A Further Thought on the Computational Complexity and Decidability of Unknotting Problem and Unknotting Number

Gabriel Chen[1*]

[1*]e-mail: gabrielchen@berkeley.edu.

Abstract. *We give out several further thoughts on the complexity and decidability of Unknotting Problem and Unknotting Number. This paper assumes no backgroud on Computational Complexity theory or Computability theory, and we start with the basic knowledge of NP completeness and P = NP problem in the first section. We introduce the Undecidability in topology in the next section and this will serve our thoughts on the decidability of Unknotting Number. In the third section, we formally go over the Unknotting Problem and the Unknotting Number. Then we give out some further thoughts on the Unknotting Problem and Unknotting Number regarding its decidability and NP completeness.*

Our main focus on the Unknotting Problem is the complexity analysis of Haken's Unknottedness Algorithm and Hass' Unknottedness checker. On the part of Unknotting Number, we give out our main thoughts that if there exists an algorithm which computes the Unknotting Number on a Turing Machine, the algorithm of computing Unknotting Number has a complexity at least as hard as the Unknotting Problem.

1. P and NP Complexity

In Computational Complexity theory, P stands for polynomial time and NP stands for nondeterministic polynomial time[1]. These two classifier are usually used for classifying search problems. We denote all search problems as class NP, and the class of all search problems that can be solved in polynomial time is denoted as P [Dasgupta et al. 2006]. Every NP problem has a polynomial checker to check the solution, and every P class problem can be implemented in polynomial time algorithm. It is quite easy to observe that P is in NP, however, whether $P = NP$ or $P \neq NP$ is still unsolved. In this paper, we assume that $P \neq NP$, in which case P is properly contained in NP.

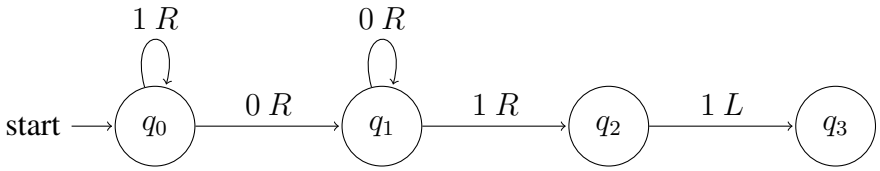
The main tool we use to show whether a problem is in P or NP is *reduction*. The general reduction method can be addressed as following: If there is an algorithm which can solve problem A, and there is a problem B can be reduced to problem A, the algorithm which can solve problem A can also solve problem B. On the other hand, if it is proved that there is no polynomial running time solver for problem A, there is no polynomial running time solver for problem B either. In which case, the process of finding whether a search problem is solvable in polynomial time, which means it is in class P, can be solved by reducing it to another solved search problem in P, or reduced to a problem in NP to

[1]A query machine is *deterministic* if its finite control specifies at most one possible operation for each configuration of the machine; otherwise the machine is *nondeterministic* [Baker et al. 1975].

show it is non-solvable in P. If a problem can be reduced to another problem in P, it is $P - reducable$ [Cook 1971]. In the following subsection, we will describe a classical reduction problem on Turing Machines, which is called the *Halting Problem*.

1.1. Turing Machine

A Turing Machine can be easily described as a Markov Chain. On each step, we do not care what happened before to decide our next action, and what action we are going to choose only depends on which state we are in right now. For a Turing Machine, it has a sequence of state, and between each state, we have the transition actions. A more concrete example is following Turing Machine which starts reading from the most left position of the tape and halts on the second group of 1's:



This abstract concept of Computability Theory can also be addressed in Logic sentences. Here we omit the formal philosophical definition and only use the above concept of Turing Machines to introduce the *Halting Problem*.

Definition 1.1. *The Halting Problem describes a halting function which is given a sequence of actions coded into a Turing Machine, and the halting function decides whether this program will halt or loop forever.*

More specifically, we construct a halting function $h(M, s)$ which takes a Turing Machine M and an input s for that Turing Machine as its input, and the output of this function will tell us whether that Turing Machine M will halt or loop on s.

Theorem 1.1. *The halting function is not Turing computable.*

Proof. We show this through assuming the reductio.

Assume there exists a halting function which computse the *Halting Problem*, which means there exist a function $h(M, s)$ which takes two inputs, a Turing Machine M and an input s for M, and $h(M, s)$ will output 1 if M halts on s, 0 if M loops on s.

