A Short Guide to NP Completeness Proofs

R. Fink

September 1, 2023

1 Introduction

After searching for many minutes, and over several semesters, I thought I'd put together a practical guide for how to prove that a computational problem is in the set NP complete. I will cover only as much formal language theory as necessary; while important to understanding why we prove things this way, too many details often get in the way.

2 Definitions

We define some items and concepts prior to going through the steps. Even if you have seen these before, it's useful to review them in order because the concepts build and the proofs make free use of them.[1]

Classical computation is one that is performed on a standard, non-quantum, computing environment. (Quantum computing changes everything in this regard.)

Problem Something that can be solved with one or more algorithms. Examples include the graph shortest path problem, the Traveling Salesperson Problem, even "add two integers" is a problem.

Runtime Relative to an algorithm, the number of steps the algorithm takes to compute an answer for a given length of input. This discussion restricts runtimes to one of *polynomial*—taking $c_1 n^{c_2}$ steps—or non-polynomial including $n!$, c^n, n^n, and others.

Complexity Class of a problem is the runtime associated with the best known classical algorithm that solves the problem. For example, sorting a list of n integers is a problem that is computable in polynomial time (e.g., $n \lg n$, but there are faster algorithms.)

Turing Machine A simple construct that models all classical forms of computation. It consists of an input tape of cells, a cursor pointing to a cell, and simple operations like move tape, read cell, write cell, increment value at cell, and so on. Theory has it[2] that every efficiently computable function can be implemented in a Turing machine. The brainf*ck[3] programming language is a modern day workalike for a Turing machine (with a few more features).

[1] I am not a computability or language theorist. While I have tried to stay true to the core ideas, I cannot promise mathematical rigor. My aim is to give practical guidance and a basic understanding of the whole thing. So called *proofs* are subjective. If you follow this guide, I'll believe your proof.

[2] Full URL: https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis

[3] Full URL: https://esolangs.org/wiki/Brainfuck

Non-deterministic Relative to an algorithm being run through a Turing machine, one that relies on randomness to compute a solution in polynomial time; achieving an optimal solution therefore relies on "luck." This is the opposite of a deterministic algorithm that always makes the same decisions and produces the same output for a specific input.

Solution or *certificate* of a problem is one possible answer to a decision problem for a specific set of inputs. For purposes of NP completeness, these problems are structured as yes/no kinds of things. For example, a solution to the Traveling Salesperson Decision Problem[4] (TSP) is usually in the form, "given cities a, b, c, d, and distances between each one, one route covering all of them is b, c, a, d spanning d distance." This is easily verified by just visiting the cities in the stated order and adding up the distances.

In particular, a *solution* to decision problems like TSP is ***not*** claimed to be optimal, despite the point of writing such algorithms being to achieve an optimal outcome. A solution in this sense is simply a solution to the decision problem, a claimed arrangement that satisfies the yes/no statement. For TSP, the "solution" simply is that "this route takes d distance," not that "this route takes the *shortest* distance among all possible routes." This fact threw me for a time. Maybe that's why some people prefer the term *certificate* instead of *solution* to describe this idea. Note that the act itself of verifying a solution has an associated runtime.

NP The complexity class of *Non-deterministic Polynomial* decision problems. These problems can be solved (decided) in polynomial time *only if* using a non-deterministic algorithm (i.e., involving randomness). There is one important rule of NP decision problems: claimed solutions to them can be *verified* in polynomial—often linear—time.

X* ∈ *NP Problem X is in the set NP if and only if a solution to X can be verified in polynomial time. Therefore, the complexity class $P \subseteq NP$ because all polynomial time problems can be verified in polynomial time: just run the algorithm on the input and see that you get the same answer.

$\leq_P$ Polynomially reducible. If we say $X \leq_P Y$, that means there exists an algorithm that runs in polynomial time that converts an instance of problem X into problem Y. More formally, there exists some $f(X) \to Y$ (also written as $f : X \to Y$) where $f(X)$ runs in polynomial—often linear—time.

NPC or NP-Complete. A set of computing problems that are all NP, where each problem is reducible in polynomial time to at least one other in the set.

***NP*-hard** Only lightly relevant to our discussion, a problem that is at least as hard as any NP-complete problem, but that might not have an associated decision problem. Formally, saying that X is NP-hard means that there exists a polynomial time algorithm for reducing every $Y \in NP$ via $Y \leq_P X$, but that X might not have solutions that are verifiable in polynomial time. One example is the halting

[4]Full URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem

Figure 1: This cute frog is skeptical about all of this.

problem[5]. Put another way, a problem is NP-complete if it is both NP-hard, and has solutions that are verifiable in polynomial time.

