P vs NP

Andzu Schaefer

January 2017

Abstract

Our approach to the topic of P vs NP is intuitive with regards to describing the classes P and NP, as well as to describing the inherent computational nature of these classes and the nature of NP-completeness. We also discuss and categorize sub-optimal solutions, as well as 'progress in the field' such as proofs concerning proofs and the impact of other fields of study. We conclude by summarizing the far reaching consequences of both P=NP and P≠NP. We do all this in a manner that is detailed enough for readers to move on to more formal sources.

1 An intuition behind P and NP

How do we quantify how "hard" problems are? If the problem is abstractable to the point where we can solve it with an algorithm, we ask: what is the most efficient algorithm that solves this problem? Once an algorithm is found, we can assess how hard the problem is by looking at how much time and space the algorithm needs. Given the low cost of computation, we're capable of solving many problems that were, until recently, out of reach; consider the fact that smartphones contain processors that are many orders of magnitude faster than the computers essential to the Apollo program. However, there are still many problems that are out of reach to even the most powerful computers. In fact, modern encryption relies on the fact that it's effectively impossible to factor the product of large primes. The question of what is in our reach, and what is not, is the question at the heart of $P \stackrel{?}{=} NP$. What is feasible, and what is effectively impossible?

2 Definitions

What are P and NP? We'll begin by defining the sets P and NP, then continue into a few brief explanations in order to properly understand the two sets.

We can identify problems by two criteria: problems that are easy to solve, and problems that, if given a solution, are easy to check to see if the given solution is valid.

Definition 2.1 (P) *The set of problems with input size n such that there exist valid-solution-generating algorithms with worst-case run time bounded by n^k, where k is any positive integer.*

P is short for polynomial time and refers to the fact that polynomial time algorithms that solve these problems are known. Problems in P include sorting numbers, finding the greatest common divisor of two numbers, and testing for primality. Given that every 'action' taken by an algorithm requires one standard unit of time, the time it takes to run is measured by the number of actions (or steps, or calculations) the algorithm takes. For example, consider finding the product of two matrices A and B:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & \dots & x_{nn} \end{bmatrix} \quad Y = \begin{bmatrix} y_{11} & y_{12} & y_{13} & \dots & y_{1n} \\ y_{21} & y_{22} & y_{23} & \dots & y_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & y_{n3} & \dots & y_{nn} \end{bmatrix}$$

If we want to compute their product, we find entry XY_{ij}, the entry in the ith row and jth column, by computing the sum of products $x_{i1}y_{1j}+x_{i2}y_{2j}...+x_{in}y_{nj}$. This calculation requires n multiplication operations and n addition operations. So the total number of actions needed to find entry XY_{ij} (or any entry) is $2n$. Since there are n^2 entries to be found, the total number of actions needed to compute XY is $2n * n^2 = 2n^3$. We say the cost of this operation is $O(n) = n^3$. n^3 is certainly a polynomial, thus computing the product of matrices is in P.

Why do we draw the line at polynomially-bounded runtimes? Clearly, certain polynomials will have early outputs much greater than certain exponential functions (x^{100} vs 1.00000001^x, for example). However, polynomials grow less quickly than exponential functions:

$$\lim_{x \to \infty} \frac{x^n}{r^x} = 0$$

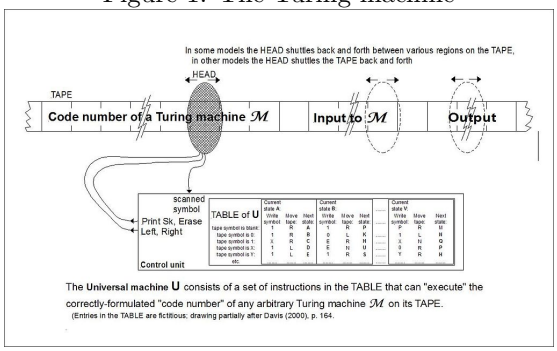
for all positive real numbers n, r, meaning that as problems get larger, at some point it will always be more efficient to use a polynomial-time algorithm as opposed to an exponentially bounded algorithm. [2]

Definition 2.2 (NP) *The set of problems with input size n, which, if given a solution, there exist valid-solution checking algorithms with runtime bounded by n^k, where k is any positive integer.*

NP is short for Nondeterministic Polynomial time. Notice that since problems in P can be solved in polynomial time, solutions can certainly be verified in polynomial time, which means that P is a subset of NP. Problems that exist in NP include Sudoku and the Traveling Salesperson problem (which we discuss in later sections). It is unknown whether these problems exist in P.

