Lecture notes for: “The science of brute force”

Lindsey Kuper

September 30, 2019

These are lecture notes to accompany Marijn J. H. Heule and Oliver Kullmann, “The science of brute force” (CACM 2017). See http://composition.al/CSE290Q-2019-09/readings.html for the full collection of notes.

Agenda

- Motivating problem #1: Pythagorean triples problem
- Why we care
- Motivating problem #2: Boolean Schur triples problem
- SAT solving terminology
- Solver internals: paradigms of SAT solving, CDCL, BCP
- Illustrating CDCL and BCP: back to Boolean Schur Triples
- Who watches the watchmen?

Welcome to CSE290Q

So, I could have just gone right to the Kroening and Strichman textbook, but I decided to have us start by reading this “Science of Brute Force” article because I thought that might be a little more fun and less dry than the textbook — the article has some colorful language (like “A mathematician using ‘brute force’ is a kind of barbaric monster, is she not?”) Maybe you found it fun, maybe you didn’t.

This article also introduces a few concepts that we'll be spending time in the next few weeks talking about in much more detail. And it does that by setting up a math problem and then discussing, at least at a high level, the techniques by which a SAT solver would solve that problem. I think this is instructive and so we'll actually walk through that math problem today.

We won't really be talking about the last third or so of the article:

- Details of "solutions-preserving modulo x"
- The Ramsey theory stuff
- Philosophical musings about "alien truths"

But I might have time to talk about that stuff some more on Wednesday.

By the way, this course is mostly about SMT solving, right? Why did I have us start out reading an article about SAT? Well, SAT solving is at the heart of SMT solving, as we'll see. Or you could also say that SAT solving *is* SMT solving; it's just the degenerate case where the theory, "T", is the boring default one. More about that in the coming weeks.

Motivating problem #1: Pythagorean triples problem

Consider all partitions of the set of natural numbers into finitely many parts. (At least one part has to be infinite, then.)

The question: does at least one part contain a Pythagorean triple, that is, numbers a, b, c such that $a^2 + b^2 = c^2$?

For instance, say we partition the natural numbers into just two parts: odd numbers and even numbers.

It turns out that the square of an odd number is always odd (since $(2k +$

$1)(2k+1) = 4k^2 + 4k + 1$). But adding two odd numbers will produce an even number. So there can't be a Pythagorean triple in the odd partition. But the even partition does have one: $6^2 + 8^2 = 10^2$.

Simplified problem

Well, that was one way to split up the set of natural numbers into two parts. What about other ways of splitting into two parts? Is there always going to be a Pythagorean triple in at least one of the parts?

They show that the answer is yes. This is the "Boolean Pythagorean triples" problem. (If it were "three parts" that would be the "three-valued Pythagorean triples" problem.)

It suffices to show the existence of a *subset* of the natural numbers such that any partition of that subset into two parts has one part containing a Pythagorean triple.

Consider only subsets $\{1, ..., n\}$ for some n. It turns out (via a brute force proof using a SAT solver) that the smallest such subset for which the property is true is $\{1, 2, 3, ..., 7825\}$.

How many ways are there to partition the set $\{1, 2, 3, ..., 7825\}$ into two parts?

- 1 can go in set 1 or set 2 (2 ways)
- 2 can go in set 1 or set 2 (4 ways)
- …
- 7825 can go in set 1 or set 2 (2^{7825} ways)

So there are 2^{7825} ways to partition the set $\{1, 2, 3, ..., 7825\}$ into two parts. How big a number is 2^{7825}? "Way too big." At least, way too big to

deal with without a computer.

No non-computational existence proof for this problem is known — that is, the *only* way we know to solve this problem is with something like a SAT solver.

How far are we from being able to solve the full Pythagorean triples problem?

We did the case of two parts; how about three? We can show that the set $\{1, 2, ..., 10^7\}$ can be partitioned into three parts such that *no part* contains a Pythagorean triple. So, if there is some n such that every 3-partitioning of n has a part containing a Pythagorean triple, we know that $n > 10^7$.

