Pythagorean Triples and SAT Solving

Moti Ben-Ari

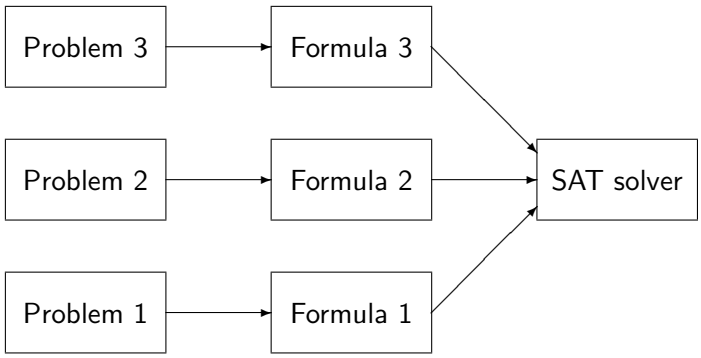
Department of Science Teaching

Weizmann Institute of Science

http://www.weizmann.ac.il/sci-tea/benari/

1 Introduction

SAT solving is a method of solving problems by translating a problem into a formula of propositional logic and searching for a satisfying assignment. The advantage of SAT solving is that one highly optimized program can be used to solve many different problems.



This document contains an overview of SAT solving and then describes how Heule and Kullman used a SAT solver to solve an open problem in mathematics.

Marijn J.H. Heule and Oliver Kullmann. The Science of Brute Force. *Communications of the ACM* 60(8), 2017, 70–79.

2 Satisfiability in propositional logic

Formulas in propositional logic are constructed from *atomic propositions* or *atoms* and operators. We use three operators: the unary operator $\neg$ (*not*, negation), and two binary operators $\wedge$ (*and*, conjunction) and $\vee$ (*or*, disjunction). We limit ourselves to formulas in *conjunctive normal form (CNF)*: a conjunction of *clauses* which are disjunctions of literals (atoms or negations of atoms). Here is a syntactically correct formula in CNF:

$$A = (\neg p \vee q \vee r) \ \wedge \ (\neg q \vee r) \ \wedge \ (\neg r) \, .$$

Because the structure of a CNF formula is known, set notation is frequently used:

$$A = \{\{\neg p, q, r\}, \ \{\neg q, r\}, \ \{\neg r\}\} \, .$$

The semantics of a formula in propositional logic is given by an assignment of the truth values $\{T, F\}$ to the atoms, and then evaluating the truth value of the formula. If there is an assignment such that the truth value of the formula is T, the formula is *satisfiable*.

The satisfiability of a formula can be decided by constructing a *truth table*, which has a row for each possible assignment to the atoms and whose last column gives the truth value of the formula for that assignment. Here is the truth table for the formula A:

p	q	r	A
T	T	T	F
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	T

The assignment in the last row causes the truth value of A to be T, so A is satisfiable.

The SAT problem is to construct an algorithm whose input is a formula in CNF and whose output is a satisfying assignment or a message that the formula is unsatisfiable. A *SAT solver* is program implementing the algorithm.

SAT is decidable using the truth table algorithm, but the algorithm is extremely inefficient, because there are 2^n rows, where n is the number of atoms.

For more detail on propositional logic and SAT solving, see Chapters 2, 4, 6 of:

M. Ben-Ari. *Mathematical Logic for Computer Science (Third Edition)*, Springer, 2012.

3 SAT is NP-complete

A problem Q is NP-complete if:

- A proposed answer to Q can be checked in polynomial time.
- All other problems in the class can be reduced to Q, meaning that if Q has a polynomial-time algorithm, then so do all the other problems.

Given an assignment for a formula A in CNF, it is easy to check if the truth value of A is T, so SAT fulfills the first condition. Stephen Cook (1971) and Leonid Levin (1973) showed that the second condition also holds, so SAT is an NP-complete problem.

The first condition states that an NP-complete problem can be solved in polynomial time by a *non-deterministic* algorithm. The question of whether NP-complete problems can be solved in polynomial time by a *deterministic* algorithm is called the $\mathcal{P} = \mathcal{NP}$? problem.