The role of Turing machines

Figure 1: The Turing machine



[6]

To formalize the difficulty of problems that we use computers to solve, we need to first formalize the computers themselves. The Turing machine, invented by Alan Turing in 1936, satisfies the needs of formalized computing using an infinite tape, composed of segments that are either blank or contain a symbol. The machine reads one symbol at a time, and according to a set of rules, writes a symbol to the square, changes state, and moves either left or right along the tape. The states serve as a form of memory: a Turing machine 'remembers' what symbols it has read in the past through its state.

More formally, we can describe a Turing machine as a structure M such that

$$M = \{Q, \sigma, \gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}\}$$

where Q is a finite set of states, σ is a finite input alphabet, γ is a finite tape alphabet containing σ (γ denotes a different alphabet that the Turing machine uses to compute the output on the tape, hence the phrase 'tape alphabet), δ is the transition function

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

and $q_{start}, q_{accept}, q_{reject} \subset Q$ are the starting, accept, and reject states, respectively.

Runtime and feasibility

Essentially, any understanding of theory of computation will allow us to understand the basic ideas surrounding runtime and feasibility, thanks to the Church-Turing thesis, which states that virtually any model of computation will be equivalent to Turing Machines.

Definition 2.3 (Extended Church-Turing Thesis) *Any formalism of deterministic digital computation can be represented by Turing machines, and vice versa.*

[1]

We can abstract the elements of P and NP as languages.

Definition 2.4 (alphabet) *A finite set of symbols.*

Definition 2.5 (Language) *Given an alphabet A, a language L is a set of finite strings, where each string is composed of symbols from A.*

Each "word" in the language corresponds to a particular instance of the problem in question; the symbols and the order they are in contain all the information necessary to recreate a certain 'setup'. Clearly, many strings composed of the alphabet of a certain language will be gibberish, but just because a string successfully encodes a problem state doesn't mean it's interesting. Rather, because we are interested in finding and verifying solutions, we'll decide membership by doing just that.

2.1 Formal definitions of P and NP

Definition 2.6 (P) *A string S is in a language L if the deciding Turing Machine M run on the string S such that M eventually halts and enters an 'accept' state in polynomial time. We say that M is polynomial time if, for all strings that could possibly be members of a language, M enters either an accept state or a reject state after at most $P(|S|)$ steps, where P is a polynomial and $|S|$ is the length of S. A language is in P if there exists M such that the above condition is satisfied.*

[1]

Definition 2.7 (NP) *A language L is in NP if for every S in L there exists a verifier V such that there exists a polynomial time Turing Machine M such that $M(V, S)$ accepts if S is in L and rejects if S is not in L.*

[1]

A note on the meaning of 'non-deterministic':

If deterministic algorithms answer the question "how do we set the inputs so that a certain output is output", then non-deterministic algorithms answer "CAN we set the inputs so that a certain output is output?". If we can abstract problems into decision trees, then algorithms are essentially the set of rules that we follow in order to arrive at the correct destination. Every IF statement in an algorithm's pseudocode can be represented by a fork in the road, if certain conditions are fulfilled, take a certain path, otherwise take the other.

But how do non-deterministic algorithms find solutions? Non-deterministic algorithms are not bound by the need to choose between paths whenever a fork

is presented, the algorithm can take multiple possible paths. Implementing such an algorithm would be horribly expensive, because even if the algorithm travels from the root to the leaf containing the correct solution in polynomial time, it would travel to every single leaf.

3 NP completeness: from SAT to Minesweeper

We'll illustrate NP-completeness and the relationship between all NP-complete problems by encoding a few NP-complete problems into one another in a manner that utilizes the Turing-machine-based formalization of computation.

3.1 SAT

SAT, short for "the logical satisfiability of a Boolean condition", is one of the simplest known NP-complete problems.

Definition 3.1 (SAT) *A statement composed of variables and boolean operators, evaluating to either True of False. Basic boolean operators include* $\neg, \vee, \wedge$.