How many ways are there to partition the set $\{1, 2, ..., 10^7\}$ into three parts?

- 1 can go in set 1, set 2, or set 3 (3 ways)
- 2 can go in set 1, set 2, or set 3 (9 ways)
- …
- 10^7 can go in set 1, set 2, or set 3 (3^{10^7} ways)

So there are 3^{10^7} ways to partition the set $\{1, 2, ..., 10^7\}$ into three parts. This number is so big that we might not ever solve the three-valued Pythagorean triples problem.

And the problem we originally started with, when it was any finite number of parts — well, good luck.

Why do we care?

Why are we playing these games? Because solving these problems is analogous to the problems we face in software verification and automated bug-finding.

- Bug-finding is finding counterexamples. “Finding a bug in a large hardware system is essentially the same as finding a partition avoiding all Pythagorean triples.” (If you don’t find one, it doesn’t mean there isn’t one, but if you do find one, it is definitely there.)
- Verification is proving that there is no counterexample. “Proving correctness [of a large hardware system]…is similar to proving that each partition must contain some Pythagorean triple.”

The focus of most of this course is on using solvers for automated software bug-finding and verification. So this article is honestly doing something sort of quaint by using them for mathematical proofs, instead. Like, “More secure and reliable software? Who cares about that? Let’s just do math problems for their own sake!” But I still think the article is very helpful for helping us understand how one goes about encoding a problem as a SAT problem, and then how the solver goes about solving it.

(BTW, SAT is perhaps even more important for *hardware* verification than it is for software verification. The article claims, “SAT has revolutionized hardware verification.” “Bounded model checking using satisfiability solving” by Edmund Clarke et al. (2001) is the citation for that claim. I’m actually not sure how much that paper has to do with hardware, but it’s a good one to look at if you want to know about the relationship between model checking and SAT solving.)

BTW, what is model checking?

This is a diversion, but maybe worth knowing about because it overlaps with studying SAT and SMT, and I myself didn't know this until recently.

So, a piece of hardware or software can be modeled as a finite-state machine. If you've taken a theory of computation class then you know about finite state machines. (The Clarke et al. paper talks about "transition systems", which are a generalization of finite state machines.) Model checking means exhaustively exploring the state space of a system to check whether a certain property is true. So for instance, suppose you want to make sure that some property is always true. Well, you exhaustively go through every state that the system can be in, and you check that it's true in every state. One particularly effective way of doing this is using satisfiability solvers.

Motivating problem #2: Boolean Schur triples problem

So at this point they switch problems to a different problem, which is related, but small enough to actually work out ourselves.

So, the "Boolean Schur triples problem": Does there exist a red/blue coloring of the numbers $1, ..., n$ such that there is *no monochromatic solution* of $a + b = c$ with $a < b < c \leq n$?

(A "Schur triple" is just any three numbers (a, b, c) where $a + b = c$. The "Boolean" in the name of the problem here just refers to the fact that there are two color options, red and blue.)

So this time we're dealing with all natural numbers, not just squares of natural numbers.

For $n = 8$, such a coloring exists. Anyone remember what it was from the paper?

To spill the beans: color 1, 2, 4, 8 red, and color 3, 5, 6, 7 blue.

So how would you have solved this in a naive brute-force way?

Well, there are two ways to color each number, right? Red and blue. So, with 8 numbers, that's 2^8 possibilities. How many is that? 256. And then for each possibility (say, 1 red and 2-8 all blue), you have to go through and consider each subset of 3 numbers a, b, c, where $a + b = c$, and make sure that a, b, and c weren't all colored the same color.

Or maybe you would pick out all subsets of three numbers that *are* the same color, and then make sure that two of them don't add up to the remaining one. I dunno.

This is not necessarily the most efficient way to solve the problem, though. What if you could avoid having to go through all 256 colorings?