If there is an efficient (*deterministic polynomial-time*) algorithm for one NP-complete problem, then there are efficient algorithms for all the problems in the class. Currently, no efficient algorithm is known for any problem in the class.

If there is *no* efficient algorithm for one NP-complete problem, then there are no efficient algorithms for all the problems in the class. Currently, there is no proof that any of the problem in the class cannot be solved in deterministic polynomial time.

The Clay Mathematics Institute offers a $1 million prize for a proof that answers $\mathcal{P} = \mathcal{NP}$?. `http://claymath.org/millennium-problems/p-vs-np-problem`.

4 The DPLL algorithm for SAT

One need not become depressed upon finding out that a problem is NP-complete and therefore almost certainly does not have a efficient algorithm. The meaning of "does not have an efficient algorithm" refers to efficiency on *all* instances of the problem. In practice, we can be satisfied with a less strict requirement. We will be happy to have an algorithm for SAT that is efficient on *most* or *many* CNF formulas.

The DPLL algorithm for SAT was developed in 1960–62 by Martin Davis, Hilary Putnam, George Logemann and Donald Loveland. This algorithm is the basis of most modern SAT solvers. It can be proved that the DPLL algorithm is *not* efficient, because there is a family of formulas which it cannot solve in polynomial time. However, in practice, it is very efficient on many formulas.

The DPLL algorithm consists of two steps, repeated iteratively or recursively.

- **Decision** Choose an atom that has not yet been assigned and assign it T or F.
- **Unit propagation** *Unit clauses* (clauses that consist of one literal only) are used to simplify the formula.

If the result of these steps is a contradiction, the clause causing the contradiction is called a *conflict clause*. Upon reaching a conflict clause, the computation backtracks to try another decision (F instead of T for this atom, or an assignment to a different atom). If the result of these steps is T, we have found an assignment that satisfies the formula, so it is satisfiable.[1] If all assignments result in conflict clauses, the formula is unsatisfiable.

Let us try the DPLL algorithm on the formula A above:

$$A = (\neg p \vee q \vee r) \ \wedge \ (\neg q \vee r) \ \wedge \ (\neg p \vee \neg r) \wedge (p \vee q)\,.$$

Take a minute to understand that satisfiability is not affected if a *clause* containing a literal that evaluates to T is deleted from the *formula* or if a *literal* that evaluates to F is deleted from its *clause*.

- Decide to assign T to p. The fourth clause $p \vee q$ can be deleted, as can the literals $\neg p$ from the first and third clauses. The result is:

$$A' = (q \vee r) \ \wedge \ (\neg q \vee r) \ \wedge \ (\neg r)\,.$$

- Perform unit propagation. A' can be T only if the fourth (unit) clause $\neg r$ is T, which happens only if r is assigned F. The literal r can now be deleted from the first two clauses of A', resulting in:

$$A'' = q \wedge \neg q\,.$$

- Any decision to assign T or F to q results in the A'' becoming F, so the assignments $\{p = T, q = T, r = F\}$ and $\{p = T, q = F, r = F\}$ do not satisfy A.

[1] The assignment may be partial, but any arbitrary extension to a complete assignment still causes the formula to have the truth value T.

- The assignment $\{p = T, q = F, r = F\}$ *falsifies* the clause $\neg p \vee q \vee r$, which is called the *conflict clause* for the assignment.

5 The CDCL algorithm for SAT

The problem with the DPLL algorithm is that it blindly tries all possible assignments. From the truth table for the formula A, we know that only the assignment $\{p = F, q = F, r = F\}$ satisfies A.