3 Proving NP-Completeness

Here are a set of steps that produce mathematically acceptable proofs of a problem $X \in NPC$ by starting with some known $Y \in NPC$. This has been cobbled together from notes, the CLRS book [CLRS09], posts to StackExchange[6], and other places.

1. Prove $X \in NP$ by describing a polynomial-time algorithm that verifies a solution of X. Often, this algorithm runs in linear time.

2. Identify a known problem $Y \in NPC$. Typically, you seek a problem that closely resembles X; for instance, if X has to do with graphs and edges, look to `VERTEX-COVER` or `INDEPENDENT-SET`.

3. Describe an algorithm that maps Y into X in polynomial time. Formally stated, prove that X is NP-hard by showing $Y \leq_P X$. Note the order here - we go from known to unknown, and we have to do it in polynomial (often linear) time.

4. Prove that a solution of X maps to a solution of Y and vice-versa, in polynomial time:

    a. Find a polynomial-time function f that maps a solution of problem X to a solution of problem Y. In math-ese, $\exists f \mid x \in X \iff f(x) \in Y$.

    b. Find a polynomial-time function g that does the reverse of f, mapping a solution of Y to a solution of X.

    c. Make sure you discuss the running time of each, verifying that these functions run in polynomial (could be linear) time.

    This step usually is straightforward, especially if you've chosen a Y that looks a lot like X.

[5]Full URL: `https://en.wikipedia.org/wiki/Halting_problem`
[6]Full URL: `math.stackexchange.com`

You can conclude your proof of the above with a summary statement that says if X can be solved in polynomial time, then Y can be solved in polynomial time, because we have a polynomial mapping function that can change the solution of X to the solution of Y, and that would mean $P = NP$. This contradicts with the current belief that $P \subset NP$. Since this is generally believed, this step is left out of most proofs.

4 Example

Far be it for me to prove a theorem stated in [CLRS09], but here goes. I'll restate their proof more explicitly in terms of the steps above.

Theorem 34.14. The traveling salesperson problem is NP-complete.

Proof:

1. `TRAVELING-SALESPERSON`, which we will call TSP, is in NP: given a certificate consisting of an ordering of cities, the distances between each pair of cities, and a claimed distance for that particular ordering, we start at one end of the ordering and sum up all the distances between. If our total comes to the claimed distance, we have verified that this ordering is a solution to the decision problem. The act of moving from one city to another takes place in $O(v)$ time, with v being the number of cities, and is therefore linear in the numer of cities.

2. Per [CLRS09], we will choose `HAM-CYCLE`, shortened to HC, where $HC \in NPC$.

   (The rest of this proof is adapted from [CLRS09] with some added explanations.)

3. We prove $HC \leq_P TSP$. Let $G = (V, E)$ be an instance of HC—that is, G is a graph that has a Hamiltonian (Ham) cycle in it. Note that G in this case is not a complete graph, whereas with TSP we only have complete graphs. Therefore, we construct a new graph $G' = (V, E')$ with edges $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, and define a cost function $c(i, j)$ where $c(i, j) = 0$ if edge $(i, j) \in E$. Basically, an edge in G' has zero cost if that edge is part of the Ham cycle in G. Such a construct basically spits out a complete graph G' in polynomial time: for vertex in G', create an edge to every other vertex in G', which runs in $|V|^2$ time, and adding the costs is done inline in that same algorithm.

4. We show that a solution to HC maps to a solution of TSP and vice-versa, and it does so in polynomial time. Specifically, we prove that graph G has a Hamiltonian cycle if and only if graph G' has a tour with cost 0.

   a. We show an algorithm that maps solutions $HC \iff TSP$ that runs in polynomial time. Given a graph $G = (V, E)$ that has a Ham cycle h in it, copy all V vertices to a new graph G', producing vertices V' in linear time. Next, connect all V' vertices together creating the set of edges E', and also assign a cost of 1 to every edge for now; this takes $\Theta(n^2)$ time. Next, for each edge $(i, j) \in h$, set the cost $c(i, j) = 0$ in $O(1)$ time. This algorithm works by stepping through all vertices and edges; because it must visit each vertex/edge only once, it runs in linear time.

b. The reverse algorithm works the same way, and visits vertices/edges once, running in linear time, except that it starts with a full graph of TSP and creates edges in h only where $c(i, j) = 0$ for edges (i, j). This results in a graph that is a Ham cycle.

c. We have shown the linear runtime, but just to make sure - each transformation function processes edge/vertex pairs only once. It never has to revisit them, so they each run in linear time.