The follow are examples of statements that SAT asks us to evaluate:

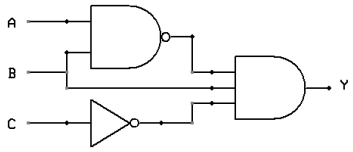
1. $a \vee b \vee c \vee d$
2. $a \wedge (b \vee c) \vee (\neg d)$
3. $a \wedge (b \vee (c \wedge d))$

We'd like to convince the reader that SAT is NP complete, which we'll accomplish on a high level by introducing the CIRCUITSAT.

3.2 CIRCUITSAT

Turning our attention to logic gates and circuits, we notice that we can compose circuits that are analogous to any boolean expression.

Figure 2: This circuit is analogous to the boolean expression $\neg(A \wedge B) \wedge (\neg C)$.



To get a clearer idea of what this implies, we'll begin by defining a few terms that will be useful in formalizing CIRCUITSAT.

Definition 3.2 (Boolean Domain) *A set with 2 elements with interpretations True and False.*

Computer scientists commonly use $\{0, 1\}$ as the boolean domain.

Definition 3.3 (Boolean Function) *a function mapping inputs from B, a Boolean domain, to B.*

Examples include AND, OR, NOT, and various combinations of these gates. generally speaking any logic gate can be constructed from these 'atoms'.

Definition 3.4 (Directed Acyclic graph (DAG)) *a finite directed graph with no directed cycles.*

The DAG is essential to representing the 'flow' of signal in a circuit: signal moves in one direction from input gates to output sink.

Definition 3.5 (Boolean Circuit) *Define B as a set of boolean functions that correspond to gates allowed in the circuit model. A Boolean circuit is a finite directed acyclic graph, with each vertex corresponding to a function in B, an input, or an output.*

Definition 3.6 (CIRCUITSAT) *Given a boolean circuit, is there an assignment of truth values True or False to the inputs such that the output of the circuit is True?*

Just as we substitute values to evaluate SAT expressions, we can evaluate CIRCUITSAT solutions in P time by simulating the circuit with the given input. Most importantly, because Turing machines were created to formalize computers, they are representative of gates, the most basic components of computers. For any problem we can represent on a computer, we can create an analogous CIRCUITSAT gate, where the input would be an encoding of a solution and the output would be whether the solution is correct.

3.3 MCP$\in$NP

Minesweeper is a game in which a player tries to guess the location of all mines on a grid covered with tiles. By flipping a tile, either the game ends, if a mine is underneath, or the number of mines in the eight adjacent squares is given to the player to aid them in their task. We'll be examining the Minesweeper consistency problem (MCP), which asks if a given minesweeper configuration is logically consistent. We say a configuration is logically consistent if the information revealed by flipping an unmined tile 'agrees' with that of its neighbors. The configuration shown in figure 3 does not abide by the rules of minesweeper, and so we call this configuration logically inconsistent. Specifically, we know that there are no mines in the squares adjacent to the square displaying '1', there cannot exist six mines in the square adjacent to the square displaying '6' (since it only has 3 adjacent neighbors, and the square marked '2' has five mines adjecent. We can show that MCP can be used to 'encode' the SAT, which implies that if we can find a polynomial time algorithm that solves MCP, then we can solve the SAT.

Figure 3: A logically inconsistent configuration, assuming mines lie in flagged squares.

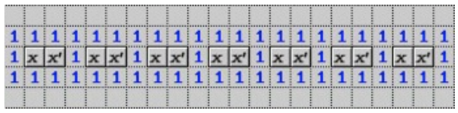


Figure 4: The Minesweeper wire propagates a 'signal' along its length

However, some cases aren't as clear cut. Consider Figure 3, the wire, along with the boolean function

$$f(b) = \begin{cases} True & \text{square b contains a mine} \\ False & \text{square b does not contain a mine} \end{cases},$$

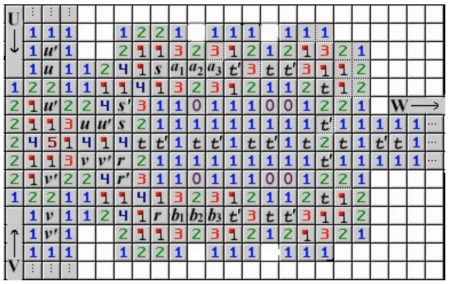
where b is given square on a minesweeper grid. A mine must be present in either squares marked x or x' but not both and not neither:

$$f(x) = \text{NOT} f(x')$$

In effect, the wire carries a signal: the mine's presence of lack therof. However, without more information, we cannot determine what 'state' the wire is in.