When we applied the DPLL algorithm to A, once we had assigned $\{p = T, r = F\}$, the value assigned to q doesn't matter. In 1996, João P. Marques Silva and Karem A. Sakallah showed that when a conflict clause is found, the reason that it is contradictory can be discovered and expressed as a new clause. This *learned clause* can be added to the original CNF formula without changing whether it is satisfiable or not. This enables the DPLL algorithm to search fewer assignments because the learned clause will become a conflict clause from a small partial assignment. The method is called *conflict-directed clause learning* and is widely used in modern SAT solvers. See:

J. P. Marques-Silva, I. Lynce, S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, 131–153, in A. Biere, M. Heule, H. Van Maaren, T. Walsh (eds.), *Handbook of Satisfiability*, IOS Press, 2009.

LearnSAT is a SAT solver that I have developed that enables the user to obtain a detailed trace of the actions of the program. It comes with a tutorial and many examples of simple problems that can be solved by LearnSAT.

M. Ben-Ari. (2018). LearnSAT: A SAT Solver for Education. *Journal of Open Source Software*, 3(24), 639, https://doi.org/10.21105/joss.00639

6 Schur triples

This section demonstrates SAT solving on a mathematical problem.

> **Schur triples** Given *any* decomposition of the natural numbers $S = \{1, \ldots, n\}$ into two mutually exclusive subsets S_1, S_2, do there exist three numbers $a, b, c \in S_i$, such that:
>
> $$a = b + c\,,$$
>
> for at least one of S_1, S_2?

Example

For $n = 8$ and the decomposition:

$$S_1 = \{1, 2, 3, 4\}, S_2 = \{5, 6, 7, 8\}\,,$$

there is a Schur triple $\{1, 2, 3\}$ since $3 = 1 + 2$. For the decomposition:

$$S_1' = \{1, 2, 4, 8\}, S_2' = \{3, 5, 6, 7\}.$$

there are no Schur triples.

For $n = 8$, some decompositions contain Schur triples and some do not.

What about $n = 9$?

Theorem
For **all** decompositions of $S = \{1, \ldots, 9\}$ into two mutually exclusive subsets, at least one subset contains a Schur triple.

Proof
Trivial, check all $2^9 = 512$ decompositions.

OK. You and I are too lazy to check all of the decompositions. Let's try another way.

Proof
We prove the theorem by trying to prove its negation and obtaining a contradiction, that is, we try to find a decomposition of $\{1, \ldots, 9\}$ that *does not contain* a Schur triple.

Let us start by checking if 1 and 3 can be in the same subset, say, S_1. Then 2 can't be in S_1 because $3 = 2 + 1$, so it must in S_2. Similarly, 4 must be in S_2 since $4 = 3 + 1$. Continuing in this fashion, we find that 9 must appear in both subsets!

S_1	S_2
$1, 3$	
$1, 3$	$2, 4$
$1, 3, 6$	$2, 4, 7, 9$
$1, 3, 6, 9*$	$2, 4, 7, 9*$

Any decomposition with 1 and 3 in the same subset cannot contain a Schur triple.

If 1 and 3 can't be in the same subset, they have to be in different subsets. We can't deduce anything from having only a single number in a subset, so let us make a second decision: 5 is in S_2. It follows that 2 and 8 must be in S_1, because $5 = 3 + 2$ and $8 = 5 + 3$. 9 must be in S_2 because $9 = 1 + 8$.

S_1	S_2
1	$3, 5$
$1, 2, 8$	$3, 5$
$1, 2, 8$	$3, 5, 9$

If 4 is placed in S_1, $6 = 2 + 4$ must be in S_2, so $9 = 3 + 6$ must be in S_1, which is a contradiction.

$1, 2, 4, 8$	$3, 5, 6, 9$
$1, 2, 4, 8, 9*$	$3, 5, 6, 7, 9*$

On the other hand, if 4 is placed in S_2, $9 = 4 + 5$ must be in S_1, again a contradiction.

$1, 2, 8$	$3, 4, 5, 9$
$1, 2, 8, 9*$	$3, 4, 5, 6, 7, 9*$

Finally, let's try to put 5 in S_1. Again, this leads to a contradiction.

