The practice and theory of SAT solvers

Mathcamp 2025

Glenn Sun

These notes may contain errors,
and should not be considered an authoritative source.

Supplemental code can be found at:
`https://github.com/glenn-sun/mcsp25-sat`

Contents

1 Introduction to SAT

Mathematicians and computer scientists have long believed that some problems are easier to verify than to solve. This is known as the *P vs. NP conjecture*.

Definition 1.1 (P, NP)**.**

- *P* problems are yes/no questions that are fast and easy to check.
- *NP* problems are problems of the form, "Does there exist ____ such that ____?" where checking the condition is a P problem. ⌟

Example 1.2 (SUBSETSUM)**.** Consider the problem where you are given a list of numbers $a_1, a_2, \ldots, a_n$ and a target sum K, and you are asked, "Does there exist a subset of the a_i's whose sum is K?" It's easy to check this condition given a subset (just add them up!), so the the SUBSETSUM problem is in NP. To *solve* SUBSETSUM, of course you can check every subset, but there are 2^n of these, so it would take a long time. But it's not clear how to do this much faster. ⌟

Before we get too deep, we need to state exactly what we mean by fast and easy to check. In what follows, an *algorithm* is just some instructions to follow.

Definition 1.3 (polynomial time)**.** A problem is "fast and easy to check" if on inputs of size n, the problem can be solved using an algorithm taking $T(n)$ time, where there exists a polynomial $p(n)$ such that $T(n) < p(n)$. We also say that the problem takes *polynomial time*. ⌟

In practice, we will never write out $T(n)$ or $p(n)$ explicitly. As long as our algorithms never try to consider all subsets of a set, all functions from a set, or something of that sort, it will probably be fine. Note that for this definition, large polynomials like n^{10} are still considered "fast", despite being questionably useful in practice. However in practice, most polynomial time algorithms are about n^3 or faster.

To clarify an important point, NP does *not* stand for "not polynomial time." It actually standard for "non-deterministic polynomial time," the details of which are not relevant for this class. But the point is that there are many problems not in NP because they are even harder. We won't get into specifics, but an intuitive example is generalized version of games such as checkers or chess. If you ask, "In $n \times n$ checkers, does there exist a strategy that guarantees a win for black?" it is not clear how to quickly check that a strategy is winning for black: you would have to check all possible strategies for red.

***

One of the amazing facts about NP is that there are *complete problems* for NP. Complete problems are in some sense the hardest problems in NP.

Definition 1.4 (NP-complete)**.** A problem L is *NP-complete* if:

1. L is in NP, and
2. If we could solve L in polynomial time, then *every* problem in NP can be solved in polynomial time (by modeling it as an L problem). ⌟

In the 1960s and 1970s, Steven Cook and Leonid Levin independently discovered the first problems known to be NP-complete. In its usual form, the Cook–Levin theorem states that a problem known as Boolean satisfiability, or SAT, is NP-complete. That is, all other problems in NP can be reframed as instances of SAT.

Definition 1.5 (SAT)**.** A *Boolean formula* is a sentence using *Boolean variables* (which can be true or false) and some logical words, such as "and", "or", "not", "implies", "if and only if", and "xor" (exclusive or). Given a Boolean formula $f(x_1, \dots, x_n)$, the *Boolean satisfiability* (SAT) problem asks, "Does there exist an assignment to $x_1, \dots, x_n$ that makes the formula f true?" ⌟

Example 1.6 (SAT examples)**.** We write $\wedge$ for "and", $\vee$ for "or", and $\neg$ for "not".

- The formula $(x \vee \neg y) \wedge (\neg x \vee z)$ is satisfiable, by taking x true, y false, and z true (among other solutions). Check this by plugging in the assignment.
- The formula $(\neg x \vee y) \wedge (\neg y \vee z) \wedge x \wedge \neg z$ is not satisfiable. You can use some ad-hoc reasoning to see this. Note that NP requires us to check: given a purported solution, whether or not it works. When the answer is "no", it does *not* require us to efficiently prove that there are no solutions. ⌟

For decades (and often still now), mathematicians and computer scientists took NP-complete to mean "hard to solve and not worth trying," and thus did not try very hard to solve SAT. But a few dedicated people believed that it *was* worth trying to solve SAT. In the last 30 years, although all known algorithms still taking exponential time in the worst case, SAT solvers has become practical tools that can solve instances with thousands of variables and hundreds of thousands of constraints in seconds.

Rather than proving the Cook–Levin theorem, that all problems in NP can be reframed as instances of SAT, we will demonstrate how to reframe a small variety of problems as SAT instances.

Example 1.7 (logic puzzle)**.** A group of people accuse each other of being witches and make statements, some of which may be lies.

- A: B is a witch. I am not a witch.
- B: C is as honest as I am.
- C: The number of witches is even. A and I are not both truthtellers. I am not a witch.
- D: At least one of the witches is a truthteller.

Who are the witches? ⌟

Solution. We use 8 Boolean variables, for each of the 4 people being a truthteller and being a witch. While normally, we use single letters for variables in math, it will be much clearer to use words for this problem (and all future SAT encodings). We will also use symbols $\iff$ for "if and only if" and $\oplus$ for "xor".

- "A: B is a witch. I am not a witch." becomes

$$\text{ATrue} \iff (\text{BWitch} \wedge \neg\text{AWitch}).$$

- "B: C is as honest as I am." becomes

$$\text{BTrue} \iff (\text{CTrue} \iff \text{BTrue}).$$

- "C: The number of witches is even. A and I are not both truthtellers. I am not a witch." becomes

$$\text{CTrue} \iff (\neg(\text{AWitch} \oplus \text{BWitch} \oplus \text{CWitch} \oplus \text{DWitch}) \wedge \neg(\text{ATrue} \wedge \text{CTrue}) \wedge \neg\text{CWitch}).$$

  Note that $\oplus$ is addition mod 2, and when the sum is 0 we want true.

- "D: At least one of the witches is a truthteller." becomes

$$\text{DTrue} \iff ((\text{ATrue} \wedge \text{AWitch}) \vee (\text{BTrue} \wedge \text{BWitch}) \vee (\text{CTrue} \wedge \text{CWitch}) \vee (\text{DTrue} \wedge \text{DWitch})).$$

Our final Boolean formula is obtained by taking AND of all of these. After running a SAT solver, it tells us that one solution is that A and D are witches, and C is the only truthteller. Thus, we answer that A and D are witches (assuming that the logic puzzle has only one correct answer). □

***

While SAT generally allows all sorts of Boolean formulas as used in the formula above, it turns out that modern SAT solvers are optimized to run on a particular kind of formula called conjunctive normal form (CNF).

Definition 1.8 (CNF)**.** A formula in *conjective normal form* (a CNF) is an AND of clauses. A *clause* is an OR of (possibly negated) variables. ⌟

We think of the clauses as constraints, and we want all of them to be true. For example, the formulas in Example 1.6 (SAT examples) are CNFs. The final formula in Example 1.7 (logic puzzle) is a set of constraints that we want all to be true, but the individual constraints are not clauses, so it was not a CNF.

In the problems, you will provide a method to convert general Boolean formulas into CNFs without significantly increasing the size of the formula. In other words, CNFs are just as powerful as Boolean formulas in general. That being said, in the examples to follow, we will write the constraints as clauses to illustrate what an input to a SAT solver should actually look like.

Example 1.9 (sudoku)**.** Solve the following difficult sudoku puzzle.

	1		8		4	9		2
		7				6		
	4		2	7				
					9		5	8
		9				1		
2	7		1					
				3	2		9	
		2				5		
4		6	5		7		3	

⌟

Solution. We use 729 variables to encode this problem, all of the form (x,y) HAS i for $1 \le x, y, i \le 9$. The rules of sudoku are as follows:

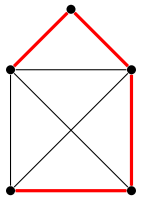
- Every cell has exactly 1 number. This splits into:
  - "Every cell has at least 1 number." becomes
    $$\bigvee_{i=1}^{9} (x,y) \text{ HAS } i$$
    for all $1 \le x, y \le 9$. This big $\vee$ means to take an OR over 9 variables.
  - "Every cell has at most 1 number." becomes
    $$\neg((x,y) \text{ HAS } i_1 \wedge (x,y) \text{ HAS } i_2)$$
    for all $1 \le x, y \le 9$ and all $\{i_1, i_2\} \in \binom{[9]}{2}$. The notation $[n]$ means $\{1, 2, \dots, n\}$, and $\binom{X}{k}$ means all k-element subsets of X. By de Morgan's laws, this is equivalent to
    $$\neg(x,y) \text{ HAS } i_1 \vee \neg(x,y) \text{ HAS } i_2,$$
    which is an OR as desired. (From here on, we will feel free to call $\neg(A \wedge B)$ a clause, because it is more intuitive to think about than $\neg A \vee \neg B$.)
- Every row has exactly 1 of each number. This splits into:
  - "Every row has at least 1 of each number." becomes
    $$\bigvee_{y=1}^{9} (x,y) \text{ HAS } i$$
    for all $1 \le x, i \le 9$.
  - "Every row has at most 1 of each number." becomes
    $$\neg((x,y_1) \text{ HAS } i \wedge (x,y_2) \text{ HAS } i)$$
    for all $1 \le x, i \le 9$ and all $\{y_1, y_2\} \in \binom{[9]}{2}$.
- Similarly, every column has exactly 1 of each number.
- Similarly, every 3×3 block has exactly 1 of each number.

