SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

P vs NP

av

Aliénor Edström Bourdeau

2023 - K23

P vs NP

Aliénor Edström Bourdeau

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Per Alexandersson

2023

July 26, 2023

Abstract

Here you will find a —hopefully— easy-to-read introduction to the P vs NP problem. I explain the basics, Turing machines, complexity classes and terminology. Following up are some techniques and advancement that have been made in the field and the sketches of some illustrating proofs.

Thank you Per, Daria, Julie, Jakob and Aisata for the help.

Contents

1

Introduction

The elusive P vs NP problem was originally formulated by Stephen Cook in *The complexity of theorem proving procedures* [5] (1971). It has gained significant prominence and was selected as one of the seven Millennium Prize Problems in 2000 [13]. The question asks whether problems whose solutions can be quickly verified can also be quickly solved. *Quickly* means an algorithm that runs in polynomial time, where the time to solve the problem varies as a polynomial function based on the input size. Problems **solvable** in polynomial time belong to class P, while problems **verifiable** in polynomial time belong to class NP. The prevailing belief is that P is not equal to NP, suggesting that fast solutions for NP problems may not exist. A proof either way would have far-reaching implications in mathematics, cryptography, artificial intelligence, philosophy, economics, and many other fields.

In order to provide a thorough grasp of the P vs NP question, this paper will begin by introducing Turing machines, the fundamental cornerstone of computer science, along with complexity classes. This foundation will enable us to explore the question's connections to the 3-SAT satisfiability problem. Subsequently, various approaches to proving or disproving the P vs NP question will be presented. The primary objective of this paper is to showcase these approches, highlighting their strengths, weaknesses, and providing illustrative examples. Through this exploration, it becomes evident that creativity within the realm of mathematics knows no bounds, making the P vs NP question more accessible and comprehensible.

2

What is it?

2.1 Turing machines

We find the problem P vs NP on the border between mathematics and computer science, in the realm of algorithms. An algorithm is a set of rules that must be followed when solving a particular problem[8] and here we want to classify their speed.

Before we can run the algorithms and examine their speed let us define the machine that will run them: the Turing Machine.

In 1936, Allan Turing writes his paper *On computable numbers, with an application to the Entscheidungsproblem* [12]. With this paper Turing wants to answer the third question about mathematics that Hilbert posed in 1928, "is mathematics decidable?". Unknowingly to Turing, the question had already been answered by the American mathematician Alonzo Church a few months prior. However, in his very different proof, Turing not just answered the question but also laid the foundation for what would become computer science.

The three questions about mathematics posed by David Hilbert are 1− is mathematics complete, 2− is mathematics consistent and 3− is mathematics decidable. 1− A system is complete if all statements can be proved to be either true or false. 2− A system is consistent if no two theorems contradict each other. Gödel with his Incompletness Theorem had already answered 1 and 2, that any sufficiently complex system can't be both complete and consistent. The last question asks whether there exists a purely mechanical procedure to determine the truth or falsehood of any mathematical assertion exist. In other words, does an algorithm exist that could determine if any mathematical statement is true or false.

In his paper, Turing first comes up with the concept of computable numbers, a set including "every number that could be arrived at through arithmetical operations,

finding roots of equations, and using mathematical functions like sines and logarithms—every number that could possibly arise in computational mathematics"[10]. This set contains any number that can be calculated with arbitrary precision in a finite amount of time. Turing then imagines a "machine" that would work like a human mind and follow definite rules to calculate and write down the computable numbers. Alan Turing lives in a industrial world where every machine is built with a unique purpose in mind, but his will be able to do anything. The theoretical machine would have an infinite tape divided into squares and would be able to read and write symbols into them, moving one step left or right based on a set of rules. That way the machine would be able to perform any calculation expressed by a mechanical process, or as we call it now, an algorithm.

The machines have languages. A **language** is the set of all possible inputs or strings that the machine accepts. When a machine accepts an input string, it halts and enters an accepting state (or final state) indicating that the input string is recognized as a valid member of the language. When a machine rejects an input string, the machine halts and enters a rejecting state (also a final state). There are cases where the machine may enter an infinite loop, neither accepting or rejecting, just theoretically running forever.

In his first example, Turing prints the number 010101... but instead of printing a number let us see how a machine, whose language is all numbers starting and ending with a one, would run with input 101. The elements we need are:

- a finite set Q of control states
- a start state $q0 \in Q$
- the symbols part of the input alphabet (here $1, 0$)
- a tape alphabet T including a blank symbol (here B)
- a control unit to know where we are on the tape
- a transition function δ, $Q \times T \rightarrow Q \times T \times \{L, R\}$, stating which states, which symbol is written in the cell, if the control unit should move Left (L) or Right (R) and what the new state should be.
- a set of Final states (for example Accept/Reject)

We draw the transition function δ table:

When we run, the tape will look like this, with the current state underlined:

States	0	1	B
q0		(q1,1,R)	
q1	(q1,0,R)	(q1,1,R)	(q2,B,L)
q2		Accept	

Table 1: The table shows for each case the machine finds itself in which state it should go to, what symbol it should write, if it should go left, right or Accept. If the machine goes to a blank cell in our table it would stop and reject.

B	**1**	**0**	**1**	**B**

The control unit sees it has 1 in q0, therefor the TM prints 1, goes a step to the right and changes to state q1.

B	**1**	**0**	**1**	**B**

Now it is in q1 on a 0, it prints 0, goes one step to the right and goes to state q1.

B	**1**	**0**	**1**	**B**

1 in state q1, the TM prints 1, goes right and goes to q1.

B	**1**	**0**	**1**	**B**

We have stumbled on a blank in q1, the TM prints B, the control unit goes left and state changes to q2.

B	**1**	**0**	**1**	**B**

Seeing a 1 in q2 the TM accepts, this number is part of the language!

Here we only printed what was already on the tape but we could have chosen anything from the tape alphabet depending on the transition function. See that the universal machine, later called Turing machine (TM), could output 0s and 1s and run forever, getting closer and closer to any computable number we could desire (We have no memory constraint in our imaginary world). Each Turing Machine, TM, can output one number, so the size of the computable numbers is the cardinality of the TMs, which is also the number of finite binary strings. We know that there

are countably infinitely many binary strings, just the same as the number of natural numbers. We also know that the cardinality of real numbers is bigger than the cardinality of natural numbers, meaning:

$$|TMs| = |\mathbb{N}| < |\mathbb{R}|.$$

Hence, Turings machine revealed the existence of uncomputable numbers, demonstrating that certain numbers cannot be calculated through a finite algorithm or mechanical process. These uncomputable numbers serve as examples of unsolvable problems, indicating that no algorithm can be devised to determine their exact values within a finite time. This discovery was Turing's response to the Entscheidungsproblem. The proof was the original purpose of Turing's paper but happened to introduce the first computer, which has evolved a little since, and to run the first algorithm in form of the Turing machine.