We construct another function $selfhalts(M)$ which takes a Turing Machine as its input as following:

$$selfhalts(M) = \begin{cases} loops & \text{if } h(M, M) \text{ halts} \\ halts & \text{if } h(M, M) \text{ loops} \end{cases}$$

If we assume the halting function is Turing computable, the above $selfhalts(M)$ function is also Turing computable. Therefore, there is a Turing Machine computes it, and we can feed it into itself as a Turing Machine. If we feed in $selfhalts(M)$ to $selfhalts(M)$ itself, we have two cases, either it halts, or it loops forever.

$selfhalts(selfhalts)$ halts: In which case its means $h(selfhalt, selfhalt)$ halts from the definition of halting function. However, in order for $selfhalts(M)$ to halt, $h(M, M)$ must loop forever, which means $selfhalts(selfhalts)$ has to loop forever.

$selfhalts(selfhalts)$ loops: In which case its means $h(selfhalt, selfhalt)$ loops from the definition of halting function. However, in order for $selfhalts(M)$ to loop, $h(M, M)$ must halt, which means $selfhalts(selfhalts)$ has to halt.

Therefore, the halting function is not Turing computable.

■

From the *Halting Problem*, we observe that there are problems which are not computable by assuming Turing's thesis [Smith and Jones 1999], and if we can reduce any problem into these uncomputable problems, the initial problem is also uncomputable. We will visit this issue in the undecidability section and conclude that there are problems do not have an algorithm exist to compute it. Specifically, some problems in Topology can not have algorithms to compute them.

1.2. NP complete and NP hard

There are another two classes of problems worth mentioning. One of it is NP complete problems. NP complete problems are included in the class of NP, and as addressed above, all NP problems have polynomial checkers, but the essential thing which makes NP complete problems different is all other NP problems can be reduced to NP complete problems in polynomial time. In other words, if we could solve NP complete problems in polynomial time, we can solve all other NP problems in polynomial time.

In general graph theory, there is one problem which is shown to be NP complete, which is the *Subgraph Isomorphism Problem*. This problems address that given a graph G and another graph H, the algorithm of finding whether G contains a subgraph which is isomorphic to H is NP complete.

There are several ways to solve the NP complete problems, and there are two most used methods which are called *Approximation* and *Randomization*. By *Approximation*, we instead of looking for an algorithm to solve the problem, we look for an algorithm which approximate the result, and we want an approximate algorithm as close to solve the problem as possible. By *Randomization*, we introduce randomness to reduce the running time, and get an expected result in the end, and we would like the expected result as close to the optimal as possible same as the *Approximation* method.

Another class of problems worth mentioning is NP hard problems. In this class, all NP hard problems are at least as hard as NP complete problems. In which case, there are some problems which are not in class NP, but at least as hard to be solved as the hardest NP problems[2]. Since we assume $P \neq NP$, NP hard problems cannot be solved in polynomial time.

[2]One problem in NP hard problem set is *Traveling Salesman problem*, where a salesman has to figure out his route to visit each vertex on a weighted undirected graph exactly once and come back to his initial starting vertex. The problem is trying to find out the shortest path.

2. Undecidability in Topology

The decidability of a statement is whithin the language of certain theory, whether there is a procedures we can follow or feed it in a Turing Machine, the answer, or the output of the Turing Machine, is going to decide whether the statement is $true$ or $false$.

Definition 2.1. *An algorithm is a certain procedure we need to follow to solve a kind of problems.*

Easily to see, if the problem is decidable, there is an algorithm to solve it; if it is not decidable, no such algorithm guaranteed to exist.

Theorem 2.1. *If a problem is undecidable, there is no algorithm to solve it.*

It is very shocking when Godel first published his two incompleteness theorem showing that there are problems which are undecidable, which means we cannot prove or disprove something in Mathematics[3]. Same in Topology, there are many problems which are not decidable either, therefore, there is no algorithm exist to solve them.

Theorem 2.2 (Godel's First Incompleteness Theorem)**.** *Every consistent, aximotizable extension of Q is incomplete.*

In which case, if a theory T is an extension of Q, there are some theorems inside the theory which cannot be proved or disproved.

Also in Godel's second incompleteness theorem, he showed that a theory cannot prove its own consistency.