In total, the rules give 11,988 clauses. Additionally, we will assert (x, y) HAS i for each pre-filled square in the puzzle. Our final CNF is the AND of all of these. After running a SAT solver, this sudoku puzzle is solved to be the following.

6	1	3	8	5	4	9	7	2
8	2	7	9	1	3	6	4	5
9	4	5	2	7	6	8	1	3
1	6	4	3	2	9	7	5	8
3	5	9	7	6	8	1	2	4
2	7	8	1	4	5	3	6	9
5	8	1	6	3	2	4	9	7
7	3	2	4	9	1	5	8	6
4	9	6	5	8	7	2	3	1

□

Example 1.10 (Hamiltonian path)**.** A graph is a set of nodes, some of which are connected by edges. A *Hamiltonian path* is a way to walk along the edges, visiting every node on the graph exactly once. For example, in the following graph, a Hamiltonian path is shown in red.



It is NP-complete to determine whether or not a given graph has a Hamiltonian path. Given a graph with nodes V and edges E, solve this problem using SAT. ⌟

Solution. Let $n = |V|$. We create n^2 variables: v INPOS i for all $v \in V$ and $i \in [n]$, meaning that we have a path of length n, and vertex v is the ith vertex along the path. Using these variables, we have the following clauses:

- Each node appears exactly once on the path. The translation of this is similar to Example 1.9 (sudoku), and contributes $n + n \cdot \binom{n}{2}$ clauses.
- Each position on the path contains exactly one node. Again, this is similar to Example 1.9 (sudoku) and contributes $n + n \cdot \binom{n}{2}$ clauses.
- Every non-edge pair of vertices is not adjacent on the path. This translates into the two clauses

  $$\neg(u \text{ INPOS } i \wedge v \text{ INPOS } i+1) \qquad \neg(v \text{ INPOS } i \wedge u \text{ INPOS } i+1)$$

  for all $\{u, v\} \in \binom{V}{2} \setminus E$ and $i \in [n-1]$. This adds $2(n-1)(\binom{n}{2} - |E|)$ clauses.

In total, this translation uses n^2 vertices and somewhere approximately between n^3 and $2n^3$ clauses, using $\binom{n}{2} \approx 0.5n^2$. Since a good rule of thumb is that modern SAT solvers can handle a few thousand variables and a couple hundred thousands of clauses, this should be generally feasible for $n = 50$ (where $n^2 = 2{,}500$ and $2n^3 = 250{,}000$), but infeasible for graphs much larger than that. □

Practice

Do these problems to reinforce the main concepts from the lesson.

1. The pigeonhole principle is the statement that given n pigeons and m holes, there is a way to assign holes to the pigeons such that no hole gets two pigeons. The formula PHP_n^m should be satisfiable for $n \leq m$ and unsatisfiable for $n > m$. A priori, we allow one pigeon to be assigned multiple holes. Define the appropriate variables and encode the pigeonhole principle as a CNF. Approximately how many variables and clauses are you using?
2. The n-queens puzzle asks, on an $n \times n$ chessboard, find a way to place n queens so that no two queens can attack each other. In chess, a queen can attack all pieces in her row, her column, and both of her diagonals. Define the appropriate variables and encode the n-queens puzzle as a CNF. Approximately how many variables and clauses are you using?

3. Given a Boolean formula involving symbols $\wedge$, $\vee$, $\neg$, $\iff$, $\Rightarrow$, $\oplus$, and some variables, provide a method that converts it into a CNF that is satisfiable if and only if the original formula is. The length of the formula (measured by the number of symbols) should not increase by more than a factor of 25 or so. This is called a *Tseitin transformation*. (Hint: Introduce a new variable to hold the value of each subformula, i.e. if given $A \Rightarrow (B \oplus C)$, introduce X_1 to mean $B \oplus C$ and X_2 to mean $A \Rightarrow X_1$.)

Extensions

Do these problems if you are interested in additional content.

4. For many problems, including the examples discussed in the lesson, the majority of clauses come from encoding "at most one of $x_1, \ldots, x_n$" is true. We wrote $\neg(x_i \wedge x_j)$ for $\{i, j\} \in \binom{[n]}{2}$, which costs about $0.5n^2$ clauses. We can do much better if we're willing to trade some clauses for some variables. Follow these parts to determine a method to encode such a statement using only about $n \log_2(n)$ clauses, at the cost of introducing $n - 1$ new variables.
   (a) Let $A(n)$ satisfy the recurrence $A(n) = 1 + A(\lfloor n/2 \rfloor) + A(n - \lfloor n/2 \rfloor)$, with $A(1) = 0$. Prove that $A(n) = n - 1$.
   (b) Let $B(n)$ satisfy the recurrence $B(n) = n + B(\lfloor n/2 \rfloor) + B(n - \lfloor n/2 \rfloor)$, with $B(1) = 0$. Prove that $B(n) \leq n\lceil \log_2(n) \rceil$.
   (c) Recursively define an encoding for "at most one of $x_1, \ldots, x_n$," relying on the fact that you already know encodings of "at most one of $x_1, \ldots, x_{\lfloor n/2 \rfloor}$" and "at most one of $x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n$." This encoding should use 1 new variable and n new clauses. Apply the previous parts to prove the claim.
5. (Optional, requires Python programming) Install the Z3 solver as a Python module and implement your solution to Problem 2, then solve the n-queens puzzle for $n = 9$. Visit `https://github.com/glenn-sun/mcsp25-sat` for starter code.

2 SAT for formal verification

Being the canonical NP-complete problem, SAT has historically attracted a lot of interest from mathematicians and theory-minded people who care about big open problems like P vs. NP. But today, as P vs. NP appears increasingly elusive, and SAT *solving* becomes increasingly powerful, the people who care most about SAT often now live in programming languages departments. This is because the most important application of SAT solving is in verifying code to be correct.

Code is said to be correct if the *implementation* matches a *specification*. In other words, for all inputs, the program outputs what it should. From a low degree of confidence to a high degree of confidence (and simultaneously low effort to high effort), the following techniques are often used to determine if code is correct:

- Handwritten unit tests, where programmers come up with a couple of inputs and outputs, and make sure that the program does the right thing on these inputs.
- Random unit tests, where programmers randomly sample inputs and hope that most errors are common errors. However, this is not a great assumption because edge case very often produce errors.
- *Bounded model checking*, where the computer formally verifies that the code works for all inputs resulting in bounded execution length. There are two techniques for this, the naive technique and the SMT technique, which extends SAT.
- Formalized proof, which automatically verifies all inputs, of which there are two main types. One is a handwritten mathematical proof formalized in Lean, Rocq, or similar proof assistant. The other is again, SMT solving.

We will focus on bounded model checking, which gives a very strong guarantees that code is correct. In practice, because these methods can still struggle to scale with large codebases and require a significant amount of developer assistance in writing a formal specification, both of these techniques are mainly only used for extremely critical systems. One example is HACL*, a cryptographic library used in the Linux kernel and Firefox's networking systems. Early Intel Pentium processors had a bug in its division instruction, costing them $495 million in 1994 US dollars. Since then, Intel began to use formal tools such as the ones we will discuss today to verify their designs.

In order to check and manipulate code, the computer must first convert the code into a form that is easy to work with. This is typically called *static single assignment* (SSA).

Example 2.1 (static single assignment form)**.** Consider the following Python code:

```
def division(x, y):
    r = x
    q = 0
    while r >= y:
        r = r - y
        q = q + 1

    assert x == y * q + r
    assert 0 <= r and r < abs(y)
    return (q, x)
```

The code computes the division algorithm on x and y using repeated subtraction. An "assert" statement is a check that the condition stated actually holds. If the condition evaluates to false when the computer gets to that point, it will throw an error. Our assertions guarantee the correctness of this algorithm by stating what the specification of the output is. Proving this algorithm correct is equivalent to proving that the assertions never evaluate to false, for all inputs x and y.

Before we start, first note that this algorithm is actually incorrect. The programmer assumed that $x \geq 0$ and $y > 0$. For inputs that are 0 or negative, this algorithm will produce the wrong output or run forever. With bounded model checking, we will be able to automatically find inputs that produce wrong output, but we will be unable to detect infinite loops. That is to be expected—the halting problem is undecidable!

Our goal is to do bounded model checking, and by "bound", we mean that we only care about computations that end within a fixed time. In other words, loops get *unrolled* a fixed number of times, say for instance, at most twice. That transforms the code into:

```
def unroll2_division(x, y):
    r = x
    q = 0
    if r >= y:
        r = r - y
        q = q + 1
    if r >= y:
        r = r - y
        q = q + 1
    assume r < y

    assert x == y * q + r
    assert 0 <= r and r < abs(y)
    return (q, r)
```

Note that we've introduced a new keyword not typically found in Python—"assume". That's okay, because we're not actually going to run the code. The assumption says that we will only care about instances of the function where this loop actually ended in at most 2 iterations. Formally, "assume P" means whenever "assert Q" appears in the future, it actually means "assert $Q \vee \neg P$".

Assumptions can also be used when, for example, you write a function that is supposed to accept positive integers, but there is no built-in type for this. (Only integers and non-negative integers are available in most languages.) In that case, you would write "assume $x > 0$."

Then, static single assignment means that all variables should be assigned exactly once. That means instead of updating the value of variables as we are doing above, we should number each instance of each variable and create a new one each time we want to assign something.

