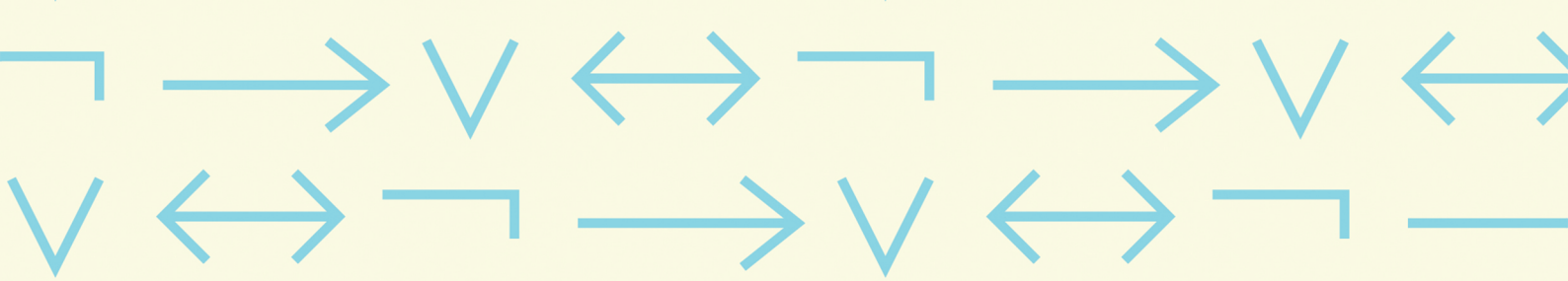


THE SAT STORY

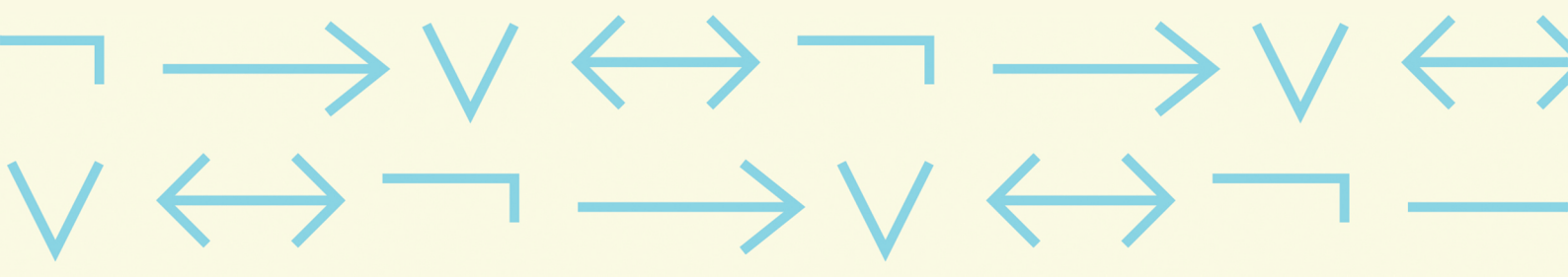
FROM ARISTOTLE TO MODERN SAT SOLVERS

Mentor:
Prof.
Zeynep Kiziltan

Author:
Francesco Giordani

ELICSIR · Scuola Ortogonale

Academic Year 2024/2025



Contents

List of Figures

1 Introduction to SATisfiability

Data-driven AI is one of the hottest topics in computer science today. This form of AI relies heavily on neural networks, which often act as black boxes that provide results without clear explanations. These missing explanations become problematic in many real-world situations, where solutions must be justified step by step. Such situations frequently involve highly complex problems, many of which belong to the class of NP-complete problems. NP-complete problems are notoriously hard to solve, and among them the satisfiability problem (SAT) is especially important, as it was the first proven to be NP-complete. Despite this theoretical hardness, modern SAT solvers are remarkably effective and can handle large and challenging instances in practice.

The aim of this article is to explain and motivate why SAT is such a fundamental problem in computer science, to explore its origins, and to discuss how researchers and practitioners approach it today. This first section presents the problem, offering a clear definition, examples, and an overview of possible applications. Since this problem is based on the concept of satisfiability that comes from logic, the second section is dedicated to the history of logic, from Aristotle to the present day. The third section focuses on the complexity of SAT, explaining why, being the first NP-complete problem, it has been of great importance. Then, in the fourth section the article discusses on SAT-solvers, the practical approach used to solve the problem. Finally, in the fifth section, a full example is provided to demonstrate how SAT can be employed to solve other computational problems, supported by a practical implementation.

1.1 Definition of SATisfiability

Definition 1 (SAT Problem)**.** *The* Satisfiability *problem (SAT)*[8] [3] *is the problem of determining whether there exists an assignment to the variables of a given propositional formula such that the formula evaluates to true; in this case, the formula is said to be satisfiable.*

Unsatisfiable Example The following is an example of a large unsatisfiable propositional formula [7] in conjunctive normal form (CNF). While CNF is not a strict requirement for expressing propositional formulas, it is commonly used due to its simplicity and suitability for some SAT solvers.

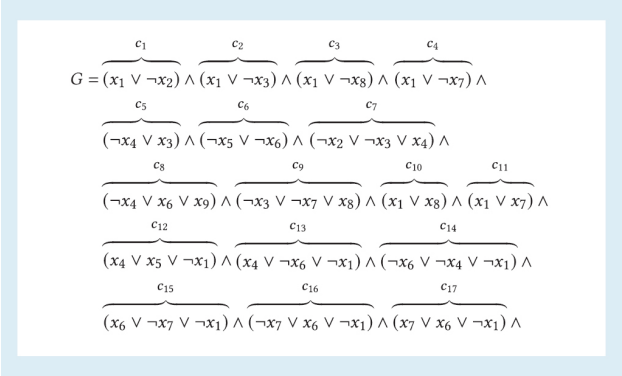


Figure 1: An unsatisfiable propositional formula [7]

At first glance, it may not be obvious that this formula is unsatisfiable. A naive approach to verify unsatisfiability would involve brute-forcing all possible assignments, which quickly becomes infeasible as the number of variables increases.

Satisfiable Example Consider the following formula [7], which is satisfiable:

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

This formula is satisfiable because there exists at least an assignment to the variables x_1, x_2, and x_3 that makes the entire expression true. For instance, the following assignment satisfies the formula: $x_1 = \text{true}$, $x_2 = \text{true}$, $x_3 = \text{false}$.

1.2 Modern Applications

SAT solving is part of the so-called symbolic AI, or rule-driven AI. Thanks to its ability to efficiently handle complex logical constraints, it has found applications across many areas of computer science [7].