In the paper they show how for $n = 9$, there's no way to do the coloring, and they figure this out without having to consider all $2^9 = 512$ possible red/blue colorings. In fact, they say that with "brute reasoning" (which I think of as meaning "brute force, but employed in a clever way"), only six (partial) red/blue colorings even need to be checked. We'll look at a SAT encoding of this problem and come back to why you only need to check six colorings in a bit.

SAT solving terminology

So let's finally talk about SAT solving, and to do that we have to define some terms.

- A “Boolean variable” can be assigned either true or false.
- A “literal” is either a Boolean variable or its negation. So if x is a Boolean variable, then x is a literal, and so is $\neg x$.
- A literal x is true if the Boolean variable x is assigned to true; a literal $\neg x$ is true if the Boolean variable x is assigned to false. (Sometimes we abuse terminology by talking about “assigning to a literal”, when it’s only variables that can be assigned to.)
- A “clause” is a disjunction of literals, that is literals connected by an “or”. So if x and y are Boolean variables, then $x \vee y$ is a clause, and so is $x \vee \neg y \vee z$, and so on. A single literal by itself is also a clause.
- A “SAT formula” is a conjunction of clauses. So $(x \vee y) \wedge (\neg z \vee q)$ is a SAT formula. This is called conjunctive normal form (CNF): a formula is in CNF if it is a conjunction of clauses (or a single clause).
- An “assignment” is a mapping of each Boolean variable in a SAT formula to either true or false. The reading says: A clause is satisfied by an assignment if that assignment makes at least one of its literals true. A formula is satisfied by an assignment if all of its clauses are satisfied. (BTW, an assignment is also called an “interpretation” of the variables.)
- A SAT formula is “satisfiable” if there exists *some* satisfying assignment for it, and “unsatisfiable” if there does not exist any satisfying assignment.
- “solving” a SAT formula is determining its satisfiability or unsatisfiability.
- A “SAT solver” is a computer program that solves SAT formulas.

Q: So, for instance, is $(x \vee \neg y) \wedge (\neg x \vee y)$ satisfiable?

A: Yes. What are the satisfying assignments? x = true, y = true is one; the other is x = false, y = false.

Q: How computationally hard is SAT?

A: Satisfiability of general Boolean formulas is a famously NP-complete problem. In fact, it was the first problem proven to be NP-complete, in 1971. There's no known polynomial-time algorithm for solving the SAT problem, and the question of whether SAT has a polynomial-time algorithm is equivalent to the P versus NP problem.

But the silver lining of NP-completeness (and the paper makes sure to point this out) is that any problem in NP can be efficiently converted to SAT. This is why SAT solvers are so damn useful, because a whole lot of interesting computational problems (especially software and hardware verification) can be converted to SAT, and then you can throw a SAT solver at them.

Practically though; speaking, modern SAT solvers often do a good job on really big formulas. You might have heard people talk about the so-called "SAT revolution" or "SMT revolution". In the early 90s, solvers could handle formulas with thousands of clauses; today, solvers can handle formulas with millions of clauses.

Also, just as an aside: if the formula is a conjunction of clauses and the clauses are each a disjunction of exactly two literals, then that's known as a 2SAT formula, and these *are* solvable in polynomial time! (Look up 2-satisfiability on Wikipedia to learn lots more about this.) However, this isn't helpful unless you have clauses that are each a disjunction of exactly two literals.

Solver internals: CDCL, BCP

The article mentions three "paradigms of SAT solving", and I like the article's description of these, so I'm just going to reproduce it here.

- "local search": incomplete; tries to find a solution via local modifications to total assignments

What does incomplete mean? It means if you give it a formula, it can only find satisfying assignments to the formula, but it cannot tell you for sure if the formula is unsatisfiable.

So it's "sound", meaning that if it tells you a satisfying assignment, then you know that that really is a satisfying assignment. But it's incomplete, meaning that if it doesn't give you a satisfying assignment, that doesn't mean that there isn't one! So the answers you get back from local search are either "yes, here is a satisfying assignment" or "shrug".