The problems we are going to be interested in for P vs NP are decision problems, problems for which the answer is "Yes" or "No" like "Is it possible to satisfy this Boolean formula?". More precisely, we are interested in categorizing these problems according to the time their most effective solving algorithm takes to run on our machine. We could simply try to time the machine running the algorithm, but different machines have different power and hence different time for the same algorithm. Instead, we count the number of elementary operations used for an entry of N data, it is called the complexity of the algorithm.

2.2 Complexity classes

Consider an algorithm that operates on a dataset of size N and requires $O(N)$ steps to finish processing. This algorithm is said to run in linear time, making it highly efficient and rapid. Calculating the complexity of algorithms enables us to estimate how much the running time will grow depending on the growth of our entry of data N. It can seem trivial for small entries but as soon as N grows you will notice a huge gap even between a complexity of $O(N^2)$ and one of $O(N\log(N))$. To understand the scale, let us say we want to sort a list of numbers and have two algorithms, the first one is a naive algorithm with a complexity of $O(\frac{N^2}{2})$ and the second has a complexity of $O(N\log(N))$. Here is a table with the growth of the number of steps compared to the growth of N:

N	$\frac{N^2}{2}$	$N\log(N)$
10	50	30
100	5 000	600
1 000	500 000	10 000

Table 2: Size of input N against the number of steps.

$N\log(N)$ is clearly way more advantageous. Or as a visual representation in comparison with some more complexities:

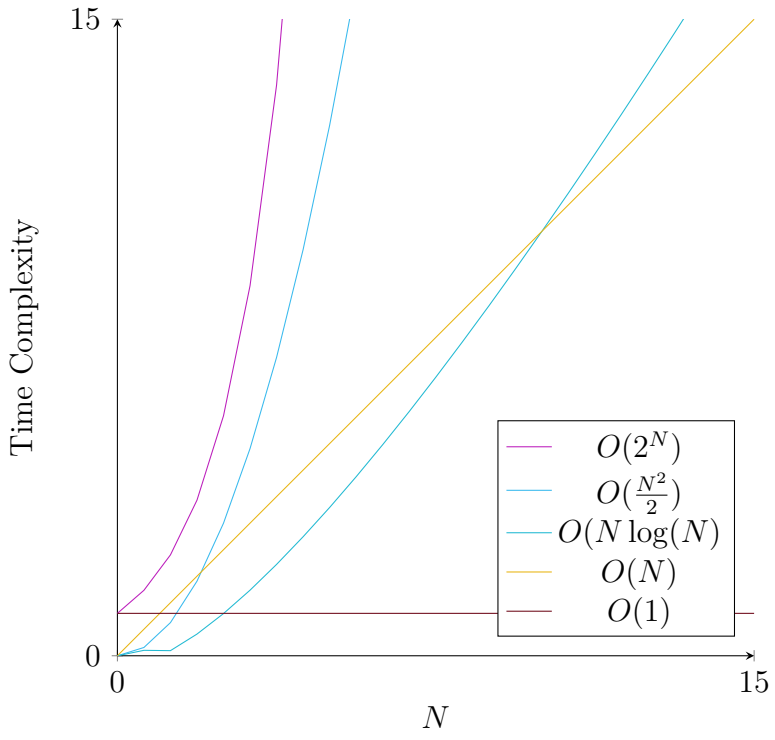


Figure 1: Visual representation of some time complexities.

The difference in growth is obvious. Now imagine we have applied any of these algorithms to sort the list. Verifying that the list truly is sorted is very easy and fast, we just need to go through it which takes N steps.

On this basis we can now build categories to sort our algorithms by their respective complexities, the complexity classes. The class P stands for polynomial and contains all the complexities like N, N^2, N^k. These algorithms are considered easy and fast. NP is a complexity class that represents the set of decision problems for which a given solution can be verified in polynomial time. This mean the solution

is easily verified, just like it is easy to go through the lines of a hard Sudoku when completed, and see if no two numbers are the same. Given a potential solution to the problem, we can check and verify whether the solution is correct or not in a time that is polynomial in the size of the input (its worst-case time complexity can be expressed as $O(N^k)$, where 'N' is the size of the input, and 'k' is a constant.). NP does not stand for non-polynomial but for nondeterministic polynomial.

Nondeterministic refers to a non-deterministic Turing machine (NTM) which has the ability to make non-deterministic choices during its computation. At each step, an NTM can have multiple possible transitions, leading to different states and symbols to write on the tape. It can explore multiple paths simultaneously, branching out into different computational branches. The machine cannot directly "guess" the correct solution to a problem; it explores all possible paths in parallel, halting and accepting an input if at least one of the computational branches leads to an accepting state. Non-deterministic Turing machines have greater computational power in terms of exploring multiple paths simultaneously than deterministic Turing machines but they are equivalent in terms of the languages they can accept or decide. This equivalence is known as the Church-Turing thesis [3], any problem that can be solved algorithmically can be solved by a Turing machine, regardless of whether it is deterministic or non-deterministic. However, deterministic Turing machines may need a higher time complexity, not being able to explore different paths at the same time.

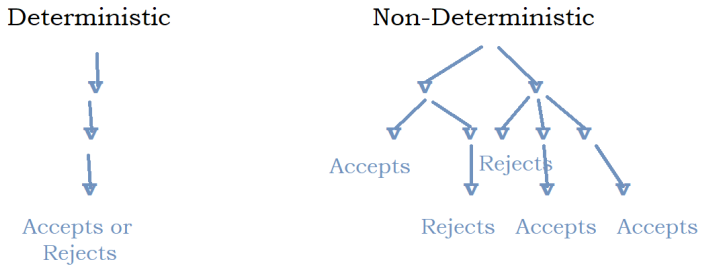


Figure 2: A comparison of a deterministic and a non-deterministic computation. The action prescribed by the NDTM rules in a situation could be *Write 1, go to state 2 and move right* ***or*** *write 0, go to state 4 and move right.*

According to our new standards, the sorting algorithms we had earlier are fast to run, so the problem would be in P if it was a decision problem. Sudoku solving algorithms (the symbols in the Sudoku puzzle are taken as the natural numbers 1

to N) are pretty complicated but once you have solved it you can easily and linearly see that it is indeed solved, so it is in NP. And some problems are not at all in these classes, Go and Chess (generalized chess has an $N \times N$ board with many pieces), for example, are complicated to solve and even when solved you cannot be sure that the chosen solution is the best one.

The complexity classes are not fixed, with people always finding new faster algorithms to solve the problems, they evolve. Problems previously considered NP became P (i.e. primality problem in 2002, finding out if a number is prime [2]) leading to the extension of the P class in the NP class. See Figure 3 for a graphic representation of the classes.