Note that SSA requires special care when considering if-statements. When a variable a_1 gets assigned inside an if-statement and we give it a new name a_2, how do we tell future lines whether to refer to a_1 or a_2? We resolve this by adding an explicit else branch that maintains the value, and joining the two branches with a new primitive operation called "ite", which stands for if-then-else. In particular, if the if-condition is b, then we write $a_3 = \texttt{ite}(b, a_2, a_1)$. This also completely flattens the code.

One last modification that we make is to avoid nesting operations, and give every subexpression its own variable name. This mostly affects the end, when we are asserting $x = yq + r$. All of these changes produce the following:

```
def ssa_division(x, y):
    r1 = x
    q1 = 0
    b1 = r1 >= y
    r2 = r1 - y
    q2 = q1 + 1
    r3 = ite(b1, r2, r1)
    q3 = ite(b1, q2, q1)
    b2 = r3 >= y
    r4 = r3 - y
    q4 = q3 + 1
    r5 = ite(b2, r4, r3)
    q5 = ite(b2, q4, q3)
    assume r5 < y

    temp1 = y * q5
    temp2 = temp1 + r5
    abs_y = abs(y)
    assert x == temp2
    assert 0 <= r5
    assert r5 < abs_y
    return (q5, r5)
```

This code is now in static single assignment form. Given a complete specification of the allowed kinds of statements in a programming language, conversion to SSA can be done automatically. ⌟

Definition 2.2 (static single assignment form)**.** A program is in *static single assignment* (SSA) form if there are no loops, no branching statements, and every variable is assigned exactly once. ⌟

Example 2.3 (bit blasting)**.** In order to verify the program above, we use a technique called *bit blasting*. On a computer, the above variables are not mathematical integers, but rather represented in n bits and can only hold values -2^{n-1} through $2^{n-1} - 1$. Typically, $n = 32$ or $n = 64$, but for the sake of example, we will use $n = 4$.

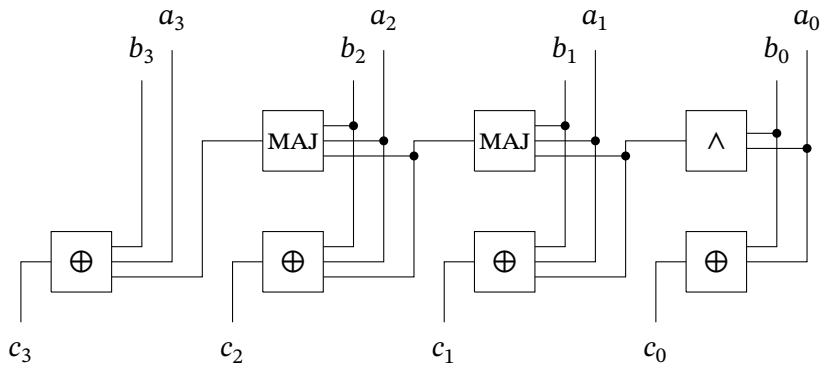
On most modern computers, non-negative integers 0 through $2^{n-1} - 1$ just get their standard binary representation. Negative integers use the *two's complement*. For a negative number $-x$, this refers to the binary representation of the number y such that $x + y = 0 \pmod{2^n}$. That gives the following conversion table:

0	1	2	⋯	7	−8	−7	⋯	−1
0000	0001	0010	⋯	0111	1000	1001	⋯	1111

This means that we can replace every integer variable in the previous example with 4 boolean variables, and use logical circuits to compute addition, subtraction, multiplication, absolute value, equality, $\geq$, and `ite`. These logical circuits, which are

built up from things like $\wedge$, $\vee$, $\neg$, and $\oplus$, can then be converted into (relatively) short CNFs via a Tseitin transformation as in Problem 1.3.

For example, a 4-bit adder circuit looks like the following, which mimics the standard addition-with-carry process that you learned in elementary school, with $\wedge$/MAJ representing the carry bit and $\oplus$ representing the addition mod 2 without carry.



(Above, the majority function $\text{MAJ}(a, b, c)$ can be implemented as $(a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$.) Thus, if $c = a + b$ was a line in the code, we would create a variable for each input, output, and intermediate gate in this circuit. For example, one of the gates in this circuit says $c_0 = a_0 \oplus b_0$. This can be expanded into

$$c_0 \iff (a_0 \vee b_0) \wedge (\neg a_0 \vee \neg b_0),$$

which can be further expanded into a CNF with clauses

$$(a_0 \vee b_0 \vee \neg c_0) \qquad (\neg a_0 \vee \neg b_0 \vee \neg c_0) \qquad (a_0 \vee \neg b_0 \vee c_0) \qquad (\neg a_0 \vee b_0 \vee c_0).$$

Finally, if $C_1, \dots, C_n$ are the clauses derived from all of these boolean circuits, $a_1, \dots, a_m$ are the boolean variables corresponding to the assertions, and $m_1, \dots, m_k$ are the boolean variables corresponding to the assumptions, we ask our SAT solver if the set of clauses

$$C_1 \qquad \cdots \qquad C_n \qquad (\neg a_1 \vee \cdots \vee \neg a_m) \qquad (m_1) \qquad \cdots \qquad (m_k)$$

is satisfiable. If it is, that means we have found a counterexample that satisfies all our equations and assumptions, but fails at least one assertion. If it is unsatisfiable, our SAT solver has proven that the code works! ⌟

Theorem 2.4 (bounded model checking)**.** *Let $k \in \mathbb{N}$. Given a computer program f that contains some assumptions and assertions , the following algorithm produces a CNF whose negation is equivalent to the theorem: "For all inputs x, if the computation $f(x)$ meets all assumptions and takes at most k iterations to terminate each loop, then the assertions in $f(x)$ hold." This is called* bounded model checking *with k unrolls.*

1. *Unroll loops k times and add termination assumptions.*
2. *Rewrite the code in static single assignment form.*
3. *Use bit blasting to create a CNF for the assignments, and adjoin the assumptions and the negation of the assertions.* ⌟

***

While bit blasting is an effective method that works for all programs (after all, everything you can do on a computer has a hardware implementation as a logical circuit), it can be slow and impractical for larger programs, especially if you want to test against real 32-bit or 64-bit data types. If you want to model mathematical data types, such as actual integers without overflow or real numbers, you are out of luck completely with bit blasting. One alternative method is called *SAT modulo theories*, or SMT solving.

A *theory* is, roughly speaking, a mathematical domain in which you choose to work. The *atoms* of a theory are the sentences that don't involve quantifiers ($\forall$, $\exists$) or logical connectives ($\wedge$, $\vee$, etc. but $\neg$ is allowed). Here are some common examples:

- Booleans: This is the theory that we've been using all along. This theory just has boolean variables. Atoms are the variables and their negations. You have seen many examples of formulas already.
- Equality of uninterpreted functions (EUF): In this theory, we have functions, variables, and equality, but we don't care what the functions mean. Atoms look like $f(a) = a$ or $g(x, y) = g(y, x)$. An example of an unsatisfiable formula is

$$[f(f(a)) = a] \wedge [f(f(f(a))) = a] \wedge [f(a) \neq a].$$

- Linear real arithmetic (LRA): This theory includes linear inequalities over real numbers. An example of an atom is $3x + \pi y \leq \sqrt{2}$. This theory also supports equality, because $a = b$ is equivalent to $[a \leq b] \wedge [a \geq b]$. This theory is sometimes also called the theory of linear programming. An example of an unsatisfiable formula is

$$[x + y \geq 1] \wedge [x \leq 0] \wedge [y \leq 0].$$

- Linear integer arithmetic (LIA): The same as LRA, but with integers as the domain.
- Nonlinear real arithmetic (NRA): The same as LRA, but with polynomials.
- Specialized theories such as the theory of arrays, theory of bit vectors, etc. which are used by programmers when thinking about code.

Mathematicians, logicians, and computer scientists have long been hard at work developing specialized tools to decide satisfiability of *conjunctions* (ANDs) of atoms in these theories. For example,

- With EUF, the *congruence closure* algorithm decides satisfiability of conjunctions of atoms in polynomial time.
- With LRA, the *simplex* algorithm decides satisfiability of conjunctions of atoms very quickly in practice, although exponential in worst case. Polynomial time algorithms for this are known, with large constants that make the simplex algorithm typicaly better in practice.
- Deciding LIA is NP-complete, but there are a variety of techniques such as *cutting planes* and *branch-and-bound* that make this doable for reasonably sized problems.
- Deciding NRA is even harder, but tools like *cylindrical algebraic decomposition* (CAD) and *Gröbner bases* allow reasonable progress that would otherwise be impossible when reasoning with mathematical real numbers.

All of these specialized tools have focused on conjunctions of atoms, where each atom is thought of as a constraint. However, real-world use of these theories often includes ORs and other logical connectives, especially when using some of these theories to model a computer program. A key technique known as DPLL(T)[1] is then used to combine the power of SAT solving and these specialized theory solvers.

Definition 2.5 (DPLL(T))**.** Consider a CNF f_T whose atoms are expressed in a theory T, and we have a solver S that can decide conjunctions (ANDs) of atoms from T. The following procedure determines if f_T is satisfiable, and is called DPLL(T):

1. Replace the atoms in f_T with boolean variables, getting f_B.
2. Repeat the following infinitely:
   (a) Solve f_B using a SAT solver.
   (b) If f_B is satisfiable with a boolean solution x_B,
      i. Convert x_B back to atoms in T, getting x_T (the reverse of step 1).
      ii. Solve the conjunction of atoms x_T using S.
      iii. If x_T is satisfiable, return satisfiable.
      iv. Otherwise, if x_T is unsatisfiable, replace f_B with $f_B \wedge \neg x_B$.
   (c) Otherwise, if f_B is unsatisfiable, return unsatisfiable.