In domains such as combinatorial decision making and optimization, SAT solvers enable the efficient exploration of large discrete search spaces. In these contexts, the focus is on satisfiability: finding an assignment that meets all constraints is often difficult, but SAT solvers can achieve it. They also play an important role in automated planning, where they are used to model and compute sequences of actions needed to achieve specific goals. Moreover, SAT-based techniques have been successfully applied to the verification and explanation of machine learning models, contributing to both their correctness and interpretability.

In the field of computer engineering, SAT is widely used to verify the correctness of certain optimizations. This validation is useful for both hardware and software improvements, provided that the components involved can be expressed using propositional logic [13]. In

this case, the interest often shifts to unsatisfiability: showing that the negation of an equivalence is unsatisfiable guarantees that the equivalence itself always holds. This is the same principle used in mathematical reasoning, for instance in the proof of De Morgan's law $\neg(A \wedge B) \equiv \neg A \vee \neg B$, where validity is established by demonstrating that the negation has no satisfying assignment."

1.2.1 Combinatorial Decision Making and Optimization Applications

In this field, the most used approach is to reduce a problem to SAT and then find a solution satisfying the formula using a SAT solver.

Discrete Geometry In discrete geometry, the problem of common unfolding of multiple boxes can be formulated in terms of polyominoes, where the goal is to determine whether a set of boxes can be unfolded into a single non-overlapping shape built from unit squares.

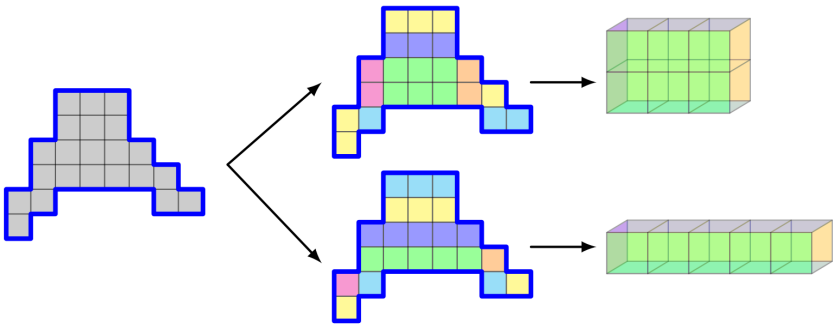


Figure 2: Unfolding multiple boxes

This task is highly combinatorial, since one must explore an exponential number of possible unfoldings. SAT solvers are applied by encoding the existence of a valid unfolding as a propositional formula: if the formula is satisfiable, the satisfying assignment corresponds to an explicit unfolding, while if it is unsatisfiable then no unfolding is possible [9].

Integer Programming Games Integer Programming Games (IPGs) are a powerful game-theoretic model where each player has a discrete set of strategies, which appear in areas like economics, transportation, and cybersecurity. The main solution concept is the pure Nash equilibrium, but deciding whether one exists is a hard problem that quickly becomes intractable as the game size grows. To address this, relaxed notions such as Locally Optimal Integer Solutions (LOIS) have been proposed, but even these can be challenging to compute with traditional optimization solvers. A promising approach is to encode the existence of LOIS as a propositional formula and apply SAT solvers: if the formula is satisfiable, the assignment corresponds to a stable solution profile, while unsatisfiability certifies that no such local optimum exists [10].

1.2.2 Engineering Applications

A very common approach in this field is to use SAT solvers to prove something, verifying that the contrary cannot hold because unsatisfiable.

Hardware Verification The following example on hardware verification provides a simple explanation of why SAT solvers are so effective.

Let F_{old} be the formula encoding the original version, and F_{new} the formula encoding the new (optimized) version. The goal is to prove that:

$$F_{\text{old}} \equiv F_{\text{new}}$$

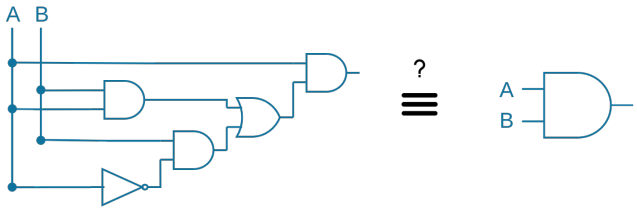
In propositional logic, this equivalence holds if the formula $F_{\text{old}} \leftrightarrow F_{\text{new}}$ is valid—i.e., true under all variable assignments.

Rather than directly proving this formula is valid, it is sufficient to show that its negation is unsatisfiable, meaning it is never true:

$$\neg(F_{\text{old}} \leftrightarrow F_{\text{new}})$$

If this negated formula is proven to be unsatisfiable, then no counterexample exists and the optimization is correct.

Consider the two logic circuits shown below. The goal is to verify whether they are functionally equivalent, meaning that for every possible input assignment, both circuits produce the same output.



We represent the behavior of the two circuits using propositional formulas:

$$C_1 = A \wedge ((B \wedge A) \vee (B \wedge \neg A))$$
$$C_2 = A \wedge B$$

We want to verify whether these two formulas are logically equivalent, that is:

$$C_1 \equiv C_2$$

This is equivalent to saying that the formula $C_1 \leftrightarrow C_2$ is valid, i.e., it evaluates to true under every possible assignment of truth values to A and B.

To verify this equivalence using SAT, we prove that its negation is unsatisfiable. Therefore, we check whether:

$$\neg(C_1 \leftrightarrow C_2)$$

is unsatisfiable. If the SAT solver determines that no assignment makes this formula true, then C_1 and C_2 are indeed equivalent.

It is important to highlight that a neural network can be trained either to predict whether two circuits are equivalent or to perform the optimization directly. In any case, the neural network cannot guarantee the correctness of the task, since it can only output metrics without providing explanations. On the other hand, SAT solvers can guarantee the correctness of the optimization and can even provide information on how to improve a given circuit.

Software Verification The same reasoning applies in the context of software optimization. Consider the following two program fragments:

Original

```
if(!a && !b) h();
else if(!a) g();
else f();
```

Optimized

```
if(a) f();
else if(b) g();
else h();
```

Both fragments can be encoded into propositional formulas F_{original} and $F_{\text{optimized}}$. For instance, an `if-then-else` structure can be represented as

$$\text{if } x \text{ then } y \text{ else } z \quad \equiv \quad (x \wedge y) \vee (\neg x \wedge z).$$

The correctness of the optimization reduces to proving that $F_{\text{original}} \leftrightarrow F_{\text{optimized}}$ is valid.

As in the hardware case, this is equivalent to checking that the negation $\neg(F_{\text{original}} \leftrightarrow F_{\text{optimized}})$ is unsatisfiable.