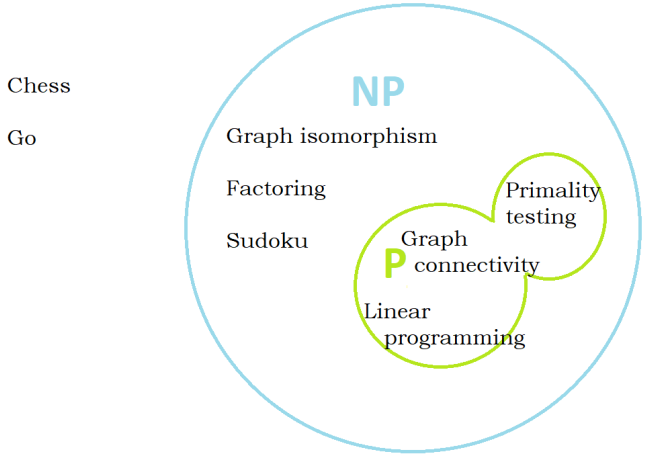


Figure 3: The complexity classes. Graph isomorphism is the usual computational problem of determining whether two finite graphs are isomorphic, graph connectivity determining if they are connected and linear programming is a method to achieve maximum profit or lowest cost in a mathematical model whose requirements are represented by linear relationships. For example *Given an integer $n \times d$ matrix A, an integer n-vector b, an integer d-vector c, is it possible to find a rational d-vector x such that $Ax \leqslant b$ and $c^t x$ is maximized.* Of course these are just some examples among many.

Now we can wonder, will P ever grow enough as to encompass NP? Is every easy-to-verify problem also easy-to-solve?

Or as a bridge to the challenge David Hilbert set to mathematicians in the 1900: "Does a purely mechanical procedure to determine the truth or falsehood of any

mathematical statement exist? Why or why not" [1]. Solving the P vs NP problem could lead to the easy factorisation of big numbers, hence the breaking of RSA codes and safety of most data on the internet or solving the protein folding problem, curing cancer. So how do we prove an NP problem cannot possibly be solved quickly, or that all NP problems are P?

3

A problem to govern them all

3.1 Proving P vs NP

Algorithms run on Turing Machines and are classified in complexity groups. The decision problems solvable with fast algorithm are in P, a growing sub-class of NP, but maybe equal to the NP class. NP problems seem to be more complex but are verifiable in polynomial time. Let us look at the two possibilities the P vs NP problem offers;

If we want to prove P is not NP we only need to prove that one problem in NP could not possibly become a P problem, like we knew we had to look at every element to be sure we really found the maximum of the list, so no algorithm could run faster than linear time. However, demonstrating the impossibility of proving something becomes considerably challenging when dealing with complex matters. Let us see how it can be done with the halting problem.

The halting problem is often attributed to Turing, to the very paper that gave us the universal machine [12]. But that is false, the first occurrence of The halting problem is from the fifties; Martin Davis inspired himself from Turing's paper and writes "[...] we wish to determine whether or not [a Turing machine] Z, if placed in a given initial state, will eventually halt. We call this problem the halting problem for Z. [...]" [7]. The problem explores the limits of what can be computed, it asks whether there exists a general algorithm that can determine, given any program and its input, whether that program will eventually halt.

To instantiate the problem we have two strings, P (which will work as the program) and I (The input). The TM Z is the algorithm and will keep on running if P halts on I, and halt if P goes into an infinite loop on I. Assume there exists a program Halt(P, I) that solves the halting problem and returns True if and only if P halts on I. Like a Russian doll Turing machine, we give the TM Z itself to

run on. Think about it, if P halts on I in first Z then P in the second Z gets the information and runs for ever, which is a contradiction. Now if P does not halt on I in the first Z and runs forever, then P will halt, which is again a contradiction. Hereby we proved by contradiction that there is no Halt program that can solve the halting problem. We could try to find something like that for P=NP.

The other option, to prove P = NP by showing, one after one, that all NP problems actually are P would be quite tedious. Luckily, we have a secret weapon, a problem to govern them all: SAT.

In the SAT (for satisfiability) problem we get true and false conditions and we want to see if we can satisfy them all. More formally, it involves determining whether there exists an assignment of truth values (TRUE or FALSE) to a given set of Boolean variables that satisfies a given Boolean formula. In SAT there are literals, either variables or the negation of variables, and clauses. A clause is a disjunction of literals, meaning it is formed by combining literals using the logical OR operation. A clause represents a condition that can be either satisfied (evaluates to TRUE) or unsatisfied (evaluates to FALSE) based on the truth values assigned to the literals. If we get a formula with literals and clauses like $(x_1\ OR\ x_2)\ AND\ (x_3\ OR\ x_2)$ the aim would be to find if it is a true formula or a false formula, satisfiable or unsatisfiable.

As an example, let us say some king wants to build a team of adventurers to send on a mission to save the world. He has a list of candidates, each with their own preferences and constraints. His goal is to find a combination that will make everyone happy and satisfy all the constraints. There are three paths representing the literals:

- Path 1: True if you choose a path through the mines, False otherwise.
- Path 2: True if you choose a path flying with eagles, False otherwise.
- Path 3: True if you choose a path across the mountains, False otherwise.

And the constrains (built with clauses) are:

1. Adventurer A prefers Path 1 or Path 3.
2. Adventurer B prefers Path 2.
3. Adventurer C cannot attend if Path 3 is chosen.

4. Either Path 1 or Path 2 must be chosen.

5. At least two paths must be chosen.

To solve this SAT problem the king needs to create logical clauses satisfying all constrains, for example:

1. Path 1 OR Path 3 - to satisfy adventurer A.

2. Path 2 - to satisfy adventurer B.

3. ...

We see here that the king can pick paths 1 and 2 and the problem will be satisfied. This particular problem is pretty simple, and in fact 1-SAT and 2-SAT problems with 1 or 2 literals in the constrains are classified P. But from 3-SAT onward, with up to 3 literals in each constrain, we have NP problems.

Note that the general SAT formula (not necessarily limited to exactly three variables per clause) can be transformed into an equivalent 3-SAT formula using a variable-duplication trick. In the variable-duplication trick we break down clauses with more than three literals into smaller 3-literal clauses while introducing new duplicated variables. If we have a clause with k literals (where $k > 3$) in the form $(X_1 \vee X_2 \vee X_3 \vee ... \vee X_k)$. We can create a set of new variables $Y_1, Y_2, ..., Y_{k-3}$ and replace the original clause with a series of 3-literal clauses:

$(X_1 \vee X_2 \vee Y_1)$, $(\neg Y_1 \vee X_3 \vee Y_2)$, $(\neg Y_2 \vee X_4 \vee Y_3)$, ..., $(\neg Y_{k-4} \vee X_{k-2} \vee X_{k-1})$, $(\neg X_{k-1} \vee \neg X_k \vee Y_{k-3})$, $(\neg Y_{k-3} \vee X_k)$.

Every new Y_k *OR NOT* Y_k will cancel out itself. By applying this transformation to each clause in the original SAT formula we create an equivalent 3-SAT formula. The number of clauses increases, and new variables are introduced, but the transformed formula remains satisfiable if and only if the original formula is satisfiable. There is a linear number of operations per clause so the variable-duplication trick is a polynomial-time transformation.