Note that when x is a boolean solution, say $(x_1 \wedge \neg x_2 \wedge x_3)$ expressing that x_1 is true, x_2 is false, and x_3 is false, the expression $\neg x$ is indeed a clause by de Morgan's laws. ⌟

SMT solving is the use of DPLL(T) or its variants. There are a few ways that people commonly extend SMT solving to automatically prove more kinds of statements:

- Often times, your mathematical domain is not just one of the theories listed previously, but you need to reason about multiple theories all at once. The tool to do this is called the *Nelson–Oppen* procedure. The basic procedure combines a solver S_1 for T_1 and a solver S_2 for T_2 into a combined solver S for $T_1 \cup T_2$. Then simply run DPLL($T_1 \cup T_2$) with this new solver.
- You may also want to prove statements that involve more quantifiers such as $\forall$. SMT solvers will use additional techniques such as *e-matching* to handle such cases, but they are not as easy to handle generally as the simple existential case in DPLL(T).

SMT solvers can function as a drop-in replacement for bit blasting for the purposes of bounded model checking. But they are also often used more broadly. Their speed and flexibility allow them to be used to prove statements for unbounded loops, called *loop invariants*, provided that the invariants are first written down by the programmer for the solver to check. These invariants say what facts remain true after every iteration of the loop. Many programming languages have add-ons that allow programmers to specify these invariants while writing their code.

[1]The algorithm is called DPLL(T) in the literature, but really uses CDCL. Both of DPLL and CDCL are algorithms that we will see in the next section. This is because there are some disagreements in the community of whether CDCL is its own algorithm, or just a variant of DPLL.

Practice

Do these problems to reinforce the main concepts from the lesson.

1. In some programming languages, there is something called a "for-loop". It usually looks something like:

   ```
   for (i = 0; i < n; i++) {
       do something depending on i
   }
   ```

   The meaning of this syntax is to repeat the inside of the loop n times, once each for $i = 0, \dots, n-1$. Suppose you were doing loop unrolling with 2 unrolls. How would you convert this code fragment into SSA?

2. Write a logical circuit that computes $z = \texttt{ite}(b, x, y)$ for a boolean variable b and x, y, and z that are 2 bits each.

3. The algorithm for DPLL(T) was written with an infinite loop. Explain why DPLL(T) always terminates in finite time.

Extensions

Do these problems if you are interested in additional content.

4. In this problem, you will explore the congruence closure algorithm to create a solver for the theory of equality of uninterpreted functions.

   The setup is that you have some function symbols $f_1, f_2, \dots, f_m$ and some variables $x_1, \dots, x_n$. A *term* is an expression made up of these above things. The *atoms* are $t_1 = t_2$ or $t_1 \neq t_2$ for terms t_1 and t_2. You know nothing about these functions, other than the fact that if $x = y$, then $f(x) = f(y)$.

   Design a process that uses polynomial time to decide whether a conjunction of atoms can be satisfied by some actual function and some values. (Hint: Try using a graph. Hint 2: A vertex for each subexpression, edges for equality.)

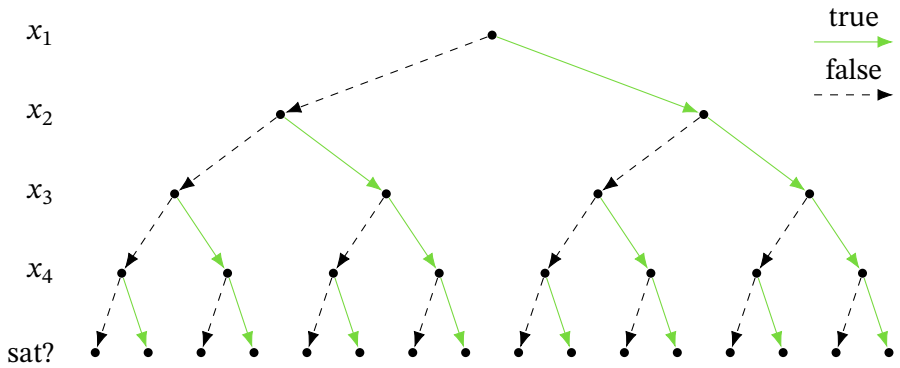
   As an extra challenge, if you are more familiar with programming, describe a data structure that makes these calculuations exceptionally fast.

5. (Optional, requires Python programming) Explore SMT solving in Z3. Z3 has built-in primitives that allow you to do SMT solving, such as integer types. Directly translate the SSA division code from the main lesson into Z3, and have it find a counterexample. Visit `https://github.com/glenn-sun/mcsp25-sat` for starter code.

3 The CDCL algorithm

Finally, we will design a SAT solver. Given a CNF with n variables, how can you determine whether or not it is satisfiable? The most simple algorithm is to just try every assignment of variables, but there are 2^n assignments to check. For unsatisfiable formulas, this algorithm will always take exponential time!

One way to visualize this brute force algorithm is with a binary tree. Arbitrarily order the variables $x_1, x_2, \ldots, x_n$. Starting at the top of the tree, we decide whether x_1 is true or false, then move on to x_2, and so on until we've decided every variable. Once we have picked all the variables, we plug them in to the formula and check if we guessed the right values. If not, we try some new guesses.



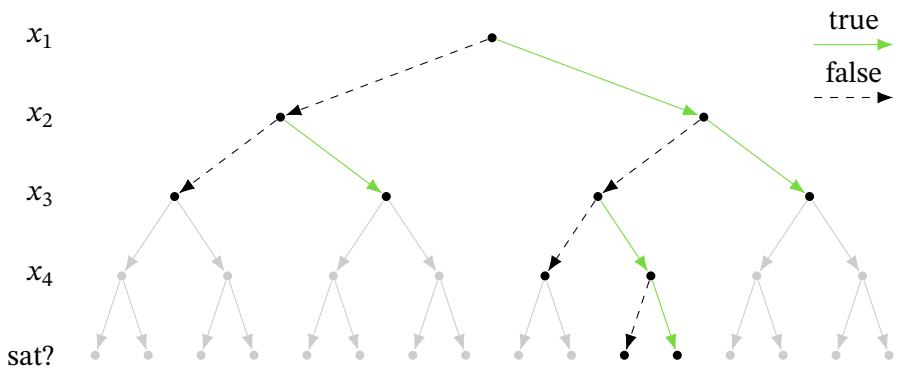
This view of the tree search algorithm immediately suggests a natural way to make it faster. Instead of waiting until all variables have been decided to check whether or not the assignment works, check every time you make an assignment. If you can say that a partial assignment already doesn't work, you have eliminated an entire subtree!

Example 3.1 (backtracking tree search)**.** Suppose our CNF has the clauses

$$(x_1 \vee x_2) \qquad (\neg x_2) \qquad (x_2 \vee x_3) \qquad (x_2 \vee \neg x_3 \vee x_4) \qquad (\neg x_3 \vee \neg x_4).$$

Suppose that by default, we pick false before true. That means we start by assigning x_1 false and x_2 false. This already contradicts the first clause!

Next, we try setting x_2 true. This contradicts $\neg x_2$. So we go back up, having exhausted all options for x_2, and set x_1 true. By default, we first try x_2 false and x_3 false. That contradicts $(x_2 \vee x_3)$, and we continue. In the end, we find that the the formula is unsatisfiable, and we did not have to check any of the assignments in gray below!



For the purpose of clarity, here is an explicit description of the backtracking tree search algorithm.

Definition 3.2 (backtracking tree search)**.** Given a list of clauses Δ, backtracking tree search does the following:

1. Let $U = \emptyset$ be the set of assignments.
2. Traverse the tree as follows:
   (a) Check if U makes any of the clauses in Δ false.
   (b) If so,
      i. If there is still an unexplored branch, revert U to the latest decision for which we have not yet explored both the true and false branches. Recursively traverse the subtree from the unexplored branch.
      ii. Otherwise, return "unsatisfiable".
   (c) Otherwise,
      i. If there is an unset variable, pick one, set it to true/false, and add it to U. Recursively traverse the subtree from this branch.
      ii. Otherwise, return "satisfiable" with U as the satisfying assignment. ⌟

Note that although our examples used a fixed variable order $x_1, \dots, x_n$, it would be equally sound algorithm to change the order in each branch. Thus, the formal description above allows you to change the variable order. Variable order will be flexible in all of our algorithms, and heuristics for good variable orders are still a subject of active research.

***

Backtracking tree search is a huge improvement over naive brute force, especially if we have many short clauses that are easy to contradict. However, there are still some ways that we can make it even faster.