Mathematician Richard Kayne has shown that minesweeper arrangements exist for NOT, AND, OR, and a variety of other boolean gates. [3]

Figure 5: The AND gate.



Hence, we can take any iteration of the SAT with output f, and create a corresponding minesweeper grid, where the square x_{final} represents the output.

If we are capable of solving the minesweeper consistency problem? Then we set $f(x_{final}) =$ True and check consistency and immediately we get whether the grid is consistent, or it's even possible for the grid to output True, which is part

of the question SAT asks. In order to verify a layout is consistent, we'd also need to "state" of the "inputs" that give a consistent layout, which would give us the inputs that would produce True if plugged into the corresponding SAT.

Note converting the SAT into the MCP is a polynomial time problem. **So if you can solve MCP in P, then you certainly can solve SAT in P.**

3.4 NP completeness

So what does creating logic gates out of minesweeper arrangements have to do with P and NP? It turns out the ability to encode one problem in another, as we've just shown with MCP and SAT, is inherent to a rather important subgroup of problems in NP.

Definition 3.7 (NP Hard) *A problem is NP-Hard if, given a polynomial-time solution, one can find a polynomial time solution to all problems in NP.*

The entirety of NP would be solved in polynomial time if a solution to a single problem in NP-hard can be found in P-time.

Definition 3.8 (NP Complete) *Problems contained in both NP and NP-Hard are NP-Complete.*

Problems in NP-hard that aren't NP are easy to envision if one keeps in mind the characteristics of problems in NP. Problems in NP must be decision problems, must have a finite number of solutions, and must be verifiable in P. time. Chess, for which we cannot check moves in P time, is an NP hard problem.

4 Feasible alternative solutions

Given a problem, assuming the problem in question is NP-hard and that $P \neq NP$, then there is no algorithm that is deterministic, 100% accurate, and always efficient. We will now explore approaches to NP problems that are feasible, but at the cost of sacrificing accuracy and dependability. We will explore the following topics:

- Approximation algorithms
- General heuristics
- Pseudopolynomial-time algorithms

A variety of NP-complete problems exist for which there exist algorithms that give solutions that are bounded by a factor of the optimal solution, which we call approximation algorithms. For example, consider the Travelings Salesperson Problem. The Traveling Salesperson Problem asks for a route, given a set of cities, that satisfies the conditions set forth by the salesperson. The salesperson needs to travel to every city, and would like to do so by traveling through each

city only once. The salesperson needs to end their route in the same city they started in.

If the graph satisfies the triangle inequality, then we can find a route which has cost of at most 3/2 that of the optimal route using the Christofides Algorithm.

Definition 4.1 (Triangle Inequality) *A complete weighted graph satisfies the conditions of the Triangle Inequality iff weight*$(u, v) \leq$ *weight*$(u, w)+$*weight*(w, v) *for all vertices* u, v, w

The performance ratio $r(A)$ of an approximation algorithm A is a metric used to gauge the ability of an approximation algorithm. We can think of the performance ratio as the lower bound of how suboptimal the solutions provided by an algorithm are.

Definition 4.2 (Performance Ratio) *Given a problem asking to maximize the output (colloquially, profit)* P*, let that problem's Objective Function be a function that maps the algorithm* A *used on instance* I *to* $P \in R$ *where* P *is the* ***profit****.*

$$OBJ : A \times I \to R, \quad OBJ(A \times I) = P$$

Let $OPT(I) = OBJ(A_{optimal}, I)$ *denote the cost of solving* I *using the (theoretical) optimal algorithm* $A_{optimal}$*.*

We say that the ***performance ratio*** $r(A)$ *of an approximation algorithm* A

$$r(A) = \min_{I} \frac{OBJ(A, I_{\min})}{OPT(I_{\min})}$$

is equal to the ratio $\frac{Cost_A}{Cost_{A_{optimal}}}$ *on instance* $I_{\min}$ *which produces the smallest value of* $\frac{Cost_A}{Cost_{A_{optimal}}}$*.*

A similar ratio is used for minimization problems, but instead the ratio represents an upper bound of cost, instead of a lower bound of profit.

