

The P versus NP Problem

Dean Casalena
University of Cape Town
CSLDEA001
dean@casalena.co.za

Contents

1. Introduction
2. Turing Machines and Syntax
 - 2.1 Overview
 - 2.2 Turing Machine Syntax.
 - 2.3 Polynomial Time and More Elementary Syntax.
 - 2.4 Computable Enumerability and Decidability
 - 2.5 Turing Machine Augmentation
3. P and NP
 - 3.1 P (and PSPACE)
 - 3.2 NP
4. P versus NP
 - 4.1 Many-One Reducibility
 - 4.2 Polynomial Time Reducibility and NP-Completeness
 - 4.3 Some Conclusions
5. In Practice
 - 5.1 Overview
 - 5.2 SAT (Boolean Satisfiability Problem)
 - 5.3 The Theorem of Cook
6. Application
 - 6.1 Feasibility
 - 6.2 Implications of $P=NP$
 - 6.3 Implications of $P\neq NP$
7. Conclusion
8. Bibliography

1. Introduction

We consider various decision problems in mathematics where given an input we seek an output of YES or NO. In particular, we consider “resources” needed to obtain such output, and hence we determine the practical feasibility of finding a solution.

In order to gain sight of where this rather vague problem description leads, we look at one such problem, that of prime factorization.

To put it simply, it can be significantly harder to factorize a number p into primes, than check if two primes multiplied together equals p . Lets take a $2n$ -digit number p and try to factorize it by simply dividing it by every integer less than p until we find one which divides p exactly. This could take up to roughly 10^n computations. However, if we started with the correct guess, it would take very few computations to verify that it was indeed a factor.

There are well defined classes of problems which can be solved within specific computational constraints and others, of which the solution can be verified under the constraints. Broadly, the P versus NP question explores the possibility that certain problems with easily verifiable solutions, could in fact be deterministically solved “from scratch”, as easily.

2. Turing Machines and Syntax

2.1 Overview

We need to define computation and accurately. To do this we make use of the standard theoretical computer model for mathematics, the Turing Machine. As you will see, while the model is primitive, its important properties are adequately preserved as we add a level of abstraction and the jump is made to modern computing in a physical sense, i.e. Turing machines can represent (broadly speaking) any other more complex computing models, in at least a way efficient enough for the classes P and NP to remain unchanged.

A Turing Machine reads in and potentially replaces symbols on a two-way tape, by performing pre-programmed instructions which consider the current state of the Turing Machine and the current symbol being read in. It then moves either one symbol to the left or right. Given an input, a Turing Machine will terminate in accepting or rejecting state, or continue computing forever. Turing Machines are deterministic and have only one possible move for each pair of state and input symbol.

2.2 Turing Machine Syntax.

Each Turing Machine has, associated with it, a finite set of 2 or more symbols known as its input alphabet. For a particular alphabet A , then we denote with A^* the set of all strings containing only characters from A . A “language over A ” is a subset of A^* . A language may for example be defined as

$$L = \{x \in A^* \mid x \text{ has property } Q\}.$$

A Turing Machine with input alphabet A has a computation associated with each string $x \in A^*$.

We say a Turing Machine M accepts a string x , if its computation associated with x terminates in the accepting state. Remember, the Turing Machine’s possible outcomes for a computation are accept, reject or continue forever. Should M continue forever in its computation on x , then M does not accept x . The language accepted by M is denoted $L(M)$ and

$$L(M) = \{x \in A^* \mid M \text{ accepts } x\}$$

where A is the input alphabet of M . Again, if we say “ M accepts x ”, we are already only considering strings on which M ’s computations halt at some point. Furthermore the computation halts in the accept state.

2.3 Polynomial Time and More Elementary Syntax.

We have hence isolated the strings causing termination in the accept state, but we need to further distinguish between such strings in terms of resource bounds on the Turing Machine. Here we are dealing with Turing Machines with two way infinite tape from which to read the symbols of the string but we will limit the time allowed for computations. For a Turing Machine M , with input alphabet A , $t_M(x)$ is the number of steps in M ’s computation on x , the number of moves M makes. Should the computation never terminate, $t_M(x) = \infty$.