Example 3.3 (unit propagation)**.** Consider the same clauses as the previous example:

$$(x_1 \vee x_2) \qquad (\neg x_2) \qquad (x_2 \vee x_3) \qquad (x_2 \vee \neg x_3 \vee x_4) \qquad (\neg x_3 \vee \neg x_4).$$

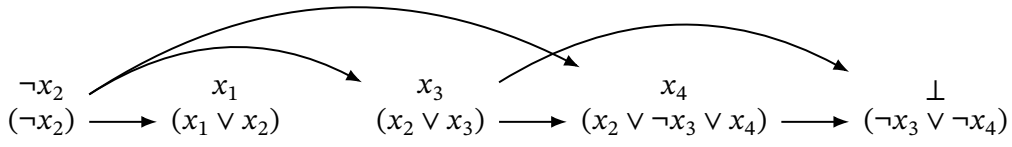
We see that there is a *unit* (a clause with only one variable), namely $(\neg x_2)$. This means in any satisfying solution, x_2 must be false.

If x_2 is false, then the clause $(x_1 \vee x_2)$ can be simplified to just (x_1). This is now also a unit, so we learn that x_1 is true. The clause $(x_2 \vee x_3)$ also lets us learn that x_3 is true. The clause $(x_2 \vee \neg x_3 \vee x_4)$ doesn't reduce to a unit, but can be reduced to $(\neg x_3 \vee x_4)$. This finishes the unit propagation of $(\neg x_2)$.

Now, (x_1) is also a unit, so we run the same process with x_1. But no other clauses have x_1, so nothing happens. We also had (x_3) as a unit. Now, the clause $(\neg x_3 \vee x_4)$ reduces to (x_4), and the clause $(\neg x_3 \vee \neg x_4)$ reduces to $(\neg x_4)$.

Thus, we learn that x_4 must be both true and false, which is impossible. We have proven that these clauses are unsatisfiable with just unit propagation! The process is summarized in the following *implication graph*, where we have listed the units, the

original clause used to deduce each one, and arrows representing which units were used in each deduction. The symbol $\bot$ denotes a contradiction.



⌟

Definition 3.4 (unit propagation)**.** Given a list of clauses Δ, each having at least 2 literals, and a set of units U, unit propagation is the following procedure:

1. Mark all everything in U as unprocessed.
2. While there is an unprocessed $(\ell) \in U$ (either $\ell = x$ or $\ell = \neg x$ for some x),
   (a) For all $C \in \Delta$, if ℓ appears in C, remove C from Δ.
   (b) For all $C \in \Delta$, if $\neg\ell$ appears in C, remove $\neg\ell$ from C. If this results in C being a unit, say $C = (\ell')$,
      i. If $(\neg\ell') \in U$, add $\bot$ to U and return U, breaking the loop.
      ii. Otherwise, add (ℓ') to U, mark it unprocessed, and remove (ℓ') from Δ.
   (c) After processing all $C \in \Delta$, mark ℓ fully processed.
3. Return the updated U.

⌟

In general, unit propagation will not be enough to learn the value of every variable. Combining unit propagation with backtracking tree search is exactly the DPLL (Davis–Putnam–Logemann–Loveland) algorithm for SAT solving.

Definition 3.5 (DPLL)**.** Given a list of clauses Δ, the DPLL algorithm[2] is a modified version of the backtracking tree search algorithm. In particular,

1. Let U be the set of units in Δ and remove them from Δ.
2. Traverse the tree as follows:
   (a) Run unit propagation on Δ and U, and update U with the result.
   (b) If $\bot \in U$,
      i. If there is still an unexplored branch, revert U to the latest decision for which we have not yet explored both the true and false branches, deleting from U both decisions and the units learned by propagating them. Recursively traverse the subtree from the unexplored branch.
      ii. Otherwise, return "unsatisfiable".
   (c) Otherwise,
      i. If there is an unset variable, pick one, set it to true/false, and add it to U. Recursively traverse the subtree from this branch.
      ii. Otherwise, return "satisfiable" with U as the satisfying assignment.

⌟

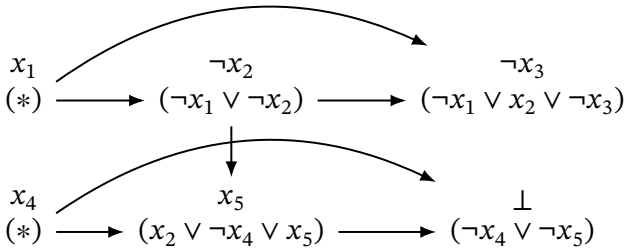
[2]To be fully historically accurate, DPLL included one other heuristic procedure called *pure literal elimination*. This was the observation that if x appears in some clauses but never $\neg x$, then it is safe to set x true, and similarly for false. However, this happens rather infrequently, and implementing this rule in modern solvers is generally believed to be not worth the overhead costs of finding such x.

Example 3.6 (DPLL)**.** Suppose you have the following clauses:

$$(\neg x_1 \vee \neg x_2) \qquad (\neg x_1 \vee x_2 \vee \neg x_3) \qquad (x_2 \vee \neg x_4 \vee x_5) \qquad (\neg x_4 \vee \neg x_5)$$

Since there are no units, DPLL makes an arbitrary decision, say x_1 true. Unit propagation learns $\neg x_2$ and then $\neg x_3$. At this point, all remaining reduced equations have at least 2 variables remaining. Thus, we make an arbitrary decision, say x_4 true, and run unit propagation again. We learn x_5, and then a contradiction.

The following implication graph summarizes the above computation. Nodes marked $(*)$ are decisions. Note that by convention, we always show unit propagations going to the right, and every time we make a new decision, we go down.



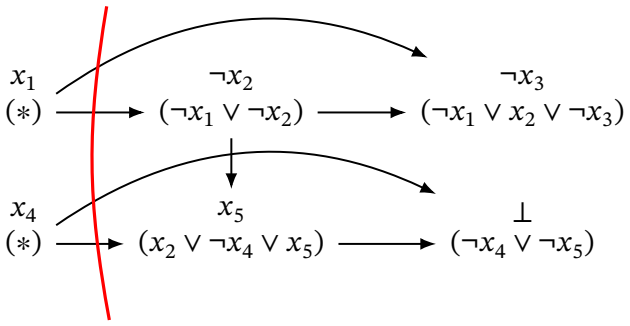
At this point, we backtrack to the last decision, which was x_4. We try x_4 being false, i.e. $\neg x_4$, and run unit propagation again. But $\neg x_4$ actually satisfies the last two clauses, and so now all clauses are satisfied. We return that the instance is satisfiable with x_1 true, x_2 false, x_3 false, x_4 false, and x_5 anything. ⌟

DPLL was the gold standard algorithm for SAT solving for over 30 years, from the early 1960s to the late 1990s. It is much, much faster than brute force, and small instances of SAT were already feasible to solve on the computers available at that time. Unit propagation is fast, and long chains of unit propagation are typical, because most SAT encodings of problems involve short clauses, often clauses of length 2. In the problems, you will further optimize unit propagation to be even faster.

***

Despite the effectiveness of DPLL, SAT solvers did not reach their full potential until the invention of *conflict driven clause learning* (CDCL) in the late 1990s. The main idea of CDCL is to generalize the idea of backtracking.

Before defining CDCL, let us analyze how decisions are made in DPLL with a slightly different framing. Consider the implication graph from Example 3.6 (DPLL).



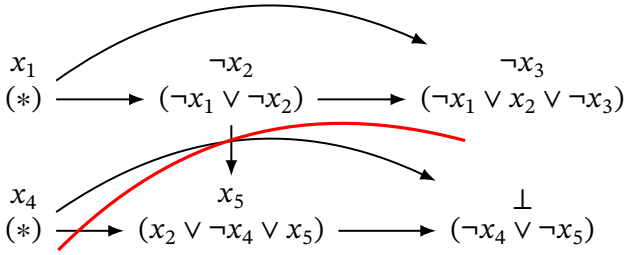
In red, the above picture highlights a *cut* that separates the decisions x_1 and x_4 from the contradiction $\bot$. The meaning of this cut is: if we know x_1 and x_4, then by unit propagation we deduce a contradiction. In other words, $(\neg x_1 \vee \neg x_4)$ is true. This is called the *conflict clause* associated with the cut. Formally, we have the following.

Definition 3.7 (conflict clause)**.** Given an implication graph that ends in $\bot$, a *cut* is a subset of the nodes that contains $\bot$ but none of the decisions. The *conflict clause* corresponding to the cut is

$$\bigvee \{\neg \ell \mid (\ell, k) \text{ is an edge across the cut}\}. \qquad \lrcorner$$

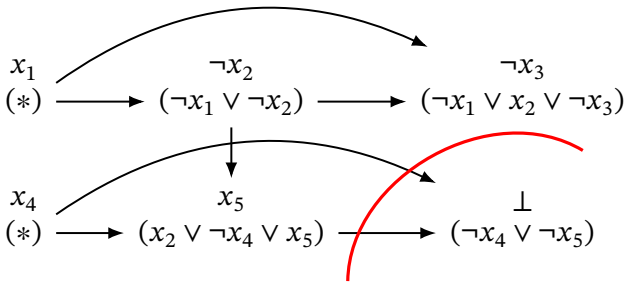
The act of backtracking is then equivalent to *learning* the clause formed by the negation of the assignments, which is a conflict clause. After all, if we add $(\neg x_1 \vee \neg x_4)$ to our set of clauses, unit propagation will derive $\neg x_4$ from x_1, which has the same effect as deciding $\neg x_4$.

CDCL generalizes DPLL by allowing us to learn conflict clauses corresponding to other cuts. For example, we will see that the following cut, which leads to the clause $(x_2 \vee \neg x_4)$, is a better clause to learn.