S_1	S_2
$1, 5$	3
$1, 5$	$3, 4, 6$
$1, 2, 5, 7, 9$	$3, 4, 6$
$1, 2, 5, 7, 9*$	$3, 4, 6, 9*$

We have shown that there is no decomposition of $\{1, \ldots, 9\}$ into two disjoint subsets, such that there is a triple a, b, c such that $a = b + c$, and all three numbers in the triple are not in the same subset S_1 or S_2. Without the negation: For all decompositions of $\{1, \ldots, 9\}$ into two subsets, at least one subset will contain a, b, c such that $a = b + c$.

Example $S_1 = \{1, 2, 5, 7\}$, $S_2 = \{3, 4, 6, 8, 9\}$ and $9 = 6 + 3$ in S_2.

7 Proving properties of Schur triples using SAT solving

We show how a SAT solver can find a decomposition of $\{1, \ldots, 8\}$ into two disjoint subsets such neither has a Schur triple; then we prove that there is no such decomposition for $\{1, \ldots, 9\}$. Traces of the computation by LearnSAT are given.

Encoding the Schur triple problem

There is an atom for each number in the sequence. For $n = 8$, these are:

```
x1,x2,x3,x4,x5,x6,x7,x8.
```

The intended meaning is that an atom is assigned F if the corresponding number is to be placed in the first subset, and it is assigned T if the corresponding number is to be placed in the second subset.

For each possible Schur triple contained in the sequence:

$$3 = 1 + 2,\ 4 = 3 + 1,\ \ldots,\ 7 = 3 + 4,\ 8 = 3 + 5,$$

there is a pair of clauses:

```
[x1,x2,x3], [~x1,~x2,~x3],
[x1,x3,x4], [~x1,~x3,~x4],
[x1,x4,x5], [~x1,~x4,~x5],
[x1,x5,x6], [~x1,~x5,~x6],
[x1,x6,x7], [~x1,~x6,~x7],
[x1,x7,x8], [~x1,~x7,~x8],
[x2,x3,x5], [~x2,~x3,~x5],
[x2,x4,x6], [~x2,~x4,~x6],
[x2,x5,x7], [~x2,~x5,~x7],
```

```
[x2,x6,x8], [~x2,~x6,~x8],
[x3,x4,x7], [~x3,~x4,~x7],
[x3,x5,x8], [~x3,~x5,~x8]
```

For example, for $7 = 3 + 4$, the clause `[x3,x4,x7]` specifies that at least one of the atoms is assigned T and the clause `[~x3,~x4,~x7]` specifies that at least one of the atoms is assigned F. Any assignment that satisfies both these clauses represents a decomposition such that $3, 4, 7$ are *not* in the same subset, so they do *not* form a Schur triple.

If we cannot find an assignment that satisfies the set of clauses, then there is no decomposition that does not contain a Schur triple. If we remove the confusing double negation: every decomposition contains a Schur triple.

Find a decomposition for $n = 8$

Here is the computation of the SAT solver:

```
LearnSAT v1.4.4.  Copyright 2012-13 by Moti Ben-Ari.  GNU GPL.
Decision assignment:  x1=0
Decision assignment:  x2=0
Propagate unit:  x3 (x3=1) derived from:  1.  [x1,x2,x3]
Decision assignment:  x4=0
Propagate unit:  x5 (x5=1) derived from:  5.  [x1,x4,x5]
Propagate unit:  x6 (x6=1) derived from:  15.  [x2,x4,x6]
Propagate unit:  ~x8 (x8=0) derived from:  24.  [~x3,~x5,~x8]
Propagate unit:  x7 (x7=1) derived from:  11.  [x1,x7,x8]
Satisfying assignments:
[x1=0,x2=0,x3=1,x4=0,x5=1,x6=1,x7=1,x8=0]
```

Only three decisions need to be taken: place `x1, x2, x4` in the first subset by assigning them F. Unit propagation quickly finds a satisfying assignment corresponding to $S_1 = \{1, 2, 4, 8\}, S_2 = \{3, 5, 6, 7\}$. It is clear that neither subset contains a Schur triple.