- "look-ahead": complete; recursively splits the problem as cleverly as possible into subproblems
- "conflict-driven clause learning" (CDCL): complete; tries to assign variables to find a satisfying assignment, and if that fails (the normal case), then the failure is transformed into a clause called a "conflict clause" which is added to the formula you're trying to satisfy. (This is what "clause learning" means.)

These two paradigms are both sound and complete. If it doesn't give you a satisfying assignment, you know that there is not one. The answers you get back are either "yes, here is a satisfying assignment" or "no, there is no satisfying assignment".

CDCL

OK, so what is CDCL? BTW, if you've heard of DPLL, CDCL is the successor to it. DPLL is the grandaddy of SAT solving algorithms and has been around since the '60s; CDCL is the modern successor to DPLL. (There'll be more on this in a future reading assignment.)

OK, so, CDCL at a very high level (we'll go into more detail on the CDCL algorithm in the coming days):

Three phases: simplify, decide, learn.

- simplify: simplify the formula you have and update the assignment with any new inferences you learn. (We'll see an example of this in a second.)
- decide: pick an unassigned variable and assign it true or false, using some heuristic. The heuristics that you use here are called, unsurprisingly, "decision heuristics".

Q: What's a heuristic, by the way?

A: A heuristic is an approach that's "fast enough" and "good enough" that we use when we don't necessarily know the optimal thing to do. Heuristics usually have some pathological case that goes wrong.

The article mentions a couple of kinds of these decision heuristics, but we're not really going to talk about that today at all; we'll read about it in the coming weeks. At a high level, though, the idea is that you have to somehow choose which variable to assign to, and you have to choose what to assign to it. Making a good decision early on can make solving the whole problem faster.

OK, so you iterate these two phases until one of two things happen:

- you have a satisfying assignment (yay, you're done).
- at least one clause has been falsified — you made a mistake somewhere.

In the "you made a mistake" case, you've created what's called a "conflict". You need to learn from your mistake and make sure you don't make it again in the future. So, you go to the "learn" phase:

- learn: add a new clause called a "conflict clause" to the formula. The point of the "conflict clause" is to prevent the decide phase from making that particular mistake again, i.e., prevent the solver from trying that particular assignment again. We'll see an example of what a conflict clause looks like in a little while. Then you backtrack to before you made the mistake, and you go back up to the simplify/decide cycle and keep going.

Q: OK, so what about unsatisfiability? What if your formula is unsatisfiable — when do you find that out?

A: During the learn phase. It turns out that if the new "conflict clause" you're forced to add is what's called the "empty clause", which is a clause that can't be satisfied, that means that you have an unsatisfiable formula (yay, you're done).

That's a very high-level overview of CDCL. It's OK if it doesn't make sense now; we'll talk about this a lot more when we read chapter 2 of the Kroening and Strichman book.

Unanswered question: in the decide phase, what heuristics do you use to pick an unassigned variable and assign it? We'll come back to this later in the course.

BCP

What this article calls “unit clause propagation” (UCP) is also known as “unit propagation” or “Boolean constraint propagation” (BCP). I’ll use the term BCP because I think it’s more common.

BCP is a part of the CDCL algorithm. This is again something we’ll study more later, but let’s talk about it really quick:

Recall that a clause is a disjunction of literals. So the way to satisfy a clause is to make at least one of its literals true.

A clause is “unit” under a partial assignment that makes all but one of its literals false, and leaves one unassigned. The only way to satisfy a unit clause under that partial assignment is to assign the necessary value to make the remaining literal true.

So, if you have a partial assignment and a formula, and the formula has unit clauses, then you go through the unit clauses and add to the assignment by assigning variables to satisfy the unit clauses. This is what’s known as Boolean constraint propagation. There are two possibilities:

- you’ve satisfied all the unit clauses
- you’ve created a conflict, because two unit clauses contradict each other (one has a literal x, and the other has $\neg x$)

If you created a conflict, you then have to deal with the conflict. We’ll talk about this more soon, but at a high level, one of two things will happen:

- you have to backtrack and do something different
- you figure out that the formula is unsatisfiable

Q: If a formula has a unary clause to begin with, is it a unit clause?