Definition 4.3 (Christofides Algorithm) *Suppose* G *is a graph representative of an instance of the traveling salesperson problem. Each node represents a city, each edge represents a route between cities, and each edge is weighted with a value representative of travel cost.*

1. *Create a minimum spanning tree* T *of* G*.*
2. *Let* O *be the set of vertices with odd degree in* T*. Since* O *has an even number of vertices, it is possible to find a minimum-weight perfect matching* M *in the subgraph given by the vertices from* O*.*
3. *Combine the edges of* M *and* T *to form graph* H*.*
4. *Form an Eulerian circuit in* H*.*

5. *Make the circuit found in previous step into a Hamiltonian circuit by skipping repeated vertices (shortcutting).*

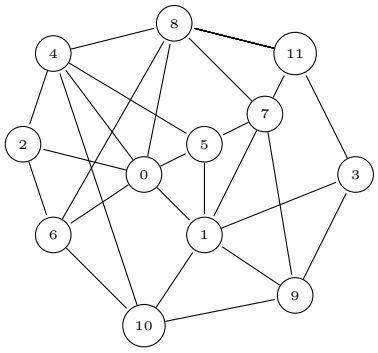


Figure 6: An instance of the Traveling Salesperson Problem. All edges are weighted equally.

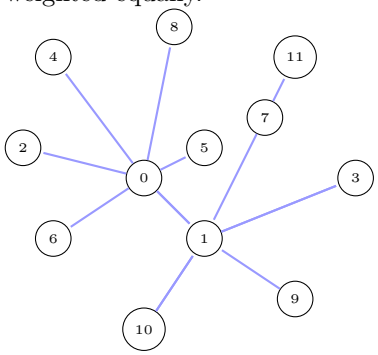


Figure 7: Minimum weight spanning tree T

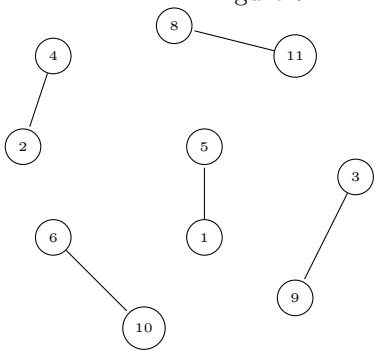


Figure 8: A minimum weight perfect matching M in the induced subgraph of vertices of O, the set of vertices with odd degree in T.

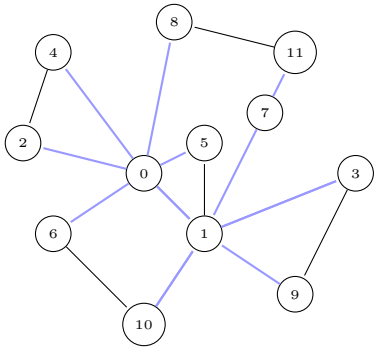


Figure 9: Combine the edges of M and T to form a connected multigraph H

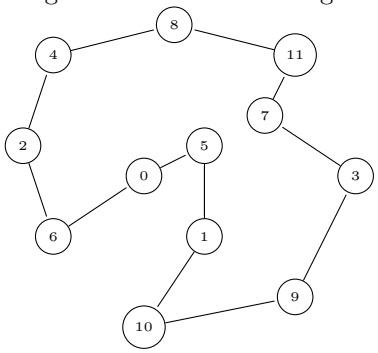


Figure 10: Form a Eularian circuit and take shortcuts to get the final route.

We'll use figures 11 and 12 construct a visual proof that the cost of the route found using the Christofides algorithm is 3/2 that of the optimal route. Suppose we know the optimal route. We can form C, a spanning tree, by removing an edge from the route. We can form M using the technique described in figure 12. By choosing the correct partition, the cost of traversing M is at most 1/2 that of C. We compose these graphs to form H, the cost of which is the sum of the costs of the component routes.

Alternatively, consider the Knapsack Problem, in which we are asked to pack a knapsack with items. Each item has a specific value and weight. The backpack has a weight limit. How do we pack our knapsack so that the total value of items packed is maximized?