Both of $(\neg x_1 \vee \neg x_4)$ and $(x_2 \vee \neg x_4)$ have the same immediate effect: they allow unit propagation to deduce $\neg x_4$ after only having decided x_1. However, $(x_2 \vee \neg x_4)$ has the potential to be more useful later on. Anytime x_1 is true, allowing us to learn $\neg x_4$ using $(\neg x_1 \vee \neg x_4)$, unit propagation also allows us to learn $\neg x_4$ using $(x_2 \vee \neg x_4)$. But if eventually we are exploring a situation where x_1 is false, unit propagation cannot use $(\neg x_1 \vee \neg x_4)$ to learn anything, while it has a chance to use $(x_2 \vee \neg x_4)$ if $\neg x_2$ is learned through other means.

Let us take a look at another possible conflict clause. The following cut is drawn very close to the contradiction $\bot$. Does this cut produce a good conflict clause?



The conflict clause is $(\neg x_4 \vee \neg x_5)$. In fact, this clause is useless—we already know it! And furthermore, even if it wasn't something we already knew, it wouldn't help us because the key use of $(\neg x_1 \vee \neg x_4)$ and $(x_2 \vee \neg x_4)$ was that they allowed to us to continue unit propagation after reverting the decision of x_4. In that sense, those clauses captured the information that x_4 was a wrong decision, and $(\neg x_4 \vee \neg x_5)$ does not.

Definition 3.8 (level, asserting clause)**.** The *level* of a unit is the number of decisions made before it, including itself. (Implication graphs are typically drawn with one level on each row.) A conflict clause is *asserting* if it permits further unit propagation from the previous level. ⌟

Definition 3.9 (CDCL)**.** Given a set of clauses Δ, the CDCL algorithm does the following:

1. Let U be the set of units in Δ and remove them from Δ.
2. Do the following in a loop:
   (a) Run unit propagation on Δ and U, and update U with the result.
   (b) If $\bot \in U$,
      i. If U still contains a decision, pick an asserting clause C to add to Δ, and revert U to the lowest level that can continue unit propagation using C.
      ii. Otherwise, return "unsatisfiable".
   (c) Otherwise,
      i. If there is an unset variable, pick one, set it to true/false, and add it to U.
      ii. Otherwise, return "satisfiable" with U as the satisfying assignment. ⌟

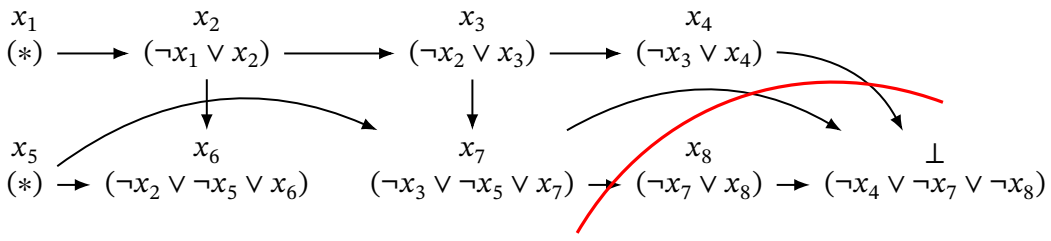
Note that we have not specified exactly how to pick an asserting clause in step 2(b).i. This is just like picking a decision—people rely on heuristics to pick them. We saw that one choice of asserting clause recovers exactly DPLL, but that other kinds of clauses may be better.

***

Though innovation is ongoing, most people today recognize the *1-unique implication point* (1UIP) asserting clause procedure and the *variable state independent decaying sum* (VSIDS) decision procedure to be the canonical heuristics, up to some small variations. We will briefly discuss 1UIP, which is the more interesting one.

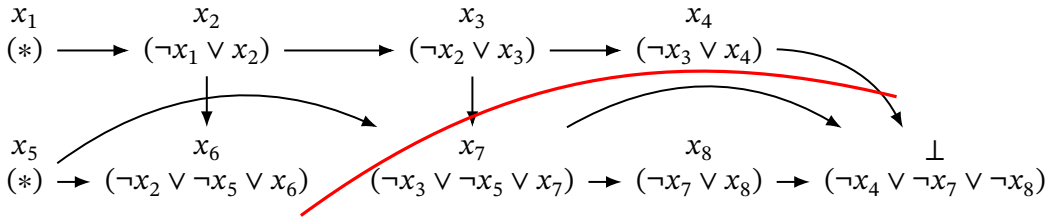
Definition 3.10 (1UIP)**.** In an implication graph, a unit on the last level is a *unique implication point* (UIP) if all paths from the last decision to $\bot$ must pass through it. The 1UIP clause is the conflict clause associated with the 1UIP cut, which takes all vertices between the last UIP and $\bot$, excluding the last UIP itself. ⌟

Example 3.11 (1UIP)**.** The 1UIP cut is drawn in the implication graph below.



In particular, the two UIPs above are the decision x_5 and the propagated unit x_7. Since x_7 is closer to $\bot$, it is the one we take for the 1UIP cut, producing the clause $(\neg x_4 \vee \neg x_7)$. In the next iteration of CDCL, the last level of the implication graph is deleted, and $\neg x_7$ is unit propagated in the first level.

For the sake of example, suppose x_5 was the actually the UIP we wanted to use. (Some people actually do this, and it is usually called the UIP-2 cut.) Then the cut is drawn below.



Note that because there is no path from x_6 to $\bot$, it is not included in the cut. The conflict clause associated with this cut is $(\neg x_3 \vee \neg x_4 \vee \neg x_5)$. ⌟

Modern CDCL solvers employ a variety of additional techniques and heuristics to improve their performance. Here are some examples that appear to be beneficial:

- Restarting often: Every so often, forget all the units U that you've decided so far and restart. This is not as drastic as it seems, because you still remember all the new clauses you learned. Intuitively, this prevents the solver from getting stuck in a particularly difficult region of the search tree.
- Deleting clauses: The learned clauses are actually kept separately from the original clauses and occasionally randomly deleted. This prevents the solver from getting bogged down from the large memory requirements of learning many clauses. (DPLL does not learn, so it does not have this memory problem.)

Improving CDCL-based solvers with new or better-tuned heuristics such as these remains an active area of research.

Practice

Do these problems to reinforce the main concepts from the lesson.

1. Fix a variable order and use the decision strategy of always deciding the first unset variable to be true. For any set of clauses Δ, explain why DPLL makes at most as many decisions as backtracking tree search.
2. Prove that a conflict clause is asserting if and only if it has exactly 1 variable belonging to the last level of the implication graph. Conclude that a 1UIP clause must be an asserting clause.
3. Run CDCL on the following set of clauses. Use 1UIP to pick asserting clauses, and when making decisions, set the alphabetically smallest unset variable set to true. Draw each implication graph.

$$(\neg a \vee \neg b \vee c) \qquad (a \vee b) \qquad (\neg a \vee \neg c \vee d) \qquad (c \vee d)$$

$$(a \vee \neg c \vee d) \qquad (\neg c \vee \neg d) \qquad (\neg a \vee c \vee \neg d) \qquad (a \vee c \vee \neg d)$$

Extensions

Do these problems if you are interested in additional content.

4. Let n be the number of variables. As defined in the lesson, unit propagation requires looping through all of Δ to process a single unit (ℓ), which is highly inefficient when ℓ does not appear in most clauses. Since one run of unit propagation processes up to n units, the number of clauses considered is proportional to $n|\Delta|$. Devise a method to use some preprocessing in order to reduce this overhead. If k is the average length of the clauses, your method should allow unit propagation to only look at approximately $k|\Delta|$ clauses.

4 The resolution proof system

We have seen that CDCL-based SAT solvers are extremely powerful, letting us solve sudoku puzzles instantly and formally verify codebases. However, SAT is NP-complete, so assuming P $\neq$ NP, there must exist some problems that CDCL fails to solve efficiently. Nevertheless, finding specific problems that make CDCL fail is a challenging task.

The area of theoretical computer science that answers this question is *proof complexity*. Take a run of CDCL that ended in "unsatisfiable", and record the state of the solver throughout this run. We can verify that each deduction made by the solver was valid, so this running history can be thought of as a formal proof in some proof system that the formula is unsatisfiable. Therefore, if we can show that there are no short proofs of the formula's unsatisfiability in CDCL's proof system, the solver *must* take a long time to even just write down the proof.

Definition 4.1 (resolution)**.** The *resolution* proof system has clauses as its lines, and only one rule for deduction, called the resolution rule:

$$\frac{A \vee x \qquad B \vee \neg x}{A \vee B}.$$

(The reasons are above the line and the deduction below the line.) ⌟

Example 4.2 (resolution)**.** From the following clauses, prove $\bot$.

$$(x \vee y) \qquad (\neg x \vee \neg y) \qquad (\neg x \vee y) \qquad (\neg y \vee \neg z) \qquad (x \vee \neg y \vee z)$$ ⌟

Proof. There are many possible proofs, here is one.

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg y \vee \neg z \qquad x \vee \neg y \vee z}{x \vee \neg y} \qquad x \vee y}{x} \qquad \dfrac{\neg x \vee y \qquad \neg x \vee \neg y}{\neg x}}{\bot}$$ □

Notice how the above resolution proof has a tree-like structure. When this happens, we call it a *tree-like resolution proof*. It is not generally a requirement of the proof system, and in fact, non–tree-like proofs can be exponentially shorter than tree-like proofs, but this fact is fairly involved to prove. See "Near Optimal separation of Tree-like and General Resolution" by Ben-Sasson, Impagliazzo, and Widgerson for details.