Now what is so special with 3-SAT and above? Cook and Levin have shown that any NP problem is reducible to SAT in polynomial time [5], for any problem in the complexity class NP there exists a polynomial-time algorithm that can convert instances of that problem into equivalent instances of the SAT problem. This grounds a new sub-class in NP: the NP-complete problems are all the NP problems you could reduce any NP problem to in polynomial time. This means that proving

any NP-complete problem is in P equals to prove all NP problems are in P, it is major. Our updated representation of the classes looks like this:

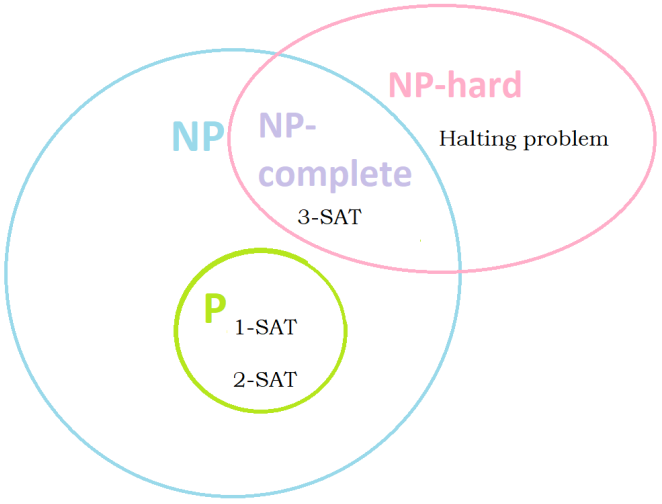


Figure 4: An NP-hard problem is a problem that is "at least as hard as the hardest problems in NP". The halting problem is NP-hard but not in NP because it is not computable.

Both alternatives seem now easy, but time has told us it is not that simple. We will see different approaches that have been tried, but first let's look at some NP-complete problems.

3.2 Some NP-complete problems

The problems that will be listed in this section are the most studied NP-complete problems and often serve as standard examples for teaching and understanding computational complexity and the intricacies of NP-completeness reductions. Let us start with *Vertex Cover* (VC).

Vertex Cover is a classic optimization problem and has practical applications in network design, resource allocation and scheduling. It is widely used in NP-completeness reductions due to its versatility. Given an undirected graph, the Vertex Cover problem asks whether there exists a set of vertices that includes at least one endpoint of every edge of the graph. The goal is to find the smallest vertex cover

possible. If there exists a vertex cover of size k (where k is a subset of the vertices), then the Vertex Cover problem is said to be satisfiable and the minimum value of k is sought to find the smallest possible vertex cover. Currently, the problem can be solved by an algorithm running in $2^k n^{O(1)}$

Next we have the *Traveling Salesman Problem* (TSP). TSP is an old optimization problem with applications in logistics, routing, and circuit design. It is an essential problem in combinatorial optimization and a classic example of an NP-hard and optimization problem. In the TSP, a salesman needs to visit a set of cities and return to the starting city, traveling the minimum distance possible. The bound for this problem is $O(n^2 2^n)$.

The *Knapsack Problem*: The Knapsack Problem is an optimization problem with applications in resource allocation, budgeting, and project scheduling. It serves as a representative of NP-hard problems in the area of integer programming. In the Knapsack Problem, we are given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The goal is to determine the maximum value of items that can be placed into the knapsack without exceeding its weight capacity. The *meet-in-the-middle* algorithm has a runtime of $O(2^{n/2})$, but for this algorithm the space complexity is also to consider. The more space we have the less time we need and vice versa.

Now the *Subset Sum*. Subset Sum is another classic optimization problem with applications in cryptography, data mining, and finance. It is often used in NP-completeness reductions to establish the hardness of other problems. Given a set of integers and a target value, the Subset Sum problem asks whether there exists a subset of the integers whose sum is equal to the target value. Here we have a usual upper bound of $O(n2^n)$. But, again with more space, a lower time complexity is reachable.

The *Hamiltonian Cycle* (HC) is a fundamental problem in graph theory and combinatorial optimization. It is often used in NP-completeness proofs due to its close relationship with the Traveling Salesman Problem. In the Hamiltonian Cycle problem, we are given an undirected graph, and the task is to find a cycle that visits every vertex exactly once and returns to the starting vertex. A dynamic programming algorithm can be used to solve the problem in time $O(n^2 2^n)$. A dynamic algorithm will break a complex problem into smaller overlapping subproblems and solve each subproblem only once. It stores the solutions to subproblems in a table or cache and reuses them to solve larger problems efficiently.

Next the *Clique Problem*. The Clique Problem is essential in graph theory and has applications in social networks, computer networks, and bioinformatics. It serves as a representative problem in graph-related NP-completeness reductions. Given an undirected graph, the Clique problem asks whether there exists a subset of vertices that are all mutually adjacent (form a clique) in the graph. The upper bound for this problem is $O(2^n)$, where n is the number of vertices in the input graph.

The *Integer Linear Programming* (ILP). We saw this problem in Figure 1, ILP is a powerful optimization technique used in various fields, including operations research and resource allocation. It plays a significant role in proving NP-completeness for problems related to integer programming. The ILP problem is a generalization of the Knapsack Problem. It involves optimizing a linear objective function subject to linear inequality constraints with integer variables. The upper bound for Integer Linear Programming is generally exponential, specifically $O(2^n)$, where n is the number of decision variables (integer variables).

And finally *Graph Coloring*. As its name hints, graph Coloring is important in graph theory, scheduling, and register allocation in compilers. It is often used in NP-completeness reductions for problems related to graph coloring and scheduling. Given an undirected graph, the Graph Coloring problem asks whether it is possible to color the vertices with a specified number of colors such that no two adjacent vertices have the same color. For an undirected graph with n vertices, the upper bound for the worst-case running time algorithm is $O(k^n)$, where k is the maximum degree of the graph. This is because there are k choices for the color of each vertex, and there are n vertices in the graph, giving k^n possible colorings to explore.

With a better understanding of the problems, let's now explore the progress made in the quest to prove or disprove P vs. NP.

4

Different approaches

Now we will explore the ideas that are concrete enough to have worked for advancement in our problem. The drawback is that they are also concrete enough that we understand why they can't work for P≠NP! The ideas I will cover are the ones Scott Aaronson chose as most relevant in his paper P=?NP [1].

4.1 Logical techniques

Logical techniques attempt to resolve the P vs NP question using formal logic principles. This can involve first-order logic, propositional logic, or similar logical systems. This kind of reasoning is just what we used with the logical clauses in our kings problem from Section 3.1. However, due to the relativization barrier, these techniques might fail to capture the whole problem, because some conditions might depend not just on the individual paths the adventrer could choose, but also on their interactions.

Strengths: They provide a rigorous, well-founded framework for understanding and reasoning about computational problems.

Weaknesses: Their main weakness is the *relativization barrier*, which is essentially a limit on the types of proofs that these techniques can achieve.

Logical techniques is the approach Turing used for the diagonalization proof of the Entscheidungsproblem and that Cook used to prove the Cook-Levin theorem [6], to come up with the SAT problem we saw earlier and the NP-complete class.

Let us take a look at the Cook-Levin theorem, showing that any NP problem can be reduced to 3-SAT and thus that 3-SAT is NP-complete.