It's possible to approximate the best answer to an arbitrary degree of precision in polynomial time (runtime for these algorithms is a function of both the size of the input n as well as the degree of precision ϵ).

Definition 4.4 (A Greedy Algorithm for the Knapsack Problem) *for all $O(kn^k)$ subsets of objects with up to k objects, put the subset in knapsack, then sort all remaining objects in terms of benefit/weight ratio in descending order, and fill the knapsack. Choose the subset that maximizes total value.*

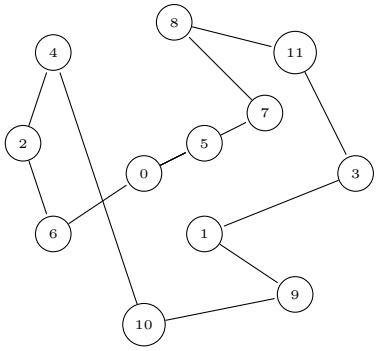


Figure 11: A given "optimal" route

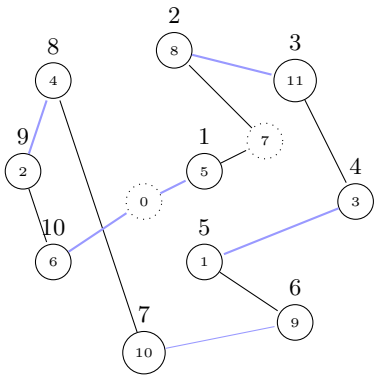


Figure 12: Form O from the spanning tree, and then label vertices in cyclic order around C. Partition into sets of paths starting at odd or even vertices. Note each set forms a perfect matching.

Theorem 1 *For a fixed integer K, the above algorithm has a performance ratio of* $\frac{k}{k+1}$. *[4]*

The above algorithm's runtime is polynomial both in terms of input and error.

Unfortunately, there isn't any unifying theory that allows for us to easily characterize problems as easily approximated or hard to approximate.

Approximation Algorithms are a specific type of heuristic, which are approaches to a problem that often (not always) return decent results, or rules of thumb that speed up the process of searching haphazardly.

The Branch and Bound process is a heuristic of the first type. Decision problems can be viewed as rooted trees, with solutions lying on every leaf. Algorithms explore these trees, in the best case always choosing the right branch to travel down in order to arrive at the best solution. On the other side of the efficiency spectrum lies brute force, which simply checks every single leaf. Branch and Bound is an algorithm that is similar to brute force, in that there isn't an algorithmic selection of branches. However, it's a step up due to the fact that it reduces the number of solutions needed to check by 'trimming' the tree. In other words, whenever a branch is traveled down, the algorithm checks that all possible solutions the branch contains satisfy the conditions of the problems- if not, none of them are explored.

As for the second type of heuristic, reconsider the SAT. After first converting the problem into a more manageable form, such that there are only 3 variables in every "clause" (a sub-statement defined by use of parenthesis), state of the art SAT solvers use the help of heuristics to guide the assignment of values to variables, using rules thumbs such as:

1. if a variable is always true or always false, then assign that variable it's forced value.

2. prioritize unit clauses

Contrary to intuition, heuristics do not perform uniformly across NP-complete problems.

Theorem 2 (No Free Lunch) *For any algorithm, any elevated performance over one class of problems is offset by performance over another class. [5]*

Note that use of heuristics can be problematic. If the data used to create the rule of thumb isn't representative of all possible scenarios, the heuristic is overfit, and solutions generated have an unnecessary amount of noise.

Generally speaking, runtime is a function of input size, but now we'll take a look at the class of pseudopolynomial-time algorithms, which have runtimes that are polynomial in regards to input size, but are exponential in regards to other factors.

Consider the Set Partition problem (the zero sum problem), which asks how to partition a set of numbers into two subsets such that the sums of the

two subsets are equal. Algorithms using dynamic programming can solve the problem in time $O(nN)$ where n is the number of elements in the set and N is the maximum value of all elements in the set. note that N is bounded by $log_2 n$, so worst-case scenario this algorithm has exponential runtime.

5 Barriers to proof

In this section we'll explore one of main barriers of proving $P \stackrel{?}{=} NP$: the vagueness of general logical techniques.