However, non–tree-like proofs cannot be written using the above syntax. The more general way to write a resolution proof is by using a table, where every non-axiom line is tagged with two previous lines as the reason, allowing us to reuse lines. Below is the proof table for the previous example.

1.	$x \vee y$	axiom
2.	$\neg x \vee \neg y$	axiom
3.	$\neg x \vee y$	axiom
4.	$\neg y \vee \neg z$	axiom
5.	$x \vee \neg y \vee z$	axiom
6.	$x \vee \neg y$	4, 5
7.	x	1, 6
8.	$\neg x$	2, 3
9.	$\bot$	7, 8

Proof complexity still cares about tree-like resolution, though, because tree-like resolution is exactly as powerful as backtracking tree search.

Theorem 4.3 (backtracking tree search and tree-like resolution)**.** *Let Δ be an unsatisfiable set of clauses.*

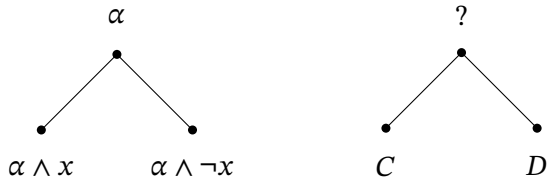
1. *Run backtracking tree search on Δ. Then there is a tree-like resolution proof of $\bot$ from Δ using at most as many lines as the number of decisions made.*
2. *If there is a tree-like resolution proof of $\bot$ from Δ, then there exists a choice of variable order for backtracking tree search that makes at most as many decisions as the number of lines in the proof.* ⌟

Proof. The correspondence is, at every point in the tree:

- The partial assignment at a node in the search tree falsifies the corresponding clause in the resolution proof tree.
- The variable branched on in the search tree is the variable resolved on in the proof tree.

To prove part (1) of the theorem, we build the proof inductively from the leaves. Because Δ is unsatisfiable, every leaf in the search tree falsifies some clause in Δ. Take any such clause to be the corresponding node in the resolution proof.

Then, at internal nodes in the search tree, suppose α is the current assignment and we branched on x. We have the following names, where $\alpha \wedge x$ falsifies C and $\alpha \wedge \neg x$ falsifies D. (The proof tree is drawn upside-down, to match the search tree.)



Consider two cases. Either α itself falsifies C, or it does not (but $\alpha \wedge x$ falsifies C). In the first case, delete the entire D subtree from the proof, and write C for the question mark. In the second case, $\neg x$ must appear in C, i.e. $C = C' \vee \neg x$, and α falsifies C'. We get the same two cases for D, and the nontrivial case is when $D = D' \vee x$. Then α falsifies $C' \vee D'$, which is the clause that fills in the question mark, and is an application of the resolution rule. At the root of the tree, the partial assignment is empty, which can only falsify the empty clause $\bot$, so this is a resolution proof of $\bot$.

To prove part (2) of the theorem, we read the resolution proof tree upside down. At the base case, which is now the root, the empty assignment certainly falsifies $\bot$. Suppose we are at a clause $C' \vee D'$ derived from $C = C' \vee \neg x$ and $D = D' \vee x$. Then in the search tree, with current assignment α, we choose to branch on x. When we decide x (positively), certainly $\alpha \wedge x$ falsifies $C = C' \vee \neg x$. Similarly, if we decide $\neg x$, then $\alpha \wedge \neg x$ falsifies $D = D' \vee x$. If we are at a clause in Δ, by induction we know that this clause is falsified by the current partial assignment. That means that if backtracking tree search reaches this point, it will will not explore further and instead backtrack. Note that it is possible for backtracking tree search to terminate this branch earlier, so this is an upper bound on the search tree. □

Corollary 4.4 (completeness of Resolution)**.** *Resolution is a* refutation complete *proof system: if Δ is a set of unsatisfiable clauses, then there is resolution proof of $\bot$ from Δ.* ⌟

Proof. Run backtracking tree search and take the proof from the correspondence above. The fact that backtracking tree search always terminates proves that there is a finite length resolution proof. □

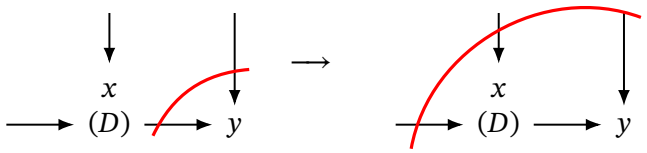
The faster algorithms of DPLL and CDCL also produce resolution proofs. The two main differences are unit propagation and clause learning. Recalling that DPLL is equivalent to CDCL with a particular choice of clause learning, the following suffices.

Proposition 4.5 (clause learning and resolution)**.** *Consider an implication graph with units U with $\bot \in U$, and let $S \subseteq U$ be a cut containing $\bot$. Let C be the conflict clause associated with S. Then there is a* linear *resolution proof of C from Δ using $|S|$ lines: not only tree-like, but consists of resolving an axiom with the previous result at each step.* ⌟

Proof. Note that because C contains all predecessors across the cut, for all $x \in S$, if there is an edge from y to x, then either $y \in S$ or y appears (negated) in C. Thus, we construct the proof inductively from $\bot$. We will consider many cuts throughout the process, and maintain that the clause associated with each of these cuts admits a linear resolution proof from Δ.

For the base case, the clause associated with the cut $\{\bot\}$ is simply the reason for $\bot$. Add the predecessors that have edges going into $\bot$ to a queue. By above, these predecessors all belong to S or appear in C.

For the inductive step, we expand the cut by 1 vertex. If the queue contains only units appearing in C, we are done by induction. Otherwise, pick $x \in S$ from the queue. Recalling that S does not contain decisions, let D be the reason for S, and so $D = D' \vee x$. Consider the picture below.



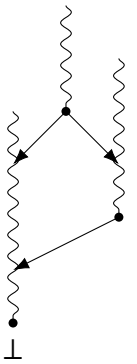
Because x was not in the previous cut and there was an edge (x, y) where y was in the previous cut, $\neg x$ appeared in the previous conflict clause. Resolving that clause with D, we get exactly the conflict clause of new cut. □

Despite being able to interpret DPLL and CDCL as both learning a particular clause, the increased generality of CDCL has a strong proof-theoretic effect: it can find general resolution proofs, whereas DPLL can only find tree-like ones. In other words, the following corollary gives a theoretical reason for why CDCL is a faster algorithm than DPLL. Any problem that has a exponential size tree-like resolution proof but a polynomial size general resolution proof must take exponential time with DPLL, but could potentially take less time with CDCL.

Corollary 4.6 (DPLL, CDCL, and resolution)**.** *Let Δ be an unsatisfiable set of clauses.*

1. *Run DPLL on Δ. Then there is a tree-like resolution proof of $\bot$ from Δ using the same number of lines as the number of decisions and unit propagations made.*
2. *Run CDCL on Δ. Then there is a resolution proof of $\bot$ from Δ using the same number of lines as the number of decisions and unit propagations made.* ⌟

Proof. We will prove (2) first. In CDCL, all of the progress is measured by clause learning, not branching. Therefore, the only kind of resolution being used is linear resolution. However, the learned clauses are remembered for future reuse. When we reuse them, we do *not* spend time rederiving them, and this reuse corresponds to general resolution. The skeleton of a CDCL-derived resolution proof would look something as follows, where the squiggly lines indicate linear resolution (resolving with clauses from Δ not shown), dots indicate conflict clauses, and arrows point from reasons to deductions.



For (1), recall that DPLL is equivalent to CDCL, when always choosing to learn clauses that negate the decisions. These learned clauses makes the above picture into a tree. To prove this, it suffices to show that every learned clause is only used once. After reaching a contradiction, call the last decision x and the previous decisions α, then DPLL will learn $\neg\alpha \vee \neg x$. This clause will be used once to learn $\neg x$ immediately, and then never again because it will always be true: in the future, either we are searching deeper in the tree with $\neg x$ still in our set of known units, or we have jumped up and changed another variable in α. □

For backtracking tree search, we also proved the converse of this corollary, that the existence of proofs can be translated back into appropriate choices of variable order that give a fast execution for the algorithm. For DPLL, this fact is again true, and you will will prove this in the problems. For CDCL, this fact is roughly true, but with a polynomial-size blowup in the number of decisions and also requiring CDCL to make the right choices for random restarts. See "On the power of clause-learning SAT solvers as resolution engines" by Pipatsrisawat and Darwiche for details.

Practice

Do these problems to reinforce the main concepts from the lesson.

1. Prove that if there exists a tree-like resolution proof of $\bot$ from Δ, then there is a choice of variable order so that DPLL makes at most as many decisions as there are lines in the proof.
2. Prove that if there exists a linear resolution proof of C from Δ, then unit propagation starting with Δ and $\neg C$ (this is a set of units) produces $\bot$.

Extensions

There are no extensions problems for this lesson.

5 Lower bounds for CDCL

The goal of the resolution proof system is to prove that certain problems must take exponential time for CDCL to prove, no matter what variable order or clause learning heuristics it uses. This is actually a considerably difficult problem, and the first problem that mathematicians managed to show difficult was the pigeonhole principle (PHP), due to Armin Haken. We will follow a presentation given by Paul Beame.