As known from earlier, SAT is a boolean satisfyability problem. 3-SAT is a version of SAT with 3 variables taking the form $(X_1 \vee X_2 \vee X_3) \wedge (...) \wedge$ and so on. For a problem to be NP-complete we remember it should be in NP and NP-hard

(see Figure 4), let us start with checking the NP condition; To solve the 3-SAT problem we need in the worst case to check all possible variables before the formula is satisfied, maybe see if $(1 \wedge 1 \wedge 1)$ works, then $(1 \wedge 1 \wedge 0)$, and so on... So there are $8 = 2^3$ possible assignments, giving us a time of 2^n. Verifying it is easy, we can read through the solution and see if it works, so SAT is solvable in NP time and verifiable in P time, therefore it is in NP.

Now remains to show that SAT is NPhard, that every language in NP is reducible to SAT in polynomial time, or $L \leqslant_P SAT$, $L \in NP$. We talked about languages with the Turing machines in the beginning; in computational complexity, languages are used to represent decision problems in a formal way. If we consider the decision problem *Is a given number n prime?* The language representing this problem consists of all strings for which the answer is *yes*. There are many languages in NP, so that will take a very long time. Good thing we have an ace up our sleeve, lets work some magic! Remember every problem could be represented by a non-deterministic Turing machine? Reducing the NDTM into SAT in polynomial time will do the trick.

To demonstrate that SAT is NP-hard, we will perform a polynomial-time reduction from a non-deterministic Turing machine to SAT. All along the proof we must ensure that the reduction maintains polynomial complexity: in terms of the number of variables and clauses, the size of the SAT formula should not grow exponentially with respect to the size of the original NDTM instance. To establish a polynomial upper bound on the number of variables in the SAT formula, we consider the known upper bound on the running time of the NDTM, denoted as $p(n)$ for an input size of n. As the NDTM's execution is bounded to $p(n)$ steps, we can define the maximum number of time steps in the encoded SAT formula (each time step corresponds to a specific configuration of the NDTM, represented by sets of Boolean variables). There are $p(n)$ elements that can take $p(n)$ different values each so we have $O(p(n)^2)$ Boolean variables. We can encode them in a space of $O(log(p(n))$ because $log(p(n))$ bits are enough to represent each of the $p(n)$ possible values. And the number of clauses is $O(p(n)^3)$ because each clause typically involves three different Boolean variables, and there are $p(n)$ possibilities for each of these three variables, resulting in $p(n) \times p(n) \times p(n) = p(n)^3$ possible combinations of clauses. Therefore, the total Boolean expression will be at most $O(log(p(n))p(n)^3)$. Thus, the SAT formula can be constructed and manipulated within polynomial time, ensuring the polynomial-time complexity required by the Cook-Levin theorem.

Now we convert the NDTM into SAT formulas, here are the variables for all $-p(n) \leq j \leq p(n)$ and $0 \leq t \leq p(n)$:

- Q_q^t is True if we the current state is (q) at time t.
- H_{ij}^t is True if the tape is (i) and the cell is (j) at time t (NDTM can have several tapes).
- T_s^t is True if the symbol is (s) at time t.

Now the constraints:

1. There can only be a single state at anytime, $\neg Q^t q \vee \neg Q^t q'$
2. There can only be a single cell at the time, $\neg H^t i' j' \vee \neg H^t ij$
3. There can only be a single symbol at anytime, $\neg T^t s \vee \neg T^t s'$

Now the transition function. Say the State is 0, the tape 1, the Cell 4, the symbol B and the machine should move to state 5. Then our transition function would look like:

$$Q_0^t \wedge H_{14}^t \wedge T_B^t \implies Q_5^{t+1}.$$

And now we have a SAT problem! The formula would be $(\neg Q_0^t \vee \neg H_{14}^t \vee \neg T_B^t \vee Q_5^{t+1}) \wedge$ the next clause $\wedge ...$ ($A \implies B$ can be written as $\neg A \vee B$). We only want 3 variables per clause for 3-SAT, using the variable duplication trick from Section 3.1 we split them in two and add a variable (V) that cancels out itself : $(\neg Q_0^t \vee \neg H_{14}^t \vee \neg T_{14B}^t) \wedge (Q_5^{t+1} \vee V \vee \neg V)$.

The reduction ensures polynomial complexity, the size of the resulting 3-SAT formula will grow at most polynomially with respect to the size of the original NDTM instance (in computational complexity, the size of a problem instance refers to the length or number of bits needed to represent that instance).

Polynomial-time reductions is important because it guaranties that the size of the transformed formula won't explode exponentially, making the reduction computationally impossible. When the reduction is polynomial, it also means that it can handle instances of the original NDTM problem in a complete manner, covering all possible cases without missing any relevant information. By maintaining polynomial complexity, we can establish a clear relationship between the complexities of the original NDTM problem and the 3-SAT problem. Since polynomial-time reductions preserve complexity classes, showing that the NDTM problem reduces to

3-SAT in polynomial time allows us to conclude that 3-SAT is at least as hard as the original problem.

To achieve this in our proof, we systematically encode the NDTM's information into a Boolean formula representing the SAT problem. We define variables to represent the state, tape contents, and symbols of the NDTM at each time step. The size of the SAT formula grows proportionally to the number of variables and clauses needed to encode the NDTM's behavior. During the reduction, we establish constraints that capture the NDTM's rules and transitions, ensuring that the SAT formula accurately enforces the NDTM's behavior. Each constraint can be added to the formula in a polynomial number of steps. When we maintain the SAT formula in 3-SAT form, we ensure that the reduction is still computationally tractable.

Now we have shown that the Cook-Levin Theorem establishes 3-SAT as NP-hard, as any problem in NP can be reduced to 3-SAT in polynomial time, demonstrating the NP-completness of 3-SAT.

Nevertheless, with the Baker-Gill-Solovay theorem, the relativization barrier arose which showed that there exist "relativized worlds" where P=NP and worlds where P≠NP, indicating that certain logical techniques cannot prove or disprove P=NP. The relativized P vs NP question asks whether P = NP holds in the presence of an oracle, a hypothetical black box that can provide answers to specific computational problems. If we have for example a standard Turing machine that is trying to determine whether a given undirected graph is connected, it would need to explore paths through the graph and check if every vertex is reachable from every other vertex. But if we have an oracle for the graph connectivity problem, when queried with a graph, it can immediately provide the answer whether the graph is connected or not. With the help of this oracle we can construct an oracle Turing machine which makes queries to the graph connectivity oracle and obtain immediate answers about the connectivity of any given graph. The oracle Turing machine could start by querying the oracle to determine if there is a path between two specific vertices and then proceed to explore other parts of the graph based on the oracle's responses, effectively leveraging the oracle's knowledge to solve the problem more quickly. So the relativization barrier implies that any proof technique that does not account for oracles cannot definitively resolve the P vs NP question, we need to look at other techniques.