Prove that there is no decomposition for $n = 9$

The encoding has to include pairs of clauses for the additional triples:

$$9 = 1 + 8,\ 9 = 2 + 7,\ 9 = 6 + 3,\ 9 = 5 + 4\,.$$

Here is the computation of the SAT solver:

```
LearnSAT v1.4.4.  Copyright 2012-13 by Moti Ben-Ari.  GNU GPL.
Decision assignment:  x1=0,
Decision assignment:  x2=0
Decision assignment:  x4=0
Conflict clause:  30.  [~x3,~x6,~x9]
Decision assignment:  x4=1
```

```
Conflict clause:  28.  [~x3,~x5,~x8]
Decision assignment:  x2=1,
Decision assignment:  x3=0
Conflict clause:  20.  [~x2,~x5,~x7]
Decision assignment:  x3=1
Conflict clause:  18.  [~x2,~x4,~x6]
Decision assignment:  x1=1,
Decision assignment:  x2=0
Decision assignment:  x3=0
Conflict clause:  17.  [x2,x4,x6]
Decision assignment:  x3=1
Conflict clause:  19.  [x2,x5,x7]
Decision assignment:  x2=1,
Decision assignment:  x4=0
Conflict clause:  27.  [x3,x5,x8]
Decision assignment:  x4=1
Conflict clause:  29.  [x3,x6,x9]
Unsatisfiable
```

To save space, the unit propagation is not shown, but you can obtain it by running LearnSAT. We see that all assignments lead to conflict clauses so the formula is unsatisfiable, meaning that it is impossible to decompose $\{1, \ldots, 9\}$ into two subsets such that neither has a Schur triple.

Conflict-directed clause learning (CDCL)

The CDCL algorithm is quite complex, so I will just give an example. Consider the following formula:

```
[x1, x31, ~x2], [x1, ~x3], [x2, x3, x4],
[~x4, ~x5], [x21, ~x4, ~x6], [x5, x6]
```

After taking three decisions, unit propagation leads to a conflict clause:

```
Decision assignment:  x21=0
Decision assignment:  x31=0
Decision assignment:  x1=0
...
Conflict clause:  [x5,x6]
```

CDCL results in learning a clause which is added to the original set of clauses:

```
Learned clause:  [x21,~x4]
```

Three more decisions lead to a satisfying assignment:

```
Satisfying assignments:
[x21=0,x31=0,x1=1,x2=0,x3=1,x4=0,x5=0,x6=1]
```

When CDCL is used, only six decisions are taken as opposed to nine decisions using the DPLL algorithm alone. The learned clause enables the immediate assignment of F to `x4` since `x21` has been assigned false:

```
Propagate unit:  ~x4 (x4=0) derived from:  [x21,~x4]
```

8 Pythagorean triples

Pythagorean triples are similar to Schur triples except that the relation between the numbers is Pythagoras's theorem instead of the sum. A second difference is that we ask for a decomposition of all the natural numbers, not just a finite subsequence.

> **Pythagorean triples** Given **any** decomposition of the natural numbers N into two mutually exclusive subsets N_1, N_2, do there exist three numbers $a, b, c \in N_i$,
>
> $$a^2 = b^2 + c^2\,,$$
>
> for at least one of N_1, N_2?

Example Let the two subsets be the odd and the even numbers:

$$N_1 = 1, 3, 5, 7, \ldots$$
$$N_2 = 2, 4, 6, 8, \ldots$$

There are no Pythagorean triples in N_1, but $6, 8, 10 \in N_2$ and $10^2 = 8^2 + 6^2$.

Theorem

In **all** decompositions of N into two mutually exclusive subsets, at least one subset contains a Pythagorean triple.

There are an infinite number of decompositions of the infinite number of natural numbers, so it seems hopeless to use a finite representation as a formula to prove the theorem. However, it is sufficient to find **some** $n \in N$ such that for **any** decomposition of $\{1, \ldots, n\}$ into two mutually exclusive subsets, at least one subset contains a Pythagorean triple. The reason is that any decomposition of the infinite set N must also decompose the numbers $\{1, \ldots, n\}$ into these two subsets. If all decompositions of $\{1, \ldots, n\}$ contain a Pythagorean triple, then so does the infinite decomposition.