Theorem 2.3 (Godel's Second Incompleteness Theorem)**.** *Every consistent theory T which is an extension of Q, it does not prove its own consistency.*

In which case, $\nvdash_T Cons(T)$, and $\nvdash_T \neg Prov(\ulcorner 0 = 1 \urcorner)$

Below we list several problems in Topology which are not proved to be undecidable:

1. The homeomorphism between two finite simplicial complexes.
2. The homoeomorphism between a finite simplicial complex and a manifold.
3. The triviality of the fundamental group of a finite simplicial complex.

The main methods to show a new problem is undecidable is by assuming the reductio to raise a contradiction from our assumption. Also like the reduction method in complexity analysis, if a new problem can be reduced to any undecidable problems, it is also undecidable. And a more rigorous approach would be using the Diagonal Lemma and diagonalization to show the undecidability directly.

[3]Godel's first incompleteness theorem addressed this.

3. The Unknotting Problem

The Unknotting Problem describes that given a knot diagram D, how we could decide whether the knot addressed by diagram D is unknotted or not. The problem is showed with a complexity in class NP by Kuperberg [Baker et al. 1975], and there is an interesting corollary from Lackenby that if Unknottedness is NP complete, $NP = coNP$ [4].

The challenge of this problem is that *how we could tell two diagrams represents the same knots?* We know that even a trivial knots has infinite many diagrams which can represent it, but it is very likely that we are given a knot which is not isomorphic to the trivial knot. We can certainly apply Reidemeister Move to reduce the self intersection points on a knot diagram until the self intersections can not be reduced even one point, and if the result is a trivial knot, the knot is unknotted; otherwise it is not.

For recogonizing the trivial knots, which is also called $Unknottedness$, as address by Hass, if there exists an algorithm to output an unknot certificate, it can be run as following approach:

Unknottedness Checker

1. Given a link diagram D with n crossings certify that D is a knot diagram.

2. Construct a piecewise-linear knot K in R^3 which has regular projection D. From it construct a good triangulation $M_K \cong S^3 - \bar{R}_K$ which contains t tetrahedra, with $t = O(n)$, and with a meridian of ∂R_K marked in ∂M_K.

3. Guess a suitable fundamental solution $v \in Z^{7t}$ to the Haken normal equations for M_K. Verify the Haken quadrilateral conditions. Let S denote the associated normal surface, so $v = v(S)$.

4. Certify that S is an essential disk.

(4a) Certify that S is connected.

(4b) Certify that S is a disk. Check that $\partial S \neq \emptyset$ and that it has Euler characteristic $\chi(S) = 1$.

(4c) Certify that S is essential by checking that its homology class $[\partial S] \neq (0, 0)$ in $H_1(\partial M_K; Z/2Z)$.

With the same idea, Haken described an algorithm which takes an input as diagram D into a Turing Machine and outputs whether it is a diagram of trial knot [Haken 1961]. The algorithm process is reorganized by Hass as following [Hass et al. 1998]:

Haken Unknottedness Algorithm

Input: Link diagram D of crossing number n.

Question: Is D a knot diagram of the unknot?

1. Test if D is a knot diagram. If so, denote it K.

2. Construct a finite combinatorial triangulation T of S^3 which contains a polygonal knot K in its one-skeleton which has diagram K, and also contains a good compact

[4] co NP is the class of languages whose complements belong to NP [Baker et al. 1975]

triangulated 3-manifold $M_K \cong S^3 - R_K$, where $\bar{R}_K$ is a regular neighborhood of K. Let t denote the number of tetrahedra in M_K.

3. Construct an exhausting list L of vectors $L \subseteq Z^{7t}$ that contains $v(S)$ for some embedded compressing disk in M_K, if one exists.

4. For each $v \in L$, test if v is admissible. If so let S denote the normal surface with $v = v(S)$. Test if S is an essential disk for M_K.

(4a) Is S connected?

(4b) Is S a disk? Check that the Euler characteristic $\chi(S) = 1$ and $\partial S \neq \emptyset$.

(4c) Is S essential? Check that the homology class $[\partial S] \in H_1(\partial M; Z/2Z)$ is nontrivial by computing that its intersection number (mod 2) with a meridian of the 2-torus T_K is 1 (mod 2).

If all tests are passed for S, answer yes, halt and output $v(S)$.

5. If the complete list L is examined and no essential disk is found, answer no and halt.

Haken's Unknottedness Algorithm shows that the algorithm of recogonizing trivial knots exists, where he also proved that the unknottedness of a knot is decidable since there exists an algorithm for a Turing Machine to decide it. However, the runtime of this algorithm is horrible, which is exponential since it runs an exhausted list L to decide the unknottedness. In the following subsection, we will discuss the complexity of Haken's unknottedness algorithm.