We can also consider the set of strings of length n , denoted A^n , and $T_M(n) = \max\{t_M(x) \mid x \in A^n\}$ which is called the worst case run time of M given strings of length n consisting only of symbols from M ’s input alphabet. We generalize and say that M runs in polynomial time $T_M(n) \leq n^k + k$ for all $n \in \mathbb{N}$ and some k .

To summarize: A Turing Machine M runs in polynomial time if it makes $|x|^k + k$ or fewer moves in the computation on any string $x \in A^*$, where k is fixed for M and $|x|$ is the integer length of x .

2.4 Computable Enumerability and Decidability

Let $S \subseteq A^*$. S is computably enumerable (c.e.) if and only if there is a Turing Machine with input alphabet A , such that $S = \{x \in A^* : x \text{ is accepted by the TM}\}$ i.e. S is the enumeration of all the strings accepted by some Turing Machine. To abstract, if a set can be completely listed, then it is c.e.

Let $S \subseteq A^*$. S is decidable if and only if there is a Turing Machine M with input alphabet A , such that $S = \{x \in A^* : x \text{ is accepted by the TM}\}$ and every $x \in A^*$ is either accepted or rejected by M i.e. The Turing Machine will never compute forever given any $x \in A^*$, and it will furthermore end in a rejecting state for any $x \notin S$. Thus S is decidable if there exists a Turing Machine that can decide, by way of an accept or reject state, whether or not a string x is in S .

A more general description of decidability states that a set G is decidable if and only if its characteristic function $\text{cfg} : A^* \rightarrow \{0, 1\}$, where $\text{cfg}(x) = 0$ if $x \in G$ and $\text{cfg}(x) = 1$ otherwise, is computable - i.e. if there is a computable function to assign a one or a zero to an instance x based on whether or not it belongs to G , then G is decidable.

If S is decidable then it is also computably enumerable, since from the above definitions c.e. is listed as the first condition on a set's decidability. The converse, that a c.e. set is also decidable, is not necessarily true. We do however have that if a set S is c.e. and its complement in A^* is also c.e. then S (and A^*/S) is decidable.

2.5 Turing Machine Augmentation

The Turing Machine model is augmented to modern computing, where very many reading head and tape equivalents work together, to create a more powerful and efficient computing device. For the P versus NP question and indeed throughout discrete computational complexity theory (in particular on that without resource bounds) what is important is that these properties of computable enumerability and decidability on subsets of A^* are preserved. The c.e. and decidable sets are the same whether you not you use "Turing Machine" in the definition as opposed to some other computer.

Church's Thesis, though unproven, sums this up into exactly what we would hope to hear, at least for the sake of Mathematical convenience: "Any reasonable model of discrete computation can be simulated in the Turing Machine model."

Stephen Cook states, "...it turns out Turing Machines can generally simulate more efficient computer models... ..by at most squaring or cubing computation time" and this would indicate that the statement about a computation running "within polynomial time" is also preserved, albeit a different polynomial.

3. P and NP

3.1 P (and PSPACE)

We now have the necessary terminology to define the complexity class P. P stands for Polynomial Time. To put it loosely, P is the set of polynomial time decidable languages (by a deterministic Turing Machine).

Remember a language L is a subset of A^* for a Turing Machine's input alphabet A, and for L to be decidable, there must exist a Turing Machine that accepts or rejects every input x, based on whether or not x belongs to L. In order for L to be polynomial time decidable, we further add the constraint that the Turing Machine runs in polynomial time for any input $x \in A^*$.

$P = \{L \mid L = L(M) \text{ for a Turing Machine } M \text{ which runs in polynomial time}\}$

which means:

$P = \{\text{all the languages } L_i \text{ such that } x \in L_i \text{ implies that } x \text{ is accepted by some Deterministic Turing Machine } M_i \text{ which runs in polynomial time (for any input instance } z \text{ from its input alphabet } A_i^*, t_{M_i}(z) \leq |z|^k + k \text{ for some } k)\}$

We define the complexity class $DTIME(f(n))$ to mean the set of languages for which there exists a Turing Machine M that runs in time of $O(f(n))$.