We shall skip the final concluding statement. It's good to have confidence; no need to be arrogant about it.

5 Frequently Asked Questions

Why do we care about NP Completeness?

Some problems are harder than others to solve. In computer science, that translates to some problems being computationally more difficult, time consuming, or resource consuming than others to solve. The concept of NP Completeness tells us when we've landed on one of these computationally difficult problems and that we won't find an easy / fast / resource efficient algorithm for solving it.

Why do we do reductions?

We are trying to relate something we know to something we don't know. In the case of NP Complete problems, we are trying to relate a problem known to be in NPC to our new problem. By doing so, we formally prove that there exists no efficient (polynomial time) algorithm for solving our problem as long as nobody finds an efficient algorithm for solving the other problems in NPC.

How do reductions work?

They change an instance of one problem into another. Summarizing from above, we reduce a known NPC problem Y into our unknown problem X by showing that we can change all the specifics of problem Y into those of problem X. Usually, it's things like "every counselor and sport becomes a vertex, and a link between the two means this counselor teaches this sport."

We define a function that does that transformation - could just be a couple of sentences describing the function in general - and show that this function runs in polynomial time.

What do reductions actually prove?

Reductions prove that X and Y both belong to the set NPC. ~~That means that since there is no polynomial time solution for Y, there is no polynomial time solution for X.~~ *Wait - close, but wrong.* It proves that **if** someone finds a polynomial time algorithm to solve Y, then they can use it to solve X in polynomial time. However, such an algorithm would take them both out of the set of NPC, and in fact topple the entire concept of NPC meaning that $P = NP$.

Why do we need polynomial time verifiers?

To prove we *solved* the problem. Given a certificate of a solution to some problem

X, we say $X \in NP$ if there exists a way to verify that solution in polynomial time. Why does it have to be polynomial time?

Solving a problem in computer science means giving an answer that requires less work to understand than the work it took to solve it. If we had a solution but no method to verify the solution in polynomial time, that means we would have to use a non-polynomial algorithm to verify the solution. Since we probably used a non-polytime algorithm to *get* the solution, and we need a non-polytime algorithm to *verify* the answer, then we haven't really solved the problem at all.

It would be like saying "you want to check my answer? Then solve it yourself!"[7]

Why do we need polynomial time transformers?

Because if someone finds a fast solution to an NPC problem, we want to use it to solve our hard problem efficiently. If we have a polytime (polynomial time) algorithm that creates a solution of Y, and want to solve X, we would do this: (1) find a transformer function that changes solutions of Y into solutions of X, (2) run your polytime solver on Y to get S_y, then (3) run your transformer on S_y to produce the solution to X.

The total time to solve X is the sum of the time to solve Y plus the time to transform the output to a solution of X. If that transformer runs in polytime, then we get poly + poly = poly. **If,** however, that transformer takes ***non-polynomial time*** to complete, then the sum of the runtime is itself non-polynomial.

I'm from another planet. I just discovered a *shark*. I know a bear is dangerous, is a shark also dangerous?

Welcome to the planet. You'll find it odd that many people are unhappy despite the fact that the government is filled with clowns.[8] Anyway, we can prove that a shark is in the set of dangerous animals:

1. If someone shows me a photo of a shark attack, you can tell with one glance that a shark can cause major damage.
2. You know that a bear is dangerous.
3. Imagine you are magic. A bear is fast on land - if you take a bear, and change its legs into a strong back fin, and change fur to smooth skin, you can make it fast in water. A bear has enormous teeth - take bear teeth and convert it to shark teeth. A bear is aggressive toward land animals - change the target of that aggression to water-based things. Congratulations, you now have a shark in just a couple steps.
4. Given a shark attack, you can see the victim encroached on its territory and therefore suffered big bite marks. Change the territory from the ocean to the forest, and pretty much keep the bite marks - congrats, you have a bear attack in just a few simple steps. Likewise, given a bear attack, change it to a shark attack in much the same way.

[7] This is the basis of many cybersecurity assumptions, e.g., cryptographic hash functions. To get the same output, you need to do the same work.

[8] Credit - Mork and Mindy TV show.

Since we can transform a bear's properties into a shark, and we can change a bear attack into a shark attack, (and that the transformations are generally 1-for-1,) there's no doubting that sharks are very dangerous as well. Further, if you were able to become faster / thicker skinned enough to survive a bear attack, then you could use those same attributes to survive a shark attack, too.

References

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.