If the SAT solver confirms unsatisfiability, the two program fragments are guaranteed to be functionally equivalent for all inputs. This demonstrates how SAT solvers can provide formal guarantees about the correctness of software transformations.

2 Ancient History of Satisfiability

After having introduced SAT and some main reasons that justify its relevance, it is important to outline the history of logic and how it led to the concept of satisfiability [2].

2.1 Aristotle

Aristotle (384-322 B.C.) invented logic as a science in Athens in the fourth century B.C. In his mind the concepts related to our world are combined in subject-predicate propositions. For this reason he proposed four different ways to express this type of relation between subject (S) and predicate (P):

- universal affirmative (A): “all S is P”
- universal negative (E): “no S is P”
- particular affirmative (I): “some S is P”
- particular negative (O): “some S is not P”

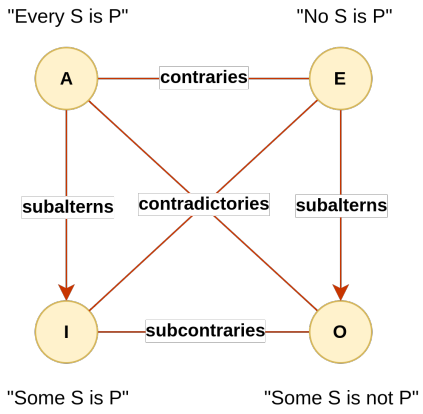


Figure 3: Aristotle subject-predicate propositions

The notion of satisfiability has been informally understood since the time of Aristotle. Although he did not formulate it explicitly, he used it implicitly for disproving statements by counterexample. The core idea behind a counterexample is to identify a case that falsifies a given hypothesis. In modern terms, we might say that there exists an ”assignment” to a formula representing the hypothesis that makes the contradictory statement true, thus proving the original hypothesis is actually false.

Let us consider the following logical implication:

If “every A is C” and “every B is C”, then “every A is B”

Aristotle would refute this implication by providing a counterexample, i.e., by assigning concrete categories to A, B, and C such that the premises are true, but the conclusion is false. For example A = “Scuola Ortogonale student”, B = “Scuola Ortogonale mentor” and C = “computer scientist”.

With this interpretation, we observe that even if

"every Scuola Ortogonale student is a computer scientist"

and

"every orthogonal professor is a computer scientist"

it is clearly not true that

"every Scuola Ortogonale student is a Scuola Ortogonale mentor"

This counterexample demonstrates that the original logical implication is not valid. In modern logical terms, this shows that the formula is not valid, as there exists a truth assignment under which the premises are true and the conclusion is false.

2.2 Boole and Venn

Aristotelian logic dominated for over two thousands years, until in the 19th century Boole (1815-1864) proposed its Boolean algebra. With this new structure, $\langle B, \wedge, \vee, \neg, 0, 1\rangle$, he provided an algebra-like notation for the elementary properties of sets, a more general theory of the logic of terms, as Aristotelian logic.

However, Boole was only interested in the theory of inference and its main goal was just showing that something is a logic consequence of something else. This means that starting from a knowledge base $p_1, ..., p_n$ and performing some algebraic manipulations, if q is obtained, we can say that q is a logical consequence of $p_1, ..., p_n$ ($p_1, ..., p_n \models q$). This can be done using Boolean algebra properties as commutativity, associativity, distributivity and others.

After Boole came Venn (1834-1923) with an even more ingenious improvement. He took Euler's circles and used them to graphically represent Boolean algebra. This brought to a diagrammatic method that pictured the structure of class logic with such visual clarity.

Let's consider three sets x, y, and z, and draw three circles to represent all possible intersections. Knowing that "some x are y" and "all y are z", we can easily infer that "some z are x". This becomes extremely clear when visualized: we know there is something in the intersection between x and y, and since all of y is contained in z, the region $y \setminus z$ must be empty. Therefore, the only part of the intersection between x and y that can contain elements is the region shared by x, y, and z.

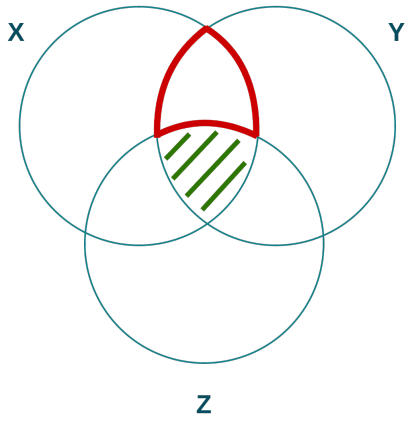


Figure 4: "some x are y" $\wedge$ "all y are z" $\rightarrow$ "some z are x"

2.3 Russell and Whitehead

Traditional logic is not able to represent inferences that depend on relations and Boole's logic, obviously, shares with Aristotle's the same inability.

Let's analyse the following example. If A=B and B=C then we would like to infer that C=A.

$$A = B \quad \wedge \quad B = C \quad \rightarrow \quad C = A$$

The problem is that in Aristotelian logic we have to express the relation of being equal to something as

"A is being-equal-to-B" and "B is being-equal-to-C"

which does not imply

"C is being-equal-to-A"

From this example, it's trivially evident that relations are not representable with traditional logic.

This weakness of traditional logic was overcome by B. Russell (1872-1970) and A. N. Whitehead (1861-1947) in the early years of the 20th century. Between 1910 and 1913, they published Principia Mathematica, a foundational work in mathematics that introduced an abstract symbolic system capable of describing both set theory and relations.

2.4 20th Century

During the 20th century, logic was mostly studied through axiomatic systems. The main objective was using inference for mathematical reasoning, determining what was derivable from logic and what was not, and so using first-order logic principally, which allows the description of relations.

Although the main focus of the time was proving mathematical theorems using logic, there were also several contributions more related to satisfiability and propositional logic. For example, in the early 1900s, Peirce (1839-1914) and Wittgenstein (1889-1951) introduced truth tables, which were really helpful to show if a certain propositional formula is valid, satisfiable, or unsatisfiable.

In the 1930s, Tarski (1902-1983) gave the first definition of satisfiability in first-order logic, which was extremely non-trivial due to the quantifiers used that make it more complex than the propositional one.

In the second half of the century, more precisely in 1958/1960, Davis (1928-2023) and Putnam (1926-2016) proposed using the conjunctive normal form, called CNF, as a standard for satisfiability testing. This would be possible because every propositional formula can be expressed in CNF. Although this leads to some preprocessing, introducing a little overhead in terms of computational cost, it allows to use more resolution rules, such as unit resolution.