$O(t) = \{g: \mathbb{N} \rightarrow \mathbb{N} : (\exists c)(\forall n) f(n) \leq ct(n) + c\}$

Hence, if a language belongs to the complexity class $DTIME(n^k)$, then there exists a Deterministic Turing Machine which can complete computations on its elements in polynomial time.

Compiling the above definitions we arrive at another notation for P.

$$P = \bigcup_{k=0}^{\infty} DTIME(m^k)$$

A Turing Machine M runs in polynomial space if for any string $x \in A^n$, M visits $n^k + k$ or fewer squares of tape. A language L for which there exists a Turing Machine M such that $PSPACE = \{L \mid L = L(M) \text{ for a Turing Machine } M \text{ which runs in polynomial space}\}$ is in the complexity class PSPACE. We can think of P as PTIME with a similar definition to the above.

The number of tape squares visited is doubtlessly a smaller number than the number of moves made by the Turing Machine in total, which implies that if a computation runs in polynomial time, it also runs in polynomial space. Thus we have $PTIME \subseteq PSPACE$.

Computable enumerability and decidability can be defined against any form of computation; not just Turing Machines. We can use the following definition for P

$$P = \{L \mid L \text{ is deterministically decidable in polynomial time}\}$$

For the purposes of real-world application, we can talk about decidability of problems by algorithms on a more abstracted level and define P informally as the set of problems for which “efficient” algorithms exist.

3.2 NP

In order to define and describe the class of languages, NP, we must investigate nondeterministic Turing Machines. In a deterministic Turing Machine, each pair of current input symbol and current state will cause a specific instruction to be carried out. For each pair of current input symbol and current state in a nondeterministic Turing Machine, there are a number of possible instructions that can be carried out. Thus while a deterministic Turing Machine follows a linear program path, a nondeterministic Turing Machine can terminate in many different states, after different numbers of moves, and having followed many different program paths.

If we say a computation on x by a Nondeterministic Turing Machine runs in polynomial time, it means that at least one program path caused termination in the accepting state in polynomial time. NP stands for Nondeterministic Polynomial Time. We define $L(M)$ for nondeterministic Turing Machines as follows:

$$L(M) = \{x \in A^* : \exists \text{ an accepting computation path for } x\}$$

Also, as with P, we have complexity class-based definition for NP.

We define the complexity class NTIME as with DTIME but with the essential difference that NTIME uses Nondeterministic Turing Machines, instead of Deterministic, in the definition.

If a language L belongs to $NTIME(n^k)$, then there exists a Nondeterministic Turing Machine N with input alphabet A which runs in polynomial time for any $x \in A^*$ and accepts any $x \in L$ in polynomial time.

We we arrive at another notation for NP, that of

$$\text{NP} = \bigcup_{k=0}^{\infty} \text{NTIME}(m^k)$$

3.3 $P \subseteq NP$

One obvious, yet important thing to notice is that $P \subseteq NP$. This is because any language L in P has a deterministic Turing Machine M which can decide $x \in L$ in polynomial time, and a non-deterministic Turing Machine N thus also exists where one computation path matches that of M . Thus we have the following, with the inclusion based on the above:

$$P = \{L \mid L = L(M) \text{ for a Turing Machine } M \text{ which runs in polynomial time}\} \subseteq \{L \mid L = L(M) \text{ for a Nondeterministic Turing Machine } N \text{ which runs in polynomial time}\} = NP$$

So any language computable in polynomial time by a Deterministic Turing Machine can be computed in polynomial time by a Nondeterministic Turing Machine. This follows intuitively from the definitions of Deterministic and Nondeterministic Turing Machines and is easy to prove.

4. P versus NP

4.1 Many-One Reducibility

We have that any problem in P is also in NP so if we could prove there is a language in NP not in P , it would mean $P \neq NP$. Alternatively if we could prove that there is no language in NP which is not in P , i.e. $NP \subseteq P$, then it would mean $P = NP$.

