Demystifying P vs NP and bridging gaps between Proof Complexity and Computational Complexity

Luke Tonon

December 2024

1 Introduction

The P vs NP problem stands as one of the most significant and enduring open questions in Mathematics and Computer Science. At its heart lies a deceptively simple inquiry: can every problem whose solution can be verified in polynomial time also be solved in polynomial time? Despite decades of research, the problem remains unresolved.

We revisit the fundamentals of computation through a new lens by reframing the transition function of Turing Machines. This perspective provides new insights into the limits of computation, offering a framework for understanding the difficulty of solving any decidable problem. The goal of this work is not merely to address P vs NP but to explore how unifying reasoning and computation may resolve long-standing mysteries in complexity theory.

2 Revising Fundamentals

We'll revisit the definition of a Turing machine given in **THE P VERUS NP PROBLEM** [2], but with some slightly different formatting to induce clarity of a new view.
A Turing Machine is frequently defined as a 7-tuple, given by:
$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$
Q : Finite set of states.
Σ : Input alphabet.
Γ : Tape alphabet ($\Sigma \subseteq \Gamma$).
δ : Transition function ($Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$) where L means move tape head one cell to the left and R means to the right.
q_0 : Start state.
$q_{\text{accept}}, q_{\text{reject}}$: Accept and reject states.

A Turing Machine is a mathematical model for computation that determines, step by step, whether a problem can be solved by systematically transitioning between states according to its transition function.

2.0.1 The Transition Function: The Program of the Machine

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

δ is deterministic, as when given the current state and tape symbol, it uniquely determines the next state, tape symbol, and head movement.

However, there is a deeper way of examining what δ is doing by reframing it. If the machine is in state q and reads symbol s, then it transitions to q', writes s', and moves L or R.

Symbolically, it is:
$(q \wedge s) \implies (q' \wedge s' \wedge d)$ where $d \in \{L, R\}$ is the direction of the tape head movement.
Each step in the computation corresponds to applying a logical rule encoded in δ. This means each step in the entire computation of a Turing Machine is a deterministic reasoning chain, progressing step by step through logical inferences. The transition function applies repeatedly, creating a reasoning chain:

$$(q_0 \wedge s_0) \implies (q_1 \wedge s_1 \wedge d_1) \implies \cdots \implies (q_{\text{halt}} \wedge s_{\text{final}})$$

q_i: The state of the Turing machine at step i.
s_i: The symbol being written or read at step i.
d_i: The direction (left or right) the tape head moves after step i.

Each state transition is like an inference step.
The final state is the conclusion of the reasoning. Or, more intuitively, the machine is performing many “if-then” steps. This means that the operation of a deterministic Turing Machine is functionally equivalent to the systematic reasoning within a propositional formal system.

2.0.2 The Transition Function Inherits Limits of Reasoning

The framing of the transition function as a process of reasoning implies Turing Machines inherit the limitations of formal systems. If a problem requires exponentially long reasoning chains, no deterministic algorithm can solve it efficiently because the Turing Machine cannot complete the computation in polynomial time.

We can now connect this to P vs NP. P problems are deterministically solvable in a polynomial growing amount of time. NP problems are where solutions can be verified through reasoning in a polynomial growing amount of time.

This allows us to explain what every "clever shortcut" to solve a problem has been all this time. They are powerful logical deductions that let many truths be propagated across the constraints that help guarantee the answer to a problem is correct every time. Propagating truths refers to the process of deducing new facts or solutions based on existing constraints and logical rules.

Knowing this, it becomes easier to find the most efficient algorithms to solve problems deterministically. "Does a clever shortcut exist?" is equivalent to asking "Are there logical deductions that always propagate enough truths to make solving a problem require less steps?". This newer question has a much clearer answer.

Now, we recall that the limits of propositional reasoning, such as in Frege proofs, are known for resolving some truths, such as tautologies, including the Pigeonhole Principle. Thus, a contradiction becomes apparent.

Frege proofs are among the most expressive systems for propositional logic, yet they require exponentially long proofs for certain truths. This demonstrates the inherent difficulty of propagating truths deterministically through reasoning.

If a polynomial deterministic solver existed for SAT, the satisfiability problem, it would be capable of reasoning through some truths to reach any other decidable truths in a polynomial growing number of steps without potentially creating false solutions, but this contradicts many known lower bounds of reasoning in propositional systems.

A deterministic SAT solver would have to overcome the limits of propositional reasoning to be polynomial growing, but this is impossible. Therefore, such a solver cannot exist, the SAT problem cannot be in P and was the first NP-Complete problem [1], so $P \neq NP$.

We can also explain why the other NP-Complete problems are hard as efficient shortcuts that always propagate enough truths to make solving a problem require significantly less steps simply do not always exist. These shortcuts, if they existed, would need to compress reasoning chains that are known to be exponentially long in certain cases, such as Frege proofs.

3 Beyond P vs NP: A New Lens for Complexity Theory

The primary goal of this paper is to resolve P vs NP, but whether the reasoning presented here ultimately succeeds may not be as important as the **new way of analyzing deterministic algorithms revealed here**. The details of this new process of analyzing deterministic algorithms will not be formally explored here and is left open for others to experiment.

Instead of past algorithm discovery primarily involving inventing "clever shortcuts" and trying them, one can now break every decidable problem into an analysis of what needs to be true to satisfy the problem's solution constraints.

By examining the patterns in the shortest reasoning chains that satisfy the problem's solution constraints, the most efficient deterministic algorithms essentially reveal themselves. **In hindsight, it would've been nice to know this decades ago to save a lot of trial and error!**

The technical details of how this process would work would probably be best discussed in more papers and opens up a new path of exploration in complexity theory.

If one applies this process, it becomes clear why many of the problems previously not in P, and some variants of NP-Complete problems, entered P, such as 2-SAT and XOR-SAT. Logic is a computer's greatest strength, but also its greatest weakness. It's **reasonable** to consider this analysis technique will answer many other Complexity Class questions.

References

[1] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)* (1971), pp. 151–158. DOI: `10.1145/800157.805047`.

[2] Stephen A. Cook. "The P Versus NP Problem". In: *The Millennium Prize Problems.* Clay Mathematics Institute Millennium Prize Problem Description. URL: `https://www.claymath.org/wp-content/uploads/2022/06/pvsnp.pdf`.