2.5 Cook

In the second part of the 20th century, the advent of computers completely revolutionized theorem proving. The primary goal was to automate this process through the use of these machines, which, given an algorithm, could help determine whether a statement followed from a given logic.

The focus until this point had always been to determine whether something was true or false, derivable or not, a logical consequence or not. However, this changed when Stephen Cook (1939-) completely revolutionized the world of research in logic, showing that "decidable problems can be unsolvable due to space and time limitations."

He highlighted that, although an algorithm may be able to solve a certain problem in theory, this doesn't imply that the time or space required are reasonable for humans. From a human perspective, large amounts of time or space can be effectively considered infinite, and this explains why these algorithms cannot be used to practically solve a problem.

Research in logic focused on the question of decidability. After Cook the focus shifted to the complexity of resolution. To risk an exaggeration, one might even divide the history of computer science into two eras: *ante Cook* and *post Cook*.

This completely shifted the focus to the complexity issues associated with logical systems. SAT is a fundamental problem in complexity theory that gained importance thanks to Cook. The next chapter will explain why, after providing some background on complexity theory.

3 SAT, the First Problem NP-complete

The history of logic goes back thousands of years and has been fundamental since the time of the ancient Greeks. The concept of satisfiability emerged in the last few centuries, yet it has always been intrinsic to logic. However, determining whether a formula is satisfiable can be a very hard problem. This chapter provides some basic theoretical concepts on problem complexity, classifies SAT within this framework, and explains why it is both hard and of central importance.

3.1 Cook-Levin Theorem

In 1971, Cook published a foundational result in computational complexity theory [4]. Levin, who had obtained the same result independently but published his proof later, is also credited with the theorem.

Theorem (Cook-Levin Theorem)**.**

SAT is NP-complete.

This theorem demonstrated that SAT is a member of a class of problems known as NP-complete problems. In the following sections, background concepts from complexity theory will be provided to clarify the meaning of this result and explain why SAT became a central problem in theoretical computer science.

3.2 Complexity and Intractability

The complexity of a problem can lead to its intractability. In general, intractable problems can be divided into two main categories:

- **Undecidable problems:** problems for which no algorithm exists that can solve them. The most well-known example is the Halting Problem (Turing, 1936).
- **Intractable problems due to resource complexity:** problems for which a solving algorithm exists, but the required time, space, or both, grow so rapidly that they are practically unsolvable.

It is important to note that when a problem demands excessive space — for instance, if the solution is too large to be written down — it is often possible to reformulate the problem. Therefore, it is sufficient to consider problems that are intractable with respect to time complexity.

The time complexity function of an algorithm is defined as follows:

$$T(n) = \text{largest number of operations needed by the algorithm}$$

where n indicates the size of the input representing a certain instance of the problem.

This leads to the categorization of algorithms into:

- **Polynomial-time algorithms**: those whose time complexity function $T(n)$ is bounded by a polynomial function $p(n) = n^k$, which means that $T(n) \in O(p(n))$.
- **Exponential-time algorithms**: those whose time complexity function $T(n)$ is not bounded by a polynomial function $p(n)$, but rather by an exponential function $e(n) = k^n$, which means that $T(n) \in O(e(n))$.

The more n grows, the greater the difference between these two types of algorithms becomes. As n increases, an exponential function starts to grow much faster than a polynomial one.

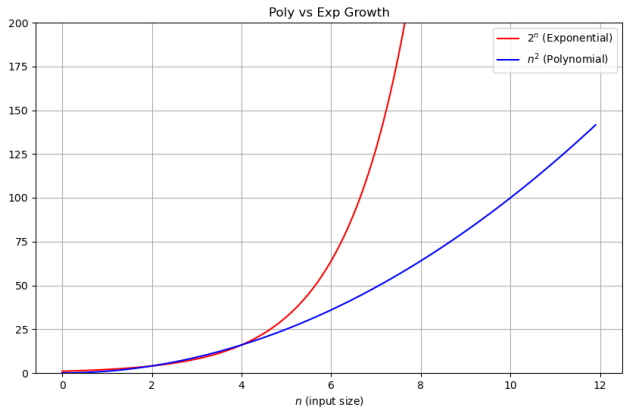


Figure 5: Comparison between polynomial and exponential functions

Another important point to specify is that, in most practical cases, the exponent k is small. In fact, reasonable algorithms typically have time complexity functions bounded by functions such as n^2, n^3, 2^n, or 3^n. Although one could imagine polynomial functions with huge bases, such as 10000^n, this would be unrealistic.

As a consequence, we can affirm that exponential-time algorithms require too much time in the majority of cases. There are some known problems which require a exponential-time algorithms to be solved efficiently, but they are very rare.

3.3 Decision and Optimization Problems

A decision problem consists of two parts:

- a generic instance of the problem (sets, graphs, numbers, functions...)
- a yes-no question in terms of the generic instance

An example is SAT, where the instance of the problem is a given propositional formula, and the yes-no question is whether it is satisfiable. A modified version of SAT that still

remains a decision problem is: *"Given a propositional formula, is it satisfiable by assigning fewer than k variables to true?"*, with $k \in \mathbb{N}$. This is still a decision problem.

On the other hand, an optimization problem consists of:

- a generic instance of the problem (sets, graphs, numbers, functions...)
- a question in terms of the generic instance that asks for an optimal solution

For example, we could slightly modify SAT so that the instance is still a propositional formula, but the question becomes: *"What is the minimum number of variables that must be assigned to true in order to satisfy the formula?"* This is a minimization problem. A maximization variant would also represent an optimization problem.

Optimization problems are harder than their decision form, because they require an optimal answer. If we can find the minimum number of variables that must be assigned to true (optimization problem), then we already know whether that number is lower than k (decision problem). However, by varying the bound k, it is possible to find the optimal answer by solving the decision problem multiple times, and this overhead is polynomial using techniques such as linear search or binary search. As a consequence, the theory of NP-completeness is generally formulated in terms of decision problems, since the additional polynomial overhead needed to obtain an optimal solution can be efficiently managed.

3.4 Complexity Classes

In this section, some of the main problem classes in complexity theory will be introduced. The goal is to understand what an NP-complete problem is, starting with the definitions of the classes P and NP. However, the definitions provided will be at a high level of abstraction, as the intention is not to construct a rigorous formal system, but rather to provide some basic knowledge - specifically, the concepts of deterministic and nondeterministic algorithms - to understand the meaning of these classes.

3.4.1 Deterministic Algorithms

A deterministic algorithm[8] is a computational procedure in which every step is precisely determined and reproducible. Given a specific input, the algorithm always produces the same output following a fixed sequence of operations. At each stage of computation, there is exactly one action to perform, with no element of chance or ambiguity involved. Formally, deterministic algorithms are modeled using deterministic Turing machines.