So it would be beneficial if we could find a way of grouping together certain NP languages / problems which are not known to be in P . To do this we explore the notion of reducibility, first in terms of computational complexity.

Consider the languages $G \subseteq A^*$ and $H \subseteq B^*$. “ G is reducible to H ” is expressed by $G \leq_m H$, which we write when there exists a computable function $f: A^* \rightarrow B^*$ such that $x \in G \Leftrightarrow f(x) \in H$ for all $x \in A^*$. This type of reducibility is called many-one reducibility.

Lemma 4.1.1

Computable functions are closed under composition. If g and f are two computable functions then $g \circ f$ is also a computable function.

(proof omitted)

If we have $G \leq_m H$ and H is decidable, then G is decidable.

Proof:

$G \leq_m H$ means that there exists a computable function $f: A^* \rightarrow B^*$ such that $x \in G \Leftrightarrow f(x) \in H$ for all $x \in A^*$.

$x \in G$ then $f(x)$ can be computed. $f(x) \in H$ is decidable which means there exists a computable characteristic function $\text{cfh}: B^* \rightarrow \{0, 1\}$ defined by $\text{cfh}(f(x)) = 0$ if $f(x) \in H$ and $\text{cfh}(f(x)) = 1$ otherwise.

By function composition, we obtain $\text{cfh} \circ f(x): A^* \rightarrow \{0, 1\}$ defined by $\text{cfh} \circ f(x) = 0$ if $x \in G$ (which is if and only if $f(x) \in H$) and $\text{cfh} \circ f(x) = 1$ otherwise.

By lemma 4.1.1 the function $\text{cfh} \circ f(x)$ is computable. Thus we have a characteristic function for G . This satisfies the conditions for G to be decidable.

Wherever $G \leq_m H$, we cannot have that H is decidable and G is undecidable. So if $G \leq_m H$ and G is undecidable, then H is undecidable.

We call a language L c.e.-complete if it is c.e. and every other c.e. language is reducible to L .

4.2 Polynomial Time Reducibility and NP-Completeness

NP-completeness parallels c.e.-completeness but the definitions are over a different reduction – that of polynomial time reduction instead of the many-one reduction used the previous definitions.

Polynomial time reduction (or p-reduction) is denoted $G \leq_p H$ for languages $G \subseteq A^*$ and $H \subseteq B^*$ where G is p-reducible to H . For $G \leq_p H$ there must exist a polynomial time computable function $f: A^* \rightarrow B^*$ such that $x \in G \Leftrightarrow f(x) \in H$ for all $x \in A^*$.

A language H is NP-complete if and only if H is in NP and $G \leq_p H$ for every language G in NP.

4.3 Some Conclusions

Lemma 4.3.1

Polynomial time computable functions are closed under composition. If g and f are two polynomial time computable functions then $g \circ f$ is also a polynomial time computable function.

(proof omitted)

If $G \leq_p H$ and $H \in P$ then $G \in P$.

Proof:

$H \in P$ means $H = L(M_1)$ for some Turing Machine M_1 which runs in polynomial time. Alternately we can say that H is polynomial time decidable.

Let $G \subseteq A^*$, $H \subseteq B^*$ and $G \leq_p H$.

$G \leq_p H$ means that there exists a polynomial time computable function $f: A^* \rightarrow B^*$ such that $x \in G \Leftrightarrow f(x) \in H$ for all $x \in A^*$.

$x \in G$ then $f(x)$ can be computed in polynomial time. $f(x) \in H$ is decidable in polynomial time since H is in P which means there exists a polynomial time computable characteristic function $\text{cfh}: B^* \rightarrow \{0, 1\}$ defined by $\text{cfh}(f(x)) = 0$ if $f(x) \in H$ and $\text{cfh}(f(x)) = 1$ otherwise.

By function composition, we obtain $\text{cfh} \circ f(x): A^* \rightarrow \{0, 1\}$ defined by $\text{cfh} \circ f(x) = 0$ if $x \in G$ (which is if and only if $f(x) \in H$) and $\text{cfh} \circ f(x) = 1$ otherwise.