For example, suppose we are given any decomposition of N into N_1, N_2 and that we have proved that any decomposition of $\{1, \ldots, 20\}$ contains a Pythagorean triple (not true, but for the sake of the example suppose that it is true). Since there is a Pythagorean triple (such as $\{6, 8, 10\}$) within the decomposition of the numbers $\{1, \ldots, 20\}$ into these subsets, there is also one within the possibly infinite subsets N_1, N_2:

$$N_1 = \{1, 3, 5, 7, 12, 15, 16, 20\} \cup \{\text{numbers in } N_1 > 20\}$$
$$N_2 = \{2, 4, 6, 8, 9, 10, 11, 13, 17, 18, 19\} \cup \{\text{numbers in } N_2 > 20\}$$

The existence of Pythagorean triples can be encoded just like we did for Schur triples:

```
[x6,x8,x10], [~x6,~x8,~x10]
```

If in fact there is a decomposition without a triple in either subset, this can be easily found, as we did when we found the decomposition for Schur triples for $n = 8$.

Theorem

For any $n \leq 7824$, there is **some** decomposition of $\{1, \ldots, n\}$ into two mutually exclusive subsets, such that both subsets **do not** contain a Pythagorean triple.

Heule and Kullman proved this using a SAT solver in only one minute of computer time.

Then they proved:

Theorem

For **all** decompositions of $\{1, \ldots, 7825\}$ into two mutually exclusive subsets, at least one subset **contains** a Pythagorean triple.

This theorem is much more difficult to prove than the previous one, because there are 2^{7825} decompositions which must be checked to ensure that there is a triple in one subset.

In order to get an idea of the numbers involved, recall that we showed that there is no decomposition without a Schur triple for $n = 9$, because $2, 7$ must be in one subset, while $3, 6$ must be in the other subset, so that $9 = 2 + 7 = 3 + 6$ must be in both subsets which is impossible. Heule and Kullman found that 5180 and 5865 must be in one subset, while 625 and 7800 must be in the other subset, which leads to a contradiction since:

$$\begin{aligned} 5180^2 + 5865^2 &= 7825^2 \\ 625^2 + 7800^2 &= 7825^2 , \end{aligned}$$

which I'm sure that you remember from high school!

Generating and checking each decomposition would take roughly 10^{600} years. Since the age of the universe is estimated to be only 10^{10} years, this method is hopeless.

Using a highly optimized SAT solver, Heule and Kullman were able to prove the theorem in "only" $35,000$ hours of computing time. The computation was carried out in only two days using a computer with 800 cores working in parallel.

9 Can you trust a proof generated by a computer?

Most computer programs have bugs, so how can we trust a mathematical proof generated by a computer? This issue first arose with the proof of the four-color theorem in 1976. The proof reduced the theorem to proving a property of 1936 maps and these were checked by a computer program.

The proof of the theorem on Pythagorean triples used a new method to increase confidence in the result. The SAT solver wrote out a trace of formulas, such that if these formulas have a certain property then the proof is correct. This property was then checked by a relatively short and simple program, which, in turn, was verified mathematically. Still, this was a major undertaking because the trace required $200,000$ gigabytes to store.

10 Homework

Consider the following generalization of the Pythagorean triples problem:

> Given **any** decomposition of the natural numbers N into **three** mutually exclusive subsets N_1, N_2, N_3, do there exist three numbers $a, b, c \in N_i$, such that:
>
> $$a^2 = b^2 + c^2 \,,$$
>
> in at least one of N_1, N_2, N_3?

This will be solved if there exists some n such that for every decomposition of $\{1, \ldots, n\}$ into three mutually exclusive subsets, at least one subset contains a Pythagorean triple.

It is possible that the answer will never be found, because n, *if it exists*, is greater than 10^7.