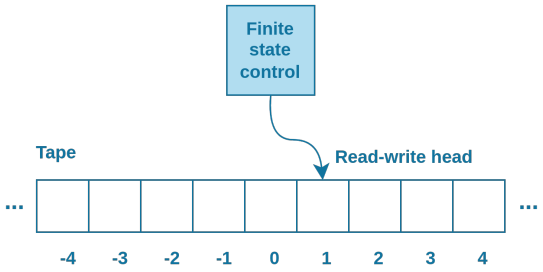


Figure 6: Deterministic Turing machine

A deterministic algorithm is said to operate in polynomial time if the number of steps it takes to terminate is bounded by a polynomial function of the size of the input.

3.4.2 Class P

The class P [8] is the class of all decision problems that can be solved in polynomial by a deterministic algorithm. Formally, a problem is in P if there exists a deterministic Turing machine that decides it and halts on every input of size n after performing at most $p(n)$ steps, where $p(n)$ is some polynomial function. Problems in P are generally considered to be efficiently solvable.

3.4.3 Nondeterministic Algorithms

A nondeterministic algorithm [8] is a theoretical computational model that, at each step, can choose between two possible actions. It is not bound to follow a single, predetermined sequence of operations. Instead, it can"guess" a solution by exploring many computational paths simultaneously. Formally, nondeterministic algorithms are modeled using nondeterministic Turing machines, which can branch into multiple computation paths at each step. A nondeterministic algorithm is said to run in polynomial time if there exists at least one computational path that leads to an accepting state within a number of steps bounded by a polynomial in the size of the input. This model allows us to bypass the exponential size of the solution space by assuming the ability to explore all possibilities in parallel. Therefore, what becomes relevant is whether a proposed solution can be verified in polynomial time to determine its correctness.

3.4.4 Class NP

The class NP [8] consists of all decision problems for which a given solution can be verified in polynomial time by a deterministic algorithm. Equivalently, NP can be seen as the class of problems solvable by a nondeterministic algorithm in polynomial time: if a solution exists, a nondeterministic machine can"guess" it and verify its correctness within a number of steps bounded by a polynomial function of the size of the input.

In this sense, NP captures the notion of problems that may be difficult to solve directly, but whose solutions are efficiently checkable once found. An example is SAT: finding an assignment that satisfies the formula is hard, but verifying if a certain assignment satisfies the formula is easy.

However, all the problems in P are also in NP. This inclusion can be easily shown: given a problem in P, by definition, there exists a deterministic algorithm that can decide it in polynomial time. To verify a candidate solution, we can simply execute the deterministic algorithm that solves the problem. Since the algorithm already decides the problem within polynomial time, verification is also completed in polynomial time. Thus, every problem that can be solved deterministically in polynomial time can also have its solutions verified deterministically in polynomial time, which means P $\subseteq$ NP.

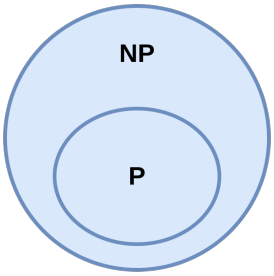


Figure 7: P $\subseteq$ NP

3.4.5 Polynomial Transformations

Before defining the concept of NP-completeness, it is still necessary to introduce another fundamental concept in complexity theory: polynomial transformations, also known as reductions.

A polynomial transformation [8] is simply a function computable in polynomial time that allows one to transform one problem into another. Since these transformations are performed in polynomial time, they do not introduce an intractable overhead in terms of computational resources required. For this reason, if a problem Π_1 can be solved in polynomial time and another problem Π_2 can be transformed into Π_1 in polynomial time, then Π_2 can also be solved in polynomial time.

The notation used in this article for polynomial transformations is the symbol $\propto$. Having two decision problems Π_1 and Π_2, $\Pi_1 \propto \Pi_2$ means that Π_1 can be reduced to Π_2.

Polynomial transformations allow us to create a kind of hardness hierarchy among problems. If $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$, then Π_1 and Π_2 are at the same level of hardness. If $\Pi_1 \propto \Pi_2$ but Π_2 cannot be reduced to Π_1, then Π_2 is harder to solve than Π_1.

3.4.6 Class NP-complete

Informally, a decision problem Π is NP-complete[8] if $\Pi \in$ NP and for every $\Pi' \in$ NP, $\Pi' \propto \Pi$. This essentially means that NP-complete problems are the hardest problems in NP, because every problem in NP can be reduced to an NP-complete problem. Therefore, solving an NP-complete problem in polynomial time would allow us to solve all problems in NP in polynomial time as well.

To prove that a certain decision problem Π is NP-complete, one must first show that $\Pi \in$ NP and then that every other problem in NP can be reduced to Π. While this approach is theoretically possible, it is impractical, since there are so many problems in NP. Instead, a much more convenient method is used: it suffices to show that a problem Π', already known to be NP-complete, can be reduced to Π in polynomial time. By proving this reduction, it follows that Π is at least as hard as Π'; and since Π' is NP-complete and $\Pi \in$ NP, we can conclude that Π is NP-complete as well.

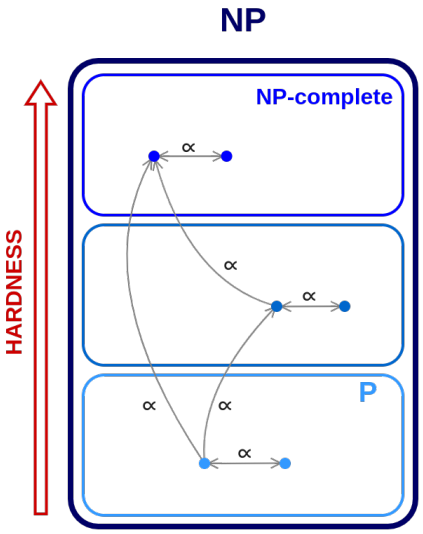


Figure 8: Hardness hierarchy in NP

3.5 Importance of SAT

SAT is considered fundamental in computer science for many reasons. Being an NP-complete problem, solving SAT in polynomial time would make it possible to solve all other problems in NP in polynomial time as well. This process would simply consist of reducing all these problems to SAT and then solving it. This is a truly ambitious goal, as no one has yet been able to prove that any NP-complete problem can be solved in polynomial time. However, neither has the contrary been proven. This remains one of the most fundamental open questions in computer science: does P = NP or P $\neq$ NP?