3.1. Complexity Analysis

We will show that Haken's algorithm runs in exponential time complexity with polynomial space complexity.

Theorem 3.1. *The runtime complexity of Haken's Unknottedness algorithm is* $O(exp(n^2))$.

Proof. Let us first look at step 1 in the algorithm. In this step, we are just simply tracing all the edges of the diagram D, which takes us polynomial time. More specifically, we are tracing starting from each vertex, and we have n vertices in total. On each tracing, every time we make a decision, the remaining part of the decision process is narrowed down to a half. Therefore, the runtime complexity is $O(nlogn)$ for step 1.

For step 2, we would like to first introduce two triangulation lemma which we omit the proof of it here [Hass et al. 1998].

Lemma 3.1. *Given a link diagram D of crossing measure n, one can construct in time $O(nlogn)$ a triangulated convex polyhedron P in R^3 such that:*

(i) The triangulation has at most $420n$ tetrahedra.

(ii) Every vertex in the triangulation is a lattice point $(x, y, z) \in Z^3$, with $0 \leq x \leq 30n$, $0 \leq y \leq 30n$ and $-4 \leq z \leq 4$.

(iii) There is a link L embedded in the 1-skeleton of the triangulation which lies entirely in the interior of P, and whose orthogonal projection on the (x, y)-plane is regular and is a link diagram isomorphic to D.

Lemma 3.2. *Given a link diagram D of crossing measure n, one can construct in time $O(nlogn)$ a combinatorial triangulation of S^3 using at most $253440(n + 1)$ tetrahedra, which contains a good triangulation of $ML \cong S^3 - R_L$, with $\hat{R}_L$ a regular neighborhood of a combinatorial link L with link diagram D, and $\partial R_L = \partial M_K$. Furthermore one can construct in time $O(n2logn)$ in the triangulation marked sets of edges in ∂M_L for a meridian on each 2-torus component of ∂M_L, and a set of marked paths of $O(n)$ edges in $M_L \backslash \partial M_L$ that connect pairs of components of ∂M_L, which between them connect all components of ∂M_L.*

From above Lemma 3.1 and Lemma 3.2, we see that there are at most $253440(n+1)$ tetrahedra of the triangulated manifold M_K and the runtime for constructing them is $O(nlogn)$. Therefore, the runtime for this step is $O(nlogn)$.

For step 3, since we are just constructing a list L to exhaust all possible $v(S)$, and we know it is finite, this step takes us finite linear time.

Next we proceed to step 4. We first have to test whether for each tetrahedron in M_K at least two of the three quadrilateral variables vanish to see whether v corresponds to a normal surface. The runtime for this part is $O(t^{7t}2^{49t^2+14t})$ since we have t tetrahedras in M_K. Next we check whether S is an essential disk for M_K.

Lemma 3.3. *If S is a connected triangulated surface in R^3 whose Euler characteristic $\chi(S) = 1$ and $\partial S \neq \emptyset$, then S is homeomorphic to a disk.*

With Lemma 3.3, we know that if S is connected, we can check whether S is homeomorphic to a disk by computing $\chi(S)$ and $\partial S \neq \emptyset$. Since we can always add diagonals to quadrilaterals to form triangles, we can treat S as a triangulated surface. Then, we check $\partial S \neq \emptyset$ by looping v_i to check $v_i \neq 0$ to show that there is some triangulation has an edge in ∂M. For the Euler's characteristic, we can compute it directly from each v since its has all information we need, vertices, edges, and faces encoded inside. The process mentioned in this paragraph takes us $O(n^2logn)$ runtime in total.

By here, we have showed that S is a disk. Now we show the way to compute whether S is essential. We omit the homology computation here but if S is known to be a disk, then we can check whether ∂S is an essential curve in ∂M in polynomial time. More specifically, in $O(nlogn)$ time.

Combining the parts of computing $v \in L$ and certifying S as an essential disk, we have an exponential running time complexity as $O(exp(n^2))$.