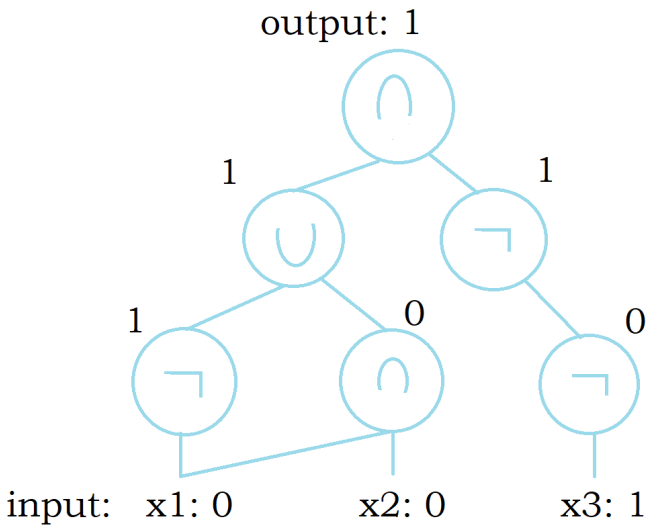
4.2 Combinatorial techniques

Combinatorial techniques involve using combinatorial principles and properties to try and separate P from NP. Suppose you're given a large number of tiles, each with a different colour and shape. The P=NP question in this case would be like trying to determine whether there's a fast method to arrange these tiles to form a specific pattern. Combinatorial techniques would look for a 'combinatorial property' (a feature related to arrangements, combinations and permutations) that separates the 'easy' tile patterns (ones that can be created quickly) from the 'hard' ones (those that take a long time to create).

Strengths: They provide a useful way of understanding the "structure" of problems.

Weaknesses: They face the "natural proofs barrier," which suggests it is hard to distinguish between polynomial time computable functions and non-polynomial time functions using a "natural" combinatorial property.

Combinatorial techniques have among other things been employed to derive lower bounds on the size of Boolean circuits. A circuit refers to a directed acyclic graph consisting of gates and wires. The circuit needs input gates with an input variable, logical gates which perform Boolean operations AND ($\wedge$), OR ($\vee$) or NOT ($\neg$) and an output gate with the final output of the circuit. See an example:



One proof for circuit lower bounds was written by the same man who found the

natural proofs barrier, Razborov. In his paper *Lower bounds for the size of circuits of functions computing inner product*[11], he analyzes the Fourier coefficients of the Inner Product function to show that any small circuit computing the function has limited influence on its Fourier spectrum. Fourier analysis breaks down a function into a combination of simpler sinusoidal functions, the Fourier coefficients provide information about the amplitudes and phases of these sinusoidal components in the function. The Fourier spectrum is the collection of the Fourier coefficients, analyzing it helps understand the function's properties and how it can be approximated or decomposed into simpler components. The Inner Product function takes two binary strings of the same length and performs bitwise operations (AND and XOR) on each corresponding pair of bits. It calculates the sum of the bitwise products, resulting in a single output bit. The function determines if the number of matching bits in the input strings is odd or even and outputs 1 or 0.

Using the analysis, Razborov constructs a low-degree polynomial that approximates the Inner Product function on a large set of inputs. Then, he applies the Discrepancy Method to demonstrate that the polynomial cannot accurately approximate the function for a significant portion of the input space. The Discrepency Method is a combinatorial argument, a study of the differences between the expected behavior of a function and its actual behavior on different input subsets. Razborov proves that small circuits computing the Inner Product function cannot accurately capture its behavior. Thus he establishes a lower bound, showing that circuits larger than linear size are necessary to compute the Inner Product function, an interesting point for our complexity class-sorting. Of course lower bounds are good for $NP \neq P$ in case the lower bound is strictly superior to P.

But as mentioned, Combinatorial techniques fail to overcome Razborov's natural proof barrier, they do not satisfy all three criterias it demands: polynomials, compositionality, and low error. Polynomials because the techniques often deal with combinatorial objects, counting arguments, or structural properties that are not naturally expressed in terms of polynomials. Polynomials ensures computational efficiency and tractability, enabling the analysis of complexity separations in a rigorous and meaningful way. Compositionality because Combinatorial techniques often rely on intricate combinatorial arguments that may heavily rely on specific properties or structures of the problem at hand, making it challenging to break down the proof into smaller, modular components that can be systematically combined. Composi-

tionality is important to generalize the technique, to prove separations for a wide range of problems. And finally low Error: Combinatorial techniques might involve complex and intricate reasoning, which can introduce a higher probability of error. Due to the complexity of combinatorial arguments, it becomes more challenging to ensure a low error rate. So for a final resolution of our P vs NP problem we will need to look at another method.

4.3 "Hybrid" techniques (logic plus arithmetization)

These methods blend logical techniques with arithmetization (essentially, a way of encoding Boolean logic using polynomials, which is good because the lack of polynomials was what caused the barrier with our combinatorial techniques). If you have a large Sudoku puzzle, the logical techniques would be the rules of Sudoku, while the arithmetization would be a process of translating the puzzle into a set of polynomial equations. The hope is that by working in this 'arithmetic' realm, we can circumvent some of the limitations of pure logical reasoning.

Strengths: By combining logic and arithmetic, these techniques have been able to bypass the relativization barrier, allowing for novel types of proofs.

Weaknesses: They encounter of a new barrier, the algebrization barrier. It is a generalization of the relativization barrier, which indicates that certain arguments that treat algebraic extensions of computation similarly can't resolve P vs NP.

To see how this technique can look like, let us implement the Sudoku puzzle. We can represent the puzzle as a set of polynomial equations using algebraic variables and constraints:

First, we assign variables to represent the values in each cell of the Sudoku grid. We define 81 variables, each representing a cell, denoted as x[i][j], where i and j range from 1 to 9.

Then the domain Constraints. Each cell can contain a number from 1 to 9, we can express this constraint mathematically as:

$$x[i][j] \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

for all i, j in the range 1 to 9.

Row Constraints: we want to ensure that each row contains unique values. For

each row r and each number n from 1 to 9, add a constraint:

$$\Sigma(x[r][j] = n) = 1$$

for all j in the range 1 to 9.

Of course we do the same for the columns and then again for the boxes. Then we incorporate the initial clues for the Sudoku puzzle as additional constraints, for example if we had a 4 in cell $(1, 1)$ we would write

$$x[1][1][4] = 1.$$

After having turned the Sudoku into polynomial equations we only need to solve the system. The "Hybrid" technique has mainly been used by Razborov, again, for circuit lower bounds.

But again these method fail, to the relativization barrier: The arithmetization process involves translating a computational problem into a set of polynomial equations, which can fail to fully capture the complexity of the problem under consideration. The techniques may also struggle to handle more general computational models that require non-algebraic reasoning or non-constructive arguments. Understanding and proving complexity conjectures often require deep insights into the structure of computations, not easily captured by algebraic techniques alone. And finally, complexity questions such as P vs. NP often involve non-constructive arguments or proofs by contradiction. These types of arguments mean reasoning about the existence or non-existence of certain computational structures or properties, maybe going beyond what can be expressed through algebraic or arithmetization methods. So here we go, to the next technique!