Solving SAT in polynomial time, or any other NP-complete problem, would prove that P = NP. On the other hand, proving that there are problems in NP that cannot be solved in polynomial time, and therefore that P $\neq$ NP, would have significant implications in other fields of computer science, such as cryptography, where problems that are not solvable efficiently correspond to systems that cannot be easily broken, ensuring security.

Another important aspect of SAT is the fact that it was the first problem to be discovered as NP-complete. It effectively established the highest level of hardness within NP, while at the same time being a very simple problem to understand. This facilitated the discovery of many other NP-complete problems, simply by showing that they are at the same level of hardness as SAT. As mentioned in the previous section, proofs of NP-completeness always rely on reductions from other NP-complete problems; thus, being the first NP-complete problem was a major contribution.

Finally, although it might seem contradictory with what has been discussed so far, modern SAT solvers are highly effective at solving SAT instances in practice. This means that we can often solve instances of computationally hard problems by reducing them to SAT instances and then applying a SAT solver. In the next section, we will focus precisely on how modern SAT solvers work.

4 Modern SAT Solvers

4.1 Introduction to SAT Solvers

As discussed so far, SAT is a computationally hard problem due to its NP-completeness. However, as already mentioned, modern SAT solvers are remarkably effective at solving instances involving millions of variables, while still maintaining impressive performance. Although this might seem surprising given the theoretical hardness of the problem, this section will explain the main techniques and approaches developed over the past three decades that have enabled such results.

A SAT solver is essentially a program that determines whether a given SAT instance is satisfiable and, if so, provides a satisfying assignment for the variables of the formula. Various types of solvers exist, employing different strategies; however, this research field is characterized by an open-source community, where progress in SAT solvers has emerged through incremental improvements starting from the earliest versions developed in the 1990s. At that time, the standard input format for solvers was a CNF instance of SAT, represented in the DIMACS CNF format, a textual encoding of formulas in conjunctive normal form. Today, many solvers no longer require a DIMACS CNF file directly to solve a SAT instance, but this is just one of the many advancements that have been introduced over the years and will be discussed in the following chapters.

Thanks to the SAT competition, which annually evaluates and ranks SAT solvers based on their performance in solving SAT instances, it is possible to systematically compare the efficiency of different solver versions.

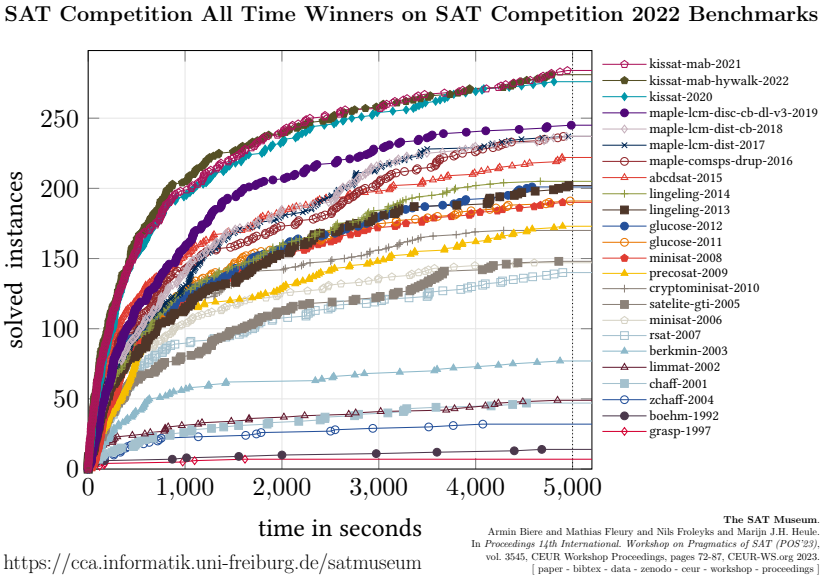


Figure 9: SAT museum (fig. from [14])

4.2 DPLL

DPLL [5] [12] stands for Davis-Putnam-Logemann-Loveland, and it is the name of the first algorithm used for Boolean satisfiability in the 1990s. This algorithm was originally introduced in 1962, but it was applied to SAT solving by solvers only thirty years later.

This algorithm is designed to be used with CNF formulas and is based on two main concepts: search and unit resolution.

By search, it is meant that the solver proceeds by assigning a value to a certain variable, building a search tree that is explored in a depth-first manner until a conflict or a solution is reached. If a solution is found such that the formula evaluates to true, the search is interrupted, and the solver returns that the formula is satisfiable. If one branch of the search tree fails, the other branch is explored through backtracking. Each variable can be assigned only one of two values, true or false, which implies that the search tree is binary. If both branches corresponding to the same variable fail, it means that neither value for that formula can satisfy the formula, and thus the formula will never evaluate to true.

The second important concept is unit resolution, which is used to force the assignment of certain variables to specific values. Unit resolution is used when we have a unit clause (a clause with a single literal). Suppose the unit clause is p. Then p must be true in every satisfying assignment. If there is another clause of the form $\neg p \vee W$, unit resolution simplifies it to W, since $\neg p$ is false under $p = \text{true}$. Formally:

$$\frac{p\ ,\ \neg p \vee W}{W}$$

By combining these two techniques, the DPLL algorithm emerges: it applies unit resolution as long as possible and then proceeds by assigning a value to a variable. This assignment represents a step in the search tree, after which unit resolution is applied again. This process alternates until either failure or success is reached.

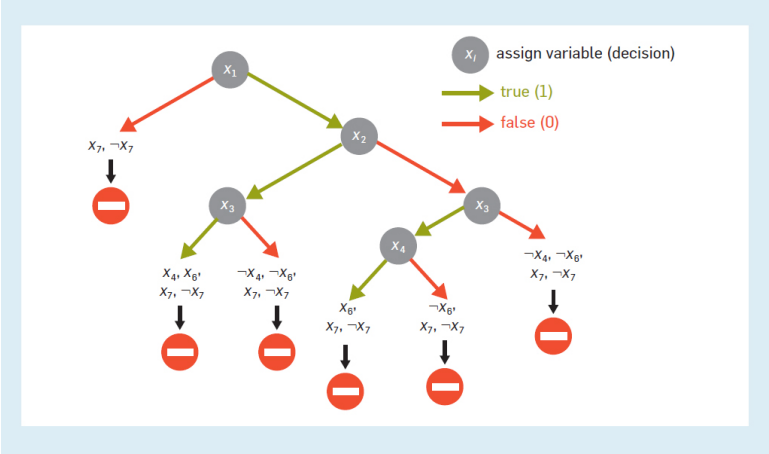


Figure 10: DPLL algorithm on 1 (fig. from [7])

This algorithm does not incorporate any optimization step beyond simple unit resolution, and for this reason it is not the most effective. However, it served as the starting point for many solvers that are still in use today.

