset(table,0,MAX_CROSSINGS+1); //all entries are zeroed
        for(i=start;i<start+Passings(K)-1;++i)       //minus 1 for proper subset
        {
                table[Point(K,i)] ^= 1; //XOR the least bit
                if(AllZeroes(table)) //all points have been seen 0 or 2 times
                {
                        free(table);
                        printf("Found a closed subset between %i and %i\n",
                                        Point(K,start),Point(K,i));
                        return i; //may be greater than Passings(K)
                }
        }
        free(table);
        return 0;
}

knot* SubKnot(knot* K, int start, int end)
{       //removes a factor from K and returns it as a new knot
        int i;
        knot* F = NewKnot(end-start+1);
        for(i=start;i<=end;++i) //copy the factor
        {
                F->point[i-start] = Point(K,i);
                F->height[i-start] = Height(K,i);
        }
        for(i=0;i<end-start+1;++i) //remove the factor from the original
                RemovePassing(K,start); //knows when start >= Passings
        return F;
}

knot* FindFactor(knot* K)
{       //checks if a knot is composite (even if one factor is trivial)
        int i,end;
        for(i=0;i<Passings(K);++i)
        {
                end = ClosedSubset(K,i);
                if(end>0) return SubKnot(K,i,end);
        }
        return 0;
}

void ReduceKnot(knot* K)
{
        int proceed = 1;
        knot* F;
        while(proceed > 0)
        {
                proceed = 0;
                proceed += FindTwist(K);
                if(proceed) continue;
                proceed += FindPoke(K);
                if(proceed) continue;
                proceed += FindExtPoke(K);
                if(proceed) continue;
                F = FindFactor(K);
                if(F != NULL)
                {
                        proceed =1; //find more factors of K
```

```
                        ReduceKnot(F);
                }
        }
        RecordKnot(K,0); //record
}

void DisplayResults()
{
        RecordKnot(NULL,2); //display
        RecordKnot(NULL,9); //cleanup
}

int main()
{
        //tricky things - access to array is modular when using accessor
        //functions. Removing a passing shifts all the indexes down by 1.
        knot* K;
        K = InputKnot();
        ReduceKnot(K);
        DisplayResults();
        return 0;
}
```