4.4 Ironic complexity theory

Ironic complexity theory involves using algorithms to again prove lower bounds, but bypasses the natural proofs, relativization, and algebrization barriers. If you're trying to find the shortest route that visits a certain set of cities, Ironic complexity theory would be like using an algorithm designed to solve another type of problem (like sorting a list of numbers) to make a statement about the minimum amount of time it takes to travel between those cities.

Strengths: This approach has led to some breakthroughs, such as Ryan Williams' result that NEXP is not contained in ACC^0 [14], a certain class of circuit models. It's a very creative approach that has led to surprising insights.

Weaknesses: The approach is somewhat counter-intuitive and its results have not directly approached the P vs NP question.

Let us look closer at Ryan Williams' theorem, it states "NEXP $\not\subset$ ACC (and indeed NTIME(2^n) $\not\subset$ ACC), where ACC is the union of $AC^0[m]$ over all constants m".

NEXP stands for Nondeterministic Exponential Time. It is the class of decision problems that can be solved by a nondeterministic Turing machine in exponential time ($2^{n^{O(1)}}$). In terms of complexity classes the hierarchy is as follow: $P \subseteq NP \subseteq EXPTIME \subseteq NEXPTIME$.

ACC stands for Arithmetic Circuit Class. It represents a class of circuits, like the ones we talked about in Section 3.2. $AC^0 \subseteq ACC^0 \subseteq ACC$. AC^0 consists of all families of circuits of depth O(1) and polynomial size, with unlimited-fanin (the number of input a logical gate can handle), AND gates and OR gates (NOT gates are only allowed at the inputs). The m adds an extra type of gate which calculates the modulo operation for some modulus m.

The proof of this theorem basically shows that there are problems that a nondeterministic Turing machine can solve in exponential time, but that these solutions cannot be efficiently computed by ACC circuits. The key idea is using diagonalization (like in Turing in Section 1.1). We assume that we have a hypothetical algorithm that can efficiently solve all problems in NEXP using ACC circuits, then we show that this assumption leads to a contradiction.

Using diagonalization, Williams constructs a specific problem that is not efficiently solvable by ACC circuits, he does this by constructing a "hard" instance of the problem for which any ACC circuit solving it would need to be of a large size (exponential in the input length). Next, he uses the assumption that there is an efficient algorithm using ACC circuits to solve all NEXP problems and applies this algorithm to the constructed hard instance. However, since the hard instance requires large ACC circuits to be solved efficiently, a contradiction is reached. By reaching this contradiction, Williams conclude that our initial assumption (the existence of an efficient algorithm using ACC circuits to solve all NEXP problems) must be false. Therefore, NEXP is not contained in ACC.

The "ironic complexity" part of the proof is because a faster-than-brute-force algorithm for a problem called ACCSat is used to proove $NTIME(2^n) \not\subset ACC$. Aaronson comments the whole proof in the previously cited paper [1].

This proof is interesting because barriers are overcome and nontrivial work is achieved using a simple diagonalization, barely different from what Turing did in 1936 (of course after some complicated work). I have no definitive argument here against this method being able to solve P vs NP; if you find it inspiring, you are free to explore its potential further.

4.5 Arithmetic circuit lower bounds

Scott Aaronson introduces this technique as a kind of computation that operates directly on elements of a field, such as the reals or complex numbers [1]. Arithmetic circuits have no limit of finite precision and can manipulate elements of a field but cannot extract individual bits (in contrast to digital circuits that we have seen earlier, such as logic gates or bitwise operations, which allow the manipulation of individual bits within a binary representation of numbers).

Strengths: This approach provides a different way of looking at the problem, focusing on numerical computation rather than Boolean logic.

Weaknesses: The possible susceptibility to arithmetic variants of the natural proofs barrier. Plus, it's not clear how results about arithmetic circuits translate to results about the Boolean circuits more commonly used in computational theory.

An interesting conjecture born from arithmetic circuit is the Valient conjecture. Named after Leslie Valiant, it states that any arithmetic circuit computing the permanent of an $n \times n$ matrix requires superpolynomial (more than polynomial) size in n, over any field of characteristic other than 2. In other words, if Valiant's Conjecture is true, it means that there is no efficient way to compute the permanent using a small arithmetic circuit.

The permanent (Per) and determinant (Det) are functions defined on an $n \times n$ matrix X. The permanent of X, denoted as Per(X), is the sum of the products of all possible n elements of X, where each element is chosen from a distinct row and column. The determinant of X, denoted as Det(X), is the sum of the products of n elements of X, where one element is chosen from each row and column, with signs alternating based on permutations. The determinant can be computed in polynomial

time using techniques like Gaussian elimination, but the permanent is known to be #P-complete.

#P (sharp P) is closely related to the class NP, but while NP focuses on decision problems, #P is about counting the number of solutions to a given problem. Formally, a function f is in #P if there exists a nondeterministic polynomial-time Turing machine M that can guess a solution y input x, and in polynomial time, verify whether y is a valid solution to the problem encoded in x. The machine M should then output the number of valid solutions for each input x. For example consider the problem #SAT, which is related to the SAT problem. While SAT asks whether a Boolean formula has at least one satisfying assignment, #SAT asks for *the number* of satisfying assignments. Determining the number of solutions to #SAT is a #P-complete problem, meaning that any problem in #P can be polynomially reduced to #SAT.

A polynomial-time algorithm for the permanent would imply more than P = NP; it would enable counting the number of solutions to NP-complete problems efficiently. Today a lot of research is aimed at proving the Valiant conjecture to advance in P vs NP.

Now there are some open questions and conjectures related to arithmetic circuit lower bounds and the permanent vs. determinant problem.

The main open question is whether there are pseudorandom polynomial families that cannot be efficiently distinguished from random polynomials by any algorithm. Constructing such families would establish the natural proofs barrier in arithmetic complexity theory and disqualify this kind of technique for our quest. Aaronson presents a conjecture regarding a pseudorandom family of polynomials that are homogeneous of degree $d = n^{O(1)}$. The conjecture suggests that distinguishing these polynomials from random polynomials of the same degree would require exponential time, an evidence for a natural proofs barrier.

Another open problem that remains is to prove a lower bound for the permanent that surpasses the corresponding lower bound known for the determinant. Currently, all known lower bounds for arithmetic circuits fail to differentiate the permanent from the determinant.

4.6 Mulmuley and Sohoni's Geometric Complexity Theory (GCT)

GCT is a program that tries to tackle the P vs NP problem by translating it into a problem in algebraic geometry and representation theory. If P=NP were like trying to solve a giant Rubik's cube, then GCT would be like translating the problem into a different domain, like the realm of shapes and symmetries. You'd study the properties of these shapes and symmetries in hopes of gleaning insights that you can translate back to solve the original Rubik's cube problem. If we were to use GCT for the previous kings problem, we might interpret adventurer and its preferences as a point in a geometric space, and the problem of finding the best route becomes a problem of understanding the geometry of this space.

Strengths: GCT uses the rich mathematical theory of algebraic geometry and representation theory, potentially allowing for powerful new techniques to be applied to the P vs NP problem.

Weaknesses: The approach is very complex and highly mathematical, which makes it difficult for many researchers to approach. GCT's approach is also quite far removed from the "natural" formulation of the P vs NP problem, which makes it harder to connect results in GCT back to traditional complexity theory.