4.3 CDCL

The DPLL algorithm is effective due to its simplicity; however, this simplicity also introduces some weaknesses. One major limitation is that, upon encountering a failure, the algorithm does not retain any information from the conflict. It is easy to observe that certain branches of the binary search tree explored by DPLL will never lead to satisfiability. However, because of the algorithm's design, which combines depth-first exploration with chronological backtracking, these unproductive branches will be explored regardless.

For this reason, CDCL (Conflict-Driven Clause Learning) [11] represents an evolution of DPLL. In CDCL, chronological backtracking is avoided by learning new clauses that allow the algorithm to back-jump to higher levels of the search tree. This mechanism significantly speeds up the process of determining whether a formula is satisfiable.

To make this possible, the relationships between variables are recorded in a structure called *implication graph*. In the event of a conflict, the implication graph enables both the derivation of a learned clause and the identification of the appropriate back-jumping point in the search tree. The details concerning the implication graph will not be discussed in this article.

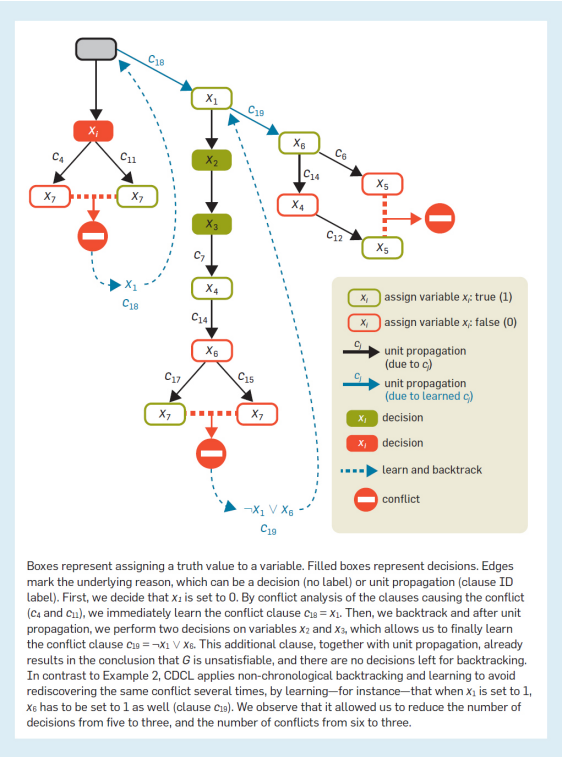


Figure 11: CDCL algorithm on 1 (fig. from [7])

4.4 SAT Solvers Nowadays

The CDCL algorithm represents the first significant improvement over the legacy DPLL approach. However, modern SAT solvers have evolved considerably [7], thanks to several clever techniques aimed at accelerating the satisfiability checking process.

Heuristics are often employed to guide the search. A clear example is the use of learned clauses in CDCL, which add valuable information to the formula. Another heuristic approach is the use of *restarts*: after an initial search, restarting from the root of the search tree - while retaining knowledge acquired during the previous attempt - can be highly effective. This strategy allows exploration of different parts of the tree and can lead to a solution more efficiently than continuing along a single search path.

Another powerful strategy involves the use of efficient SAT encodings. The relationships among variables in the propositional formula can significantly impact computation time. Efficient encodings, which may introduce additional variables, can reduce the time complexity of the encoding process. Although this may initially seem counter-intuitive, due to the increased number of variables, such encodings can transform cubic or quadratic complexity into linear or logarithmic complexity.

Modern SAT solvers also rely heavily on *preprocessing* and *in-processing* techniques, which

are essential components of the solving pipeline. These steps simplify the original formula by eliminating redundancies, identifying unit clauses, reducing the overall search space, and ultimately improving the solver performance.

Recent advances have allowed SAT solvers to prove the correctness of their answers. Verifying that a formula is satisfiable has always been straightforward: providing a satisfactory assignment suffices. However, proving that a formula is unsatisfiable has been always more challenging. In early SAT competitions, a formula was considered unsatisfiable if no solver could find a satisfying assignment. Over time, new proof systems were introduced and continuously improved. In the 2010s, techniques were developed that produce proofs which are compact, easy to generate, efficiently verifiable, and highly expressive.

In 2014, Moshe Vardi highlighted a particularly intriguing aspect of SAT solving. He observed that despite the remarkable performance of solvers on real-world instances, there is still a limited theoretical understanding of the reasons behind their success. This contrast is especially evident in their comparatively poor performance on random and cryptographic instances. The reason lies in the *hidden structure* present in real-world problems, which SAT solvers exploit implicitly. For this reason, SAT competitions distinguish between random and industrial (real-world) benchmarks, and dedicated solvers have been developed to perform better on random instances.

This observation has led to a line of research focused on identifying and leveraging structural properties within SAT instances, with the goal of explaining and further improving solver performance.

5 Encoding Orthogonal Hacked Phone

In this final section, we demonstrate how SAT, and in particular SAT solvers, can be applied in practice to efficiently solve problems of considerable complexity. To illustrate this, we introduce a simple variation of the well-known children's game Chinese Whispers, which we call Orthogonal Hacked Phone (OHP).

5.1 Game's Rules

In this game, one player takes on the role of the hacker, whose goal is to intercept and alter a secret message. The remaining players collaborate to ensure the message is transmitted correctly along a predetermined sequence of connections, while avoiding interception. The rules are as follows:

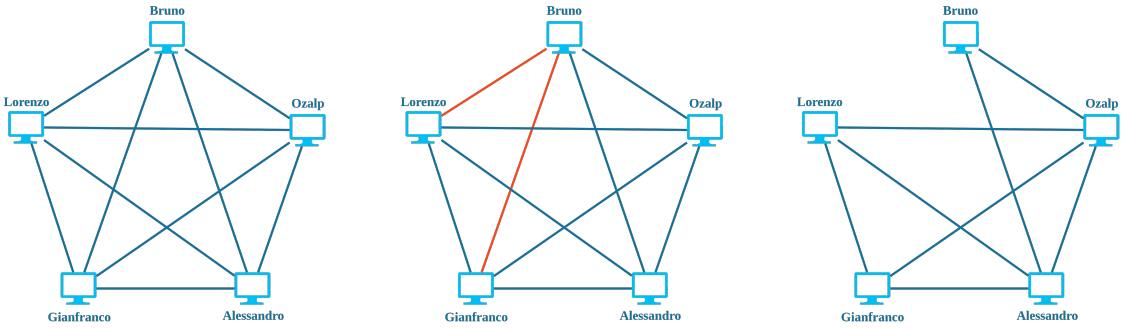
- the first player chooses a secret message;
- each player receives the message from exactly one other player, is allowed to view it only once, and must then forward it to the next designated player;
- the hacker may hack some of the connections between players, intercepting and modifying the transmitted message;

- the players win if, after passing through the entire sequence, the message returns to the first player unchanged;
- the hacker wins if the final message that reaches the first player has been modified.