By lemma 4.3.1 the function $\text{cfh} \circ f(x)$ is polynomial time computable. Thus we have a characteristic function for G and satisfy the conditions for G to be polynomial time decidable. Thus $G \in P$.

If G is NP-complete, $H \in NP$, and $G \leq_p H$ then H is NP-complete.

Proof:

G is NP-complete so G is in NP and for any arbitrary $L \in NP$, $L \leq_p G$.

$G \leq_p H$ (given)

$L \leq_p G$. So there exists a polynomial time computable function f such that $x \in L \Leftrightarrow f(x) \in G$

$G \leq_p H$. So there exists a polynomial time computable function g such that $f(x) \in G \Leftrightarrow g(f(x)) \in H$

So $x \in L \Leftrightarrow f(x) \in G \Leftrightarrow g(f(x)) \in H$

And $x \in L \Leftrightarrow g \circ f(x) \in H$. Since $g \circ f$ is a polynomial time computable function, by lemma 4.3.1, $L \leq_p H$.

So we have that H is in NP and any $L \in NP$ is reducible to H .

This makes H NP-complete.

Since all problems in NP are p-reducible to NP-complete problems, if we were to find a language L that is NP-complete and in P, it would mean that all languages in NP would be reducible to L which would make all these languages elements of P and so P would equal NP. This same conclusion can be reached by noting that the set of NP-complete languages is closed under p-reducibility as is P. Thus if these sets shared any elements, they would be equal.

5. In Practice

5.1 Overview

“In practice, a member of NP is expressed as a decision problem and the corresponding language is understood to mean the set of strings coding YES instances to the decision problem using standard coding methods.” - Steven Cook

The properties of NP-complete problems as described previously, in particular with reference to reducibility means that if somehow someone could find a deterministic polynomial-time algorithm to solve one of the NP-complete problems, that same algorithm could be modified to solve all the other NP-complete problems in polynomial time, and we would have that $P=NP$. This is however not the only direction being taken to answer the P versus NP question. In fact the general consensus is that P and NP are not equal. It is a “general” consensus though, as there is still very much uncertainty over the question. It is possible that someone finds the algorithm in P for an NP-complete problem, though many people over decades have searched for faster algorithms with or without the knowledge of complexity classes.

Alternatively someone may otherwise prove that $P=NP$ in such a way that polynomial time algorithms for NP-complete problems never see practical implementation, or perhaps just not for a long time still to come.

More likely, (just by way of researched survey of opinions) it will be proven that P cannot equal NP. To do this one may just need to prove that a single NP-problem can

never be solved in polynomial time deterministically.

5.2 SAT (Boolean satisfiability problem)

We turn to explore the NP-complete problem of Boolean satisfiability. An instance of the problem, for example may look as follows:

$$F = A \wedge B \wedge C \vee \neg C$$

or

$$F = A \wedge B \wedge \neg A$$

The SAT question is simply:

Is a formula F satisfiable?

Now the question is easy to answer for formulae with few variables, since we can use the obvious method of systematically trying every possible assignment to satisfy it. If one such assignment succeeds in satisfying the formula, then we yield a YES answer, if none, NO. A very concrete algorithm exists to perform these tests on a computer.

So what's the problem?

If a formula is unsatisfiable, the algorithm will try every possible assignment on the variables before terminating. This makes the worst-case run time for an SAT instance with n variables at least 2^n computations. I say *at least* 2^n since for each assignment, more than one computation may need to be carried out in calculation. The worst-case run time is still $O(2^n)$. For a small 100 variable formula, around 2^{100} computations may need to be run; that's 1 267 650 600 228 229 401 496 703 205 376 computations.

If the problem were given to a theoretical Nondeterministic Turing Machine, it could in a sense take 2^n computation paths for its first computation, one for each possible solution. The existence of one accepting computation path in polynomial time would make the problem in NP, but the worst case run time here is $O(1)$ which is clearly within polynomial time. So SAT is undoubtedly in NP. Given the correct solution as part of the input, SAT is solved in polynomial time.