The pigeonhole principle has variables $[i \in j]$ (this is the name of the variable, read "pigeon i in hole j") with n pigeons and $n-1$ holes, and two kinds of clauses:

- Every pigeon goes to at least one hole:

  $$[i \in 1] \vee \cdots \vee [i \in n-1]$$

  for all $1 \leq i \leq n$.
- Every hole has at most one pigeon:

  $$\neg[i_1 \in j] \vee \neg[i_2 \in j]$$

  for all $1 \leq j \leq n-1$ and all $i_1, i_2 \in \binom{[n]}{2}$.

The CNF is unsatisfiable, and proving PHP means proving $\bot$ from these clauses.

Note that one feature that we typically think of as part of PHP are missing from this definition: that every pigeon goes to exactly one hole (we don't clone pigeons). If you like, you can think of us as having this constraint, but note that adding this constraint does not affect the satisfiability of the formula, so we will be okay with omitting it. Another unnecessary but allowable constraint that we can add is that every hole gets exactly one pigeon (there are no empty holes).

Haken not only showed that every resolution proof of PHP (on n pigeons and $n-1$ holes) must have many clauses, in fact he showed that it must have many *large* clauses. The proof is by contradiction, and it has the following two parts.

1. If there are not many large clauses, then we can remove them and generate a new proof of PHP on n' pigeons and $n'-1$ holes, for n' just slightly smaller than n.
2. Every resolution proof of the pigeonhole principle must have a large clause in the middle somewhere. In particular, the long clause will be long for n', but because $n' \approx n$, it will be long for n too.

Actually, that is a small lie, because we will not be defining the size of a clause the way that you expect, as the number of variables in the clause. Instead, we will modify the clause first.

Definition 5.1 (modified clause in PHP)**.** Let C be a clause. The modified version of C, denoted $M(C)$, contains only positive variables $[i \in j]$ and no negative variables $\neg[i \in j]$. It is obtained by replacing each $\neg[i \in j]$ with

$$[1 \in j] \vee \cdots \vee [i-1 \in j] \vee [i+1 \in j] \vee \ldots [n \in j],$$

i.e. an OR of $[i' \in j]$ for all $i' \neq i$. ⌟

Note that the modified clause has exactly the same meaning as the original clause if we assert that there are no empty holes. We are not asserting this by default, but because our argument will only end up using assignments where there are no empty holes, this will be a useful modification to make. It ensures that all variables appear positively, and when we spoke about "large" clauses before, we really meant that $M(C)$ has many variables, not C.

Proposition 5.2 (restricting PHP)**.** *Let $C_1, \dots, C_N$ be a resolution proof of PHP with n pigeons and $n-1$ holes (these are the lines of the table), and suppose we put pigeon 1 into hole 1, setting $[1 \in 1]$ true, $[1 \in j]$ false for all $j \neq 1$, and $[i \in 1]$ false for all $i \neq 1$.*

Update $C_1, \dots, C_N$ to reflect knowledge of these units (i.e. deleting the clauses that are satisfied, removing literals that are falsified). The resulting sequence of clauses, up to a change of variable naming and potentially making some clauses shorter, is a resolution proof of PHP with $n-1$ pigeons and $n-2$ holes. ⌟

Proof. You will prove this in the problems. Intuitively, we have gotten rid of all the variables that involve pigeon 1 or hole 1, so the problem is really about the remaining $n-1$ pigeons and $n-2$ holes. We write the same proof and the settings above correspond to ignoring the old variables whenever they appeared. □

One more important fact for the following lemma is that the following two operations produce the same result. Note that after applying $M(C)$ to every clause, the resulting sequence of clauses is not a resolution refutation, but we can still set the variables and simplify clauses.

- Set the variables as in the above proposition, then apply the modification $M(C)$ to every clause.
- Apply the modification $M(C)$ to every clause, then set the variables.

Lemma 5.3 (removing large clauses)**.** *Let $C_1, \dots, C_N$ be a resolution proof of PHP with n pigeons and $n-1$ holes. Suppose there are $L < 2^{n/20}$ large clauses in $M(C_1), \dots, M(C_N)$ with $n^2/10$ or more variables each. Then there exists a resolution proof of PHP with n' pigeons and $n'-1$ holes with no large clauses and $n' = cn$, where $c = (1-\log_{10/9}(2))/20 \approx 0.671$.* ⌟

Proof. With L large clauses and $n^2/10$ or more variables each, by averaging over all $n(n-1)$ variables, there is a variable that appears at least

$$\frac{Ln^2}{10n(n-1)} \geq \frac{L}{10}$$

times, call it $[i \in j]$. By the previous proposition, up to a change of variable names, we can set this variable and some related variables to get a proof of PHP with $n-1$ pigeons and $n-2$ holes. Because $[i \in j]$ is set to true with all variables in $M(C_i)$ appearing positively, at least a $L/10$ of the large clauses are now satisfied, and satisfied clauses are no longer in the proof. Thus this is a proof of PHP with $n-1$ pigeons and $n-2$ holes, using at most $9L/10$ large clauses.

Repeating this process $\log_{10/9}(L)$ times will thus remove all of the large clauses from the proof, and we end with a proof of PHP with n' pigeons and $n'-1$ holes, where

$$n' \geq n - \log_{10/9}(L) \geq n - \log_{10/9}(2^{n/20}) = n(1 - \log_{10/9}(2))/20,$$

as was to be shown. □

Lemma 5.4 (existence of large clauses)**.** *Let $C_1, \ldots, C_N$ be a resolution proof of PHP with n pigeons and $n-1$ holes. Then some $M(C_i)$ must have at least $2n^2/9$ variables.* ⌟

Proof. Recall that the modification $M(C_i)$ is a logically sound modification only when we consider cases where every hole is required to have exactly one pigeon. Let us restrict ourselves to the assignments where the holes and pigeons are paired up, except for one leftover pigeon, call such assignments full. Let

$$\text{leftovers}(C) = \{i \mid C \text{ can be falsified by a full assignment leaving pigeon } i \text{ left over}\}.$$

We note the following properties:

- $\text{leftovers}(\bot) = [n]$.
- If C is an axiom that says pigeon i goes to at least one hole, then $\text{leftovers}(C) = 1$. Namely, to falsify C, i cannot go into any hole, so it is left over. Because we are only considering full assignments, if pigeon i gets no hole, then everyone else gets a hole and is not left over.
- If C is an axiom that says pigeons i_1 and i_2 cannot share hole j, then $\text{leftovers}(C) = 0$, simply because there are no full assignments where pigeons share holes.
- If $C = C' \vee \neg x$ and $D = D' \vee x$ are resolved to create $C' \vee D'$, then $\text{leftovers}(C' \vee D') \subseteq \text{leftovers}(C) \cup \text{leftovers}(D)$. This is just because any full assignment that falsifies $C' \vee D'$ must either falsify C (if it sets x true) or D (if it sets x false).

As the number of leftover pigeons increase from the axioms to $\bot$, we claim that when about $n/2$ pigeons are leftover, the clause must be very long. We are not sure if there is a point where exactly $n/2$ pigeons are leftover, by the last observation, there is a clause C_i with $n/3 \leq \text{leftovers}(C_i) \leq 2n/3$.

We will show that if there are ℓ pigeons leftover in a clause C, then $M(C)$ has at least $n(n-\ell)$ pigeons, which completes the proof as $(n/3)(2n/3) = 2n^2/9$.

Let $i \in \text{leftovers}(C)$, specifically let α be the assignment that falsifies C and leaves i left over. Let $i' \notin \text{leftovers}(C)$ arbitrarily, and have α' be α with i and i' swapped. Because i was originally left over, this swap only changes 2 variables: if $[i' \in j]$ originally, now

$$\begin{aligned} [i' \in j] &\quad\to\quad \neg[i' \in j] \\ \neg[i \in j] &\quad\to\quad [i \in j]. \end{aligned}$$

However, because now i' is leftover and $i' \notin \text{leftovers}(C)$, the assignment α' must make C true. Thus, because $M(C)$ only contains variables positively and is equivalent to C for full assignments, it must contain the variable $[i \in j]$. Repeating this argument for all pairs of pigeons, one of which is left over and not the other, we conclude the lemma. □

Theorem 5.5 (Haken's theorem)**.** *PHP with n pigeons and $n-1$ holes takes at least $2^{n/20}$ lines to prove in resolution.* ⌟

Proof. Suppose for contradiction that there exists a proof in less than $2^{n/20}$ lines. Then there are less than $2^{n/20}$ lines C_i where $M(C_i)$ has $n^2/10$ or more variables each. By Lemma 5.3 (removing large clauses), there exists a proof of PHP with n' pigeons and

$n' - 1$ holes with no large $M(C_i)$, where $n' = cn$ with $c = (1 - \log_{10/9}(2)/20)$. But by Lemma 5.4, this proof must have an $M(C_i)$ with

$$\frac{2(n')^2}{9} \approx 0.100071n^2 > \frac{n^2}{10}$$

variables, contradiction. □

SAT solving and proof complexity remain active fields of research, with major innovations happening very often from all sides, including practical applications, algorithmic advancements, and lower bounds. For further reading, the following sources contain good information:

- *Handbook of Satisfiability*, Armin Biere et al.
- *Proof Complexity*, Jan Krajíček
- *CSE 599S Course Notes*, Paul Beame

Practice

Do these problems to reinforce the main concepts from the lesson.

1. Prove Proposition 5.2 (restricting PHP).
2. Identify every place in the proof where properties of the resolution proof system were used. State these properties, so that the result can be generalized to any proof system satisfying your list of properties.

Extensions

There are no extensions problems for this lesson.