5.2 OHP as an NP-complete Problem

In OHP, knowing the hacked connections allows us to determine whether the players have a chance to win the match or not. This problem can be seen as the famous NP-complete problem called the Hamiltonian Cycle problem. A Hamiltonian cycle in a graph is a cycle that starts at some vertex, visits every vertex of the graph exactly once, and then returns to the starting vertex.

If we represent the match as a graph, where the vertices are the players and the edges are the safe connections, determining whether there is a Hamiltonian cycle in the graph is equivalent to determining whether the players can win the match.



(a) Possible connections **(b)** Hacked connections **(c)** Safe connections

Figure 12: OHP in a graph

5.3 Encoding OHP in SAT

Now that it has been shown how OHP can be reduced to the Hamiltonian Circuit problem, we need to reduce Hamiltonian Circuit to SAT. In this way, by using any SAT solver, it becomes possible to solve the given problem efficiently.

The idea behind the encoding is the following: for n vertices in the graph, we need n^2 boolean variables. The assignment of these variables represents a possible cycle in the graph.

$$v_{i,j} = \begin{cases} 1 & \text{if player } i \text{ is in position } j \text{ of the cycle,} \\ 0 & \text{otherwise.} \end{cases}$$

For example, the following cycle is a possible solution:

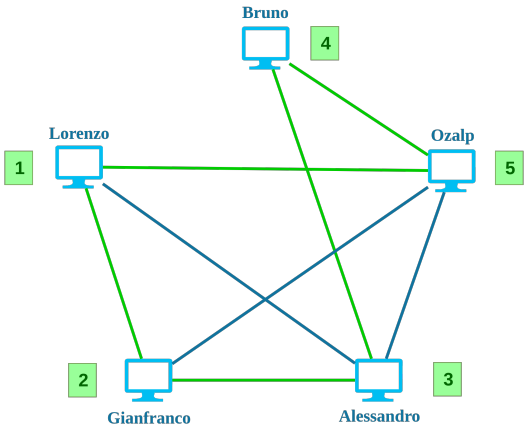


Figure 13: Solved OHP instance with 5 players.

This cycle is represented by a single assignment of the variables we have defined. In particular:

- $v_{L,1}$ is set to true because Lorenzo is the first player in the cycle,
- $v_{G,2}$ is set to true because Gianfranco is the second player in the cycle,
- $v_{A,3}$ is set to true because Alessandro is the third player in the cycle,
- $v_{B,4}$ is set to true because Bruno is the fourth player in the cycle,
- $v_{O,5}$ is set to true because Ozalp is the fifth player in the cycle,
- all the other variables are set to false.

Given the graph $G = (V, E)$, where V is the set of players and E is the set of edges (safe connections) between players, the following constraints must be imposed on the variables:

- exactly one player for each position:
  - at least one player for each position

$$\forall pos \in \{1, \dots, n\} \quad \bigvee_{i \in V} v_{i,pos}$$

  - at most one player for each position

$$\forall pl_1, pl_2 \in V,\ pl_1 \neq pl_2,\ \forall pos \in \{1, \dots, n\} \quad (\neg v_{pl_1,pos} \vee \neg v_{pl_2,pos})$$

- exactly one position for each player:
  - at least one position for each player

$$\forall pl \in V \quad \bigvee_{i=1}^{n} v_{pl,i}$$

– at most one position for each player

$$\forall i, j \in \{1, \ldots, n\},\ i < j,\ \forall pl \in V \quad (\neg v_{pl,i} \vee \neg v_{pl,j})$$

- consecutive players must be connected:

$$\forall pl_1, pl_2 \in V,\ (pl_1, pl_2) \notin E,\ \forall pos \in \{1, \ldots, n\} \quad (\neg v_{pl_1,pos} \vee \neg v_{pl_2,(pos+1) \bmod n})$$

After defining the variables and constraints, the solving step can be carried out with any SAT solver. As an example, a practical implementation using the Z3 solver [6] can be found at the following link: `https://github.com/KNGLJordan/Orthogonal-Hacked-Phone-SAT/blob/main/ohp.ipynb`.

6 Conclusion

In conclusion, the story of SAT illustrates how a problem that emerged from logic has become a foundational problem in modern computer science. SAT stands at the very heart of one of the greatest unsolved problems in computer science: the P vs NP question. As the first NP-complete problem, it not only represents a milestone of complexity theory but also connects to a wide range of other open problems across the field.

Despite its theoretical hardness, modern SAT solvers show that we can efficiently tackle many practical instances, making them powerful tools to address problems where data-driven AI often struggles to provide guarantees or explanations. This makes SAT solving an opened and interesting research area, with much still to explore and an high potential.

A related example is the Sport Tournament Scheduling (STS) [1] problem, a challenging case of combinatorial optimization. This problem highlights how real-world complexity can be addressed through multiple approaches, including SAT solving, and demonstrates the interplay between different optimization paradigms. An implementation can be found at: `https://github.com/KNGLJordan/Sport-Scheduling-Tournament.git`.

References

[1] Shaerf Andrea. Scheduling sport tournaments using constraint logic programming. *Constraints*, 1999.

[2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*. IOS Press, 2021.

[3] Wikipedia Contributors. Boolean satisfiability problem. `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`.

[4] Stephen Cook. The complexity of theorem-proving procedures. *New York, NY (USA), ACM Press*, 1971.

[5] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 1960.

[6] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*. Springer, 2008.

[7] Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of sat. *arXiv preprint arXiv:2007.01977*, 2020.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[9] Marijn J.H. Heule. Automated reasoning in discrete geometry: Discovery, verification, and symmetry. https://modref.github.io/ModRef2025.html.

[10] Pravesh Koirala, Aditya Shrey, and Forrest Laine. An Application of SAT Solvers in Integer Programming Games. In Jeremias Berg and Jakob Nordström, editors, *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:12, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[11] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1996.

[12] George Logemann Martin Davis and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 1962.

[13] University of Freiburg. Sat course. `https://cca.informatik.uni-freiburg.de/sat/ss25/`.

[14] University of Freiburg. Sat museum. `https://cca.informatik.uni-freiburg.de/satmuseum/`.