GCT is an ambitious program started by Ketan Mulmuley in the late 1990s, with the goal of proving P $\neq$ NP. It uses algebraic geometry and representation theory to study the relationship between complexity theory and algebraic geometry. Algebraic geometry looks at how equations describe geometric shapes and vice versa. Instead of focusing solely on numerical solutions to equations, it considers solutions in terms of geometric objects like points, curves, surfaces, and higher-dimensional shapes. For example, a straight line in a plane can be described by an equation like $y = mx + b$, where m represents the slope and b represents the y-intercept. The solutions to a system of two equations in two variables can form a curve in the plane and the solutions to a system of three equations in three variables can form a surface in three-dimensional space.

GCT focuses on proving Valiant's Conjecture we saw in the previous section; "any affine embedding of the $n \times n$ permanent into the $m \times m$ determinant requires $m = 2^{n^{\Omega(1)}}$" (Ω = lower bound), implying that the permanent requires exponential-size arithmetic circuits. In other words, GCT wants to establish lower bounds on the size

of arithmetic circuits computing certain symmetric functions, like the permanent and determinant. Geometrically, the permanent and determinant are both symmetric functions and can be characterized by their symmetries among all homogeneous polynomials of the same degree. We know they are symetric because the permanent of an $n \times n$ matrix A is a sum of products of entries of A, where each term in the sum corresponds to a permutation of the rows and columns of A. The determinant, on the other hand, is a sum of products of entries of A, where each term corresponds to a permutation of the columns of A with an alternating sign. GCT translates the problem of embedding the permanent into the determinant in an algebraic-geometry conjecture about orbit closures. An orbit closure is the set of points obtained by applying all possible symmetries to a given object. In this case, the object of interest is a certain type of polynomial.

A bit of definitions: in a given representation, the multiplicity of an irreducible representation refers to the number of times it appears as a direct summand (an object that obtained through the process of taking a direct sum). An irreducible representation is a specific type of representation that cannot be further decomposed into smaller, nontrivial sub-representations. Multiplicity obstructions is attempting to embed one representation into another. Specifically, GCT seeks to find explicit obstructions or barriers to embedding the representation of the permanent into the representation of the determinant. It tells us how many copies of the irreducible representation exist within the larger representation.

GCT seeks to find irreducible representations that act as multiplicity obstructions, i.e., representations where the multiplicity of the permanent is greater than the multiplicity of the determinant. Finding multiplicity obstructions would prove that the determinant lacks the necessary symmetries to embed the permanent and provide evidence for the hardness of the permanent. Efficient algorithms for computing multiplicities are essential for progress in GCT, and research has focused on finding positive formulas and representations for it.

So the approach involves defining an NP function called E and a P-complete function called H, which are characterized by symmetries similar to the permanent and determinant. The idea is to find explicit representation-theoretic obstructions associated with the orbit of E but not with the orbit of H. The E function is defined using matrices over a finite field. It tests whether a certain determinant is zero, which is an NP problem. E itself being NP-complete is not necessary to prove P $\neq$ NP as it is sufficient to put any NP problem outside of P.

Mulmuley and Sohoni's Geometric Complexity Theory is interesting because it joins all the previous techniques into a beautiful cooperation, we have the circuits, we have the hybrid techniques, the ironic complexity and so on. That way all the barriers are jumped and the possibilities wide. But it also adds complexity, some people believe it is necessary for a complex problem. Recently there has been some result in CGT with the paper *Positivity of the symmetric group characters is PH-hard* [4] from C.Ikenmeyer, I.Pak, and G.Panova. Among other results, the study of positivity of symmetric group characters leads to the compelling outcome that, if we can decide positivity in polynomial time, it implies P = NP. The positivity of symmetric group characters refers to the study of whether the value of a symmetric group character is non-negative. A character of the symmetric group is a function that associates each permutation of the symmetric group with a complex number.

The current suggestion of GCT for progress is to look for faster obstruction-finding algorithms, it certainly is not the only way but food for thoughts.

5

Conclusion

The P vs NP problem is an interesting question on the verge of many sciences that could lead to many advancements in various fields if solved. At its core, the problem investigates the relationship between two complexity classes: P (polynomial time) and NP (nondeterministic polynomial time). It seeks to determine whether these classes are equivalent, meaning problems that can be efficiently solved (P) are also efficiently verifiable (NP).

Mathematicians have pursued diverse and creative approaches in tackling this problem. Despite notable progress, the question remains elusive, and a definitive solution is yet to be found. One might be tempted to venture into this pursuit, enticed by the recognition and rewards associated with the Millennium Prize Problems. Moreover, the potential implications extend far beyond academic curiosity, as the resolution could have profound consequences for the field of online security and cryptography. But keep in mind, even if P were to equal NP, it would not automatically guarantee super-fast algorithms for all problems, the theoretical classification of a problem as P does not necessarily translate into practical usability. For instance, algorithms with time complexities such as $N^{10,000}$ fall under P but are impractical in real-world scenarios. And considering the prevailing sentiment among mathematicians, it is widely believed that P and NP are distinct complexity classes: W. Gasarch has been making surveys among mathematicians every few years. In 2002 61% of the asked mathematicians thought there was no equality, in 2012 it grows to 83% and 88% in 2019 [9].

One of them said, people who still believe P=NP are like the people who believe Elvis is still alive. But even if the question remains unanswered, it is still meaningfull. As Scott Aaronson puts it: "modern cryptography, quantum computing, and parts of machine learning could all be seen as flowers that bloomed in the garden of P ?= NP".

References

[1] Scott Aaronson. P=?NP. *Electron. Colloquium Comput. Complex.*, TR17-004, 2017.

[2] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[3] Alonzo Church. An unsolvable problem of elementary number theory, 1936. Unpublished manuscript.

[4] I.Pak C.Ikenmeyer and G.Panova. Positivity of the symmetric group characters is PH-hard, 2022.

[5] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium*, pages 151–158, New York, 1971. ACM.

[6] Stephen A. Cook. *The Complexity of Theorem-Proving Procedures*. Association for Computing Machinery (ACM), 1971.

[7] Martin D. Davis. *Computability and Unsolvability*. McGraw-Hill Series in Information Processing and Computers. McGraw-Hill, 1958.

[8] Oxford Advanced Learner's Dictionary. Definition of algorithm noun, 2023.

[9] William I. Gasarch. Guest column: The third P=?NP poll. *SIGACT News*, 50(1):38–59, 2019.

[10] Andrew Hodges. *Alan Turing, Enigma*. 1983.

[11] Alexander A Razborov. Lower bounds for the size of circuits of functions computing inner product. *Journal of Computer and System Sciences*, 33(2):223–228, 1987.

[12] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[13] Wikipedia contributors. Millennium prize problems — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Millennium_Prize_Problems&oldid=1163231816, 2023. [Online; accessed 18-July-2023].

[14] Ryan Williams. Non-uniform acc circuit lower bounds. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, IBM Almaden Research Center, November 2010. IEEE.