Anyone exposed to Cantor's diagonalization argument has already been exposed to what we refer to as 'logical techniques', which are proofs that rely less on the nature or characteristics of the objects being examined and more on characteristics of sets of objects. To demonstrate such a proof techniques, we'll prove there exists a language that is not computable by all Turing machines. Let COMPENUM be the set of all languages decidable by some Turing machine, and ALL the set of all languages.

Theorem 3 (COMPENUM $\neq$ ALL) *There exists a language that is not decidable by any Turing machine.*

Let $\langle M \rangle$ be the binary encoding of a Turing machine M, and let language L be the set of binary encodings of Turing machines such that M does not accept $\langle M \rangle$.

We'll show by contradiction that L is not computable by any Turing machine, so we'll suppose the contradiction: that L is computable by a Turing machine A. By definition, $A(\langle M \rangle)$ accepts if and only if M is computable.

Consider $A(\langle A \rangle)$. In the case that $A(\langle A \rangle)$ computes and accepts, then by definition A does not decide on input $\langle A \rangle$, which is a contradiction. If $A(\langle A \rangle$ does not decide, then $\langle A \rangle$ is not in L, which implies that $\langle A \rangle$ accepts. Another contradiction.

Hence, L is not decidable by any Turing machine.

In the 1960's, mathematicians realized that by scaling down preexisting diagonalization-based proofs, one could prove distinctions between some complexity classes. These types of proofs are intuitively straightforward, but they fail due to the fact that they are too general. Suppose we have a black box, to which we can submit an iteration i of a problem P, and if i has a solution, the black box gives us the solution in time $O(1)$. We call this black box an **oracle**. The issue arises when we consider the fact that there exist oracles A and A' such that P^A, the set of all languages that are in P if we have access to A, is equal to NP^A (similarly, the set of languages that are in Pif we have access to A), and $P^{A'} \neq NP^{A'}$[**?**]. Simply put, there exist oracles such that either result of $P \stackrel{?}{=} NP$ is true. The diagonalization proof shown above, and most proof techniques shown above, are general enough that if two classes P and Q are shown to have some sort of relationship, that relationship holds with the

use of all Oracles. Hence, if we used complexity theory to show $P \stackrel{?}{=} NP$, then $P^A \stackrel{?}{=} NP^A$ for all oracles A, which is a contradiction.

6 $P \stackrel{?}{=} NP$

So why do we care about these sets of problems? If $P = NP$, then there exist polynomial time solutions to all problems in NP, including NP-complete problems. Modern cryptographic systems would become obsolete. Such a result would transform mathematics by allowing computers to create proofs of reasonable length, since recognizing a valid proof is in NP. Recognizing a valid proof would be reduced to creating a recognition algorithm. Similar could be said for other creative human endeavors. Great work would no longer require some creative genius, only the ability to recognize great work. Such a result would have huge implications for human identity.

Shor's algorithm relies heavily on the algebraic structures of numbers that we don't see in the known NP-complete problems. We know that his algorithm cannot be applied to generic "black-box" search problems so any algorithm would have to use some special structure of NP-complete problems that we don't know about. We have used some algebraic structure of NP-complete problems for interactive and zero-knowledge proofs but quantum algorithms would seem to require much more.

7 sources

References

[1] Aaron Cook. $P \stackrel{?}{=} NP$. http://www.scottaaronson.com/papers/pnp.pdf

[2] Stephen Cook. *THE P VERSUS NP PROBLEM* . http://www.claymath.org/sites/default/files/pvsnp.pdf

[3] Richard Kaynes. *Minesweeper is NP-complete!* Mathematical Intelligencer Spring 2000, volume 22 number 2, pages 9–15.

[4] Katherine Lai. *The Knapsack Problem and Fully Polynomial Time Approximation Schemes (FPTAS)* . 18.434: Seminar in Theoretical Computer Science Prof. M. X. Goemans March 10, 2006

[5] David H. Wolpert and William G. Macready. *No Free Lunch Theorems for Optimization*. IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 1, NO. 1, APRIL 1997

[6] wvbaileyWvbailey 20:31, 4 August 2006 (UTC), original drawing in Autosketch by wvbailey. https://en.wikipedia.org/wiki/File:Turing$_m achine_1$.JPG