■

3.2. Unknotting Number

There is another part Knot Theory which is brought up by the Unknotting Problem, which is the Unknotting Number of a knot. Let us have a look with the following knot 10_11[5]:

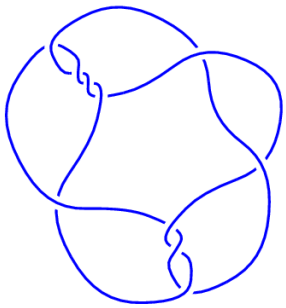


Figure 1. Knot 10_11

In knot 10_11, it crosses itself ten times, and this what we refer to *self intersection number*. In order to unknot it, we change its crossing and apply Reidemeister Moves until we get the trivial knot. The process of cross changing can be done in three crossing change as shown below, and the Reidemeister Moves can be easily applied to get the trivial knot there after:

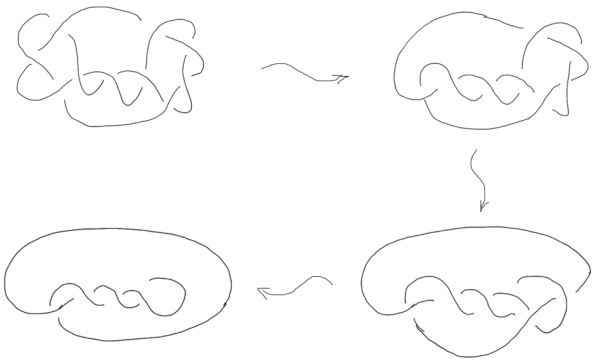


Figure 2. The process of unknotting the 10_11

Definition 3.1. *The Unknotting number of a knot is the smallest number of crossing change we have to perform in order to get the trivial knot.*

Since in the process of unknotting a knot, we have to terminate the algorithm when the knot is unknotted. Therefore, we at least have to recoganize the trivial knot to compute the unknotting number of a knot. We showed above that unknottedness is in NP, this follows,

[5]Figure created by Chuck Livingston with the assistance of Jae Choon Cha. J. C. Cha and C. Livingston, KnotInfo: Table of Knot Invariants, http://www.indiana.edu/ knotinfo, today's date (eg. May 8, 2019).

Theorem 3.2. *Computing the unknotting number is in NP.*

The challenge for computing the unknotting number has the same taste as the Unknotting Problem. *How could we tell two diagrams represents the same knots?* One naive approach for computing unknotting number on a Turing Machine could be:

1. Given a knot diagram D.

2. Apply Reidermister Moves until its has the least self intersections.

3. Classify the knots and brute force all possibility of crossing change.

4. Find the minimal number of crossing change to unknot the knot, which is the unknotting number of given diagram D.

From Haken's work, we know the knot is classifiable.

Theorem 3.3 (Haken)**.** *There is an algorithmic procedure to*

1. Recognize the Unknot

2. Classify knots

3. Compute the genus of a knot

4. Determine if a link is split

In step 3 we assume that reducing crossing number through Reidermister Move does not change the unknotting number. However, this might not be true [Abe et al. 2012].

Theorem 3.4. *There exists a knot K which does not have a minimal crossing diagram D of K with $u(D) = u(K)$.*

For example, the pretzel knot of type (5,1,4) does not have a minimal crossing diagram D of K with $u(D) = u(K)$.

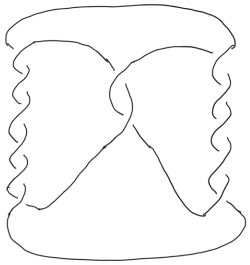


Figure 3. The pretzel knot of type (5,1,4)

We believe that the unknotting number is most likely being decidable and it is in NP. It requires more exploration into the problem further after this paper.

References

Abe, T., Hanaki, R., and Higa, R. (2012). The unknotting number and band-unknotting number of a knot. *Osaka J. Math.*, 49(2):523–550.

Baker, T., Gill, J., and Solovay, R. (1975). Relativizations of the p = ? np question. *SIAM Journal on Computing*, 4(4):431–442.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. (2006). *Algorithm*. McGraw-Hill Education, 1st edition.

Haken, W. (1961). Theorie der normalflachen: Ein isotopiekriterium fur den kreisknoten. *Acta Math.*, 105(3-4):245–375.

Hass, J., Lagarias, J. C., and Pippenger, N. (1998). The Computational Complexity of Knot and Link Problems. *arXiv Mathematics e-prints*, page math/9807016.

Smith, A. and Jones, B. (1999). On the complexity of computing. In Smith-Jones, A. B., editor, *Advances in Computer Science*, pages 555–566. Publishing Press.