5.3 The Theorem of Cook

The theorem of Cook (Stephen Cook) is a proof that SAT is NP-complete. Another mathematician, Leonid Levin, also proved the same at around the same time. The

theorem is also known as the Cook-Levin theorem.

The proof is very detailed. We expect at least once proof of NP-completeness to involve very low level Nondeterministic Turing Machine algorithm mechanics. This is because it is generally easier to reduce a known-to-be NP-complete problem to an NP problem in order to prove a problem NP-complete. In order to use this approach however, we must have at least one problem we have proven by some other means, to be NP-complete.

The idea behind the proof is that for any arbitrary Nondeterministic Polynomial time Turing Machine N , given an input string x , a formula is created which is satisfiable if and only if N accepts x . This would show that any arbitrary problem in NP is reducible to SAT. Since we already have that SAT is in NP, this completes the proof.

To prove that a new problem is NP-complete we only need to reduce SAT to that problem, since we have already seen that reducibility is transitive.

6. Implications of a Solution

6.1 Notion of Feasibility

In computational complexity, the feasibility of a solution is often paired with whether or not it will run in polynomial time. This does not always mean feasible in a semantic sense, since polynomial time algorithms of $O(n^{300})$, whilst technically in “polynomial time” will not be feasible even for $n=2$. Regardless, if the algorithm of $O(n^{300})$ replaces one of $O(k^n)$, it will still be faster for inputs of lengths greater than 300.

6.2 Implications of P=NP

The main implication of P=NP is the range of problems to which solutions would become feasible. Every NP-complete problem would become solvable in polynomial time. This includes a range of optimization problems which would allow for absolute efficiency in various practical industrial scenarios. Growth in development of artificial intelligence would make significant leaps forward.

Computers would be able to construct certain proofs in polynomial time, since proofs of a particular form can be verified in polynomial time by computers. This would allow further unsolved problems to be solved. Thus a proof that P=NP would additionally have far reaching consequences in indirect ways.

Cryptography is largely based on the assumption that it will remain very difficult to algorithmically determine an encryption key for a set of data. Worst case of guessing an

alphanumeric password 10 letters long is $10^{(26+26+10)}$ attempts, which is an impossibly difficult scenario. If one already knows the password it would take just one guess. $P=NP$ would allow for a polynomial time solution to determining the key. One can imagine the obvious consequences.

6.3 Implications of $P \neq NP$

Implications of $P \neq NP$ would not be as interesting. The method for proving this may be revolutionary in itself, but to have proven that there is no easier way to solve certain problems is not a victory for practical application.

7. Conclusion

Much work has been devoted to solving the P versus NP problem and it is doubtlessly not just because of the \$1 million reward for a proof. Despite this as well as the very importance of the problem (in a day and age where more and more polynomial time solutions are becoming practically feasible on high powered computers) the problem remains open, and much uncertainty surrounds it. Many scholars believe $P \neq NP$, but some believe the contrary, and not for lack of education.

Hopefully a solution will be discovered soon and traveling salesmen will be able to efficiently decide if their trip can be a specific distance, and their bag can be exactly the right weight.

8. Bibliography

Cook, Stephen. "The P versus NP Problem". University of Toronto

Friedman, Harvey M. "Clay Millenium Problem: $P = NP$ " Mathematics Colloquium, Ohio State University notes. 20 Oct 2005

Moshkovitz, D. Complexity. <http://www.cs.tau.ac.il/~safra/Complexity/Cook.ppt>

"NP (complexity)." *Wikipedia, The Free Encyclopedia*. 27 Jul 2006, 17:41 UTC.

Wikimedia Foundation, Inc. 7 Aug 2006

http://en.wikipedia.org/w/index.php?title=NP_%28complexity%29&oldid=66196083

"P (complexity)." *Wikipedia, The Free Encyclopedia*. 21 Jul 2006, 21:51 UTC.

Wikimedia Foundation, Inc. 7 Aug 2006

http://en.wikipedia.org/w/index.php?title=P_%28complexity%29&oldid=65106234