A: Yep. So you should do BCP right away, even before you make any decisions.

Illustrating CDCL and BCP: back to Boolean Schur triples

Let's see an example of CDCL and BCP in action.

So, back to the Boolean Schur triple problem: Does there exist a red/blue coloring of the numbers 1, …, n such that there is *no monochromatic solution* of a + b = c with $a < b < c \leq n$?

We're thinking about the case where n = 9.

First step is to encode this problem as a SAT problem.

They encode it this way:

Enumerate all the ways where a + b = c where a, b, and c are drawn from the set 1, …, 9, and a < b < c <= n.

There turn out to be 16 of these:

- 1 + 2 = 3
- 1 + 3 = 4
- 1 + 4 = 5
- 2 + 3 = 5
- 1 + 5 = 6
- 2 + 4 = 6
- 1 + 6 = 7
- 2 + 5 = 7
- 3 + 4 = 7
- 1 + 7 = 8

- 2 + 6 = 8
- 3 + 5 = 8
- 1 + 8 = 9
- 2 + 7 = 9
- 3 + 6 = 9
- 4 + 5 = 9

So our triples are (1, 2, 3), (1, 3, 4), … (4, 5, 9). And we want to show that there is a way to color each of the numbers 1, …, 9 either red and blue in such a way that all of these triples have some red and some blue in them. In other words, at least one number in each triple has to be red, and at least one number in each triple has to be blue.

We have 9 Boolean variables, x_1, x_2, …, x_9, representing each of the numbers 1 through 9, respectively. We'll encode red as true and blue as false.

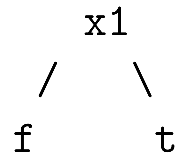
And then we just write out two clauses for each of the 16 triples. One clause will say that at least one of the numbers has to be red, so, for the triple (1, 2, 3), for instance, we have the clause $(x_1 \vee x_2 \vee x_3)$. And one will say that at least one of the numbers has to be blue, so, again for (1, 2, 3), we have the clause $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$.

And the formula is just the conjunction of all of these clauses. So if this formula is satisfiable, then we have a red/blue coloring that satisfies the property, and if it isn't, then we don't.

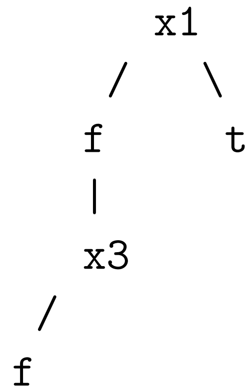
There are, as we said before, $2^9 = 512$ possible assignments of true and false to the variables. But we said that you actually only have to check six of them.

So what do they do? Well, let's look at x_1 first. It has two possibilities, false

and true.



In the world where x_1 is false, then let's think about x_3. Say that it's false.



OK, so x_1 is false and x_3 is false. Well, now we can do BCP!

Well, it says that x_2 has to be true, because x_1, x_2, and x_3 are in a clause together. And it says that x_4 has to be true, because x_1, x_3, and x_4 are in a clause together.

So x_2 and x_4 have to be true, but then they're in a triple with x_6, so x_6 has to be false.

But x_1 and x_6 are in a triple with x_7, so then x_7 has to be true. And x_3 and x_6 are in a triple with x_9, so then x_9 has to be true.

But now we have a conflict because all of x_2, x_7, and x_9 are now true and they're in a triple together. In other words, the part of our formula that says $(\neg x_2 \vee \neg x_7 \vee \neg x_9)$ is now unsatisfiable.

So we've just ruled out all assignments where both x_1 and x_3 are false, regardless of the other settings. (That's a lot of assignments.)

What do we do now? Well, we know that one of x_1 and x_3 have to be true,

right? So now we've learned our first conflict clause! The conflict clause will be $(x_1 \vee x_3)$. Then we go back to before we set x_3 to true, and try something else.

So now consider the case where x_1 is false but x_3 is true. No conflicts yet, so then we can go a little deeper. Consider x_5. Let's say it's false.

What does that tell us? Once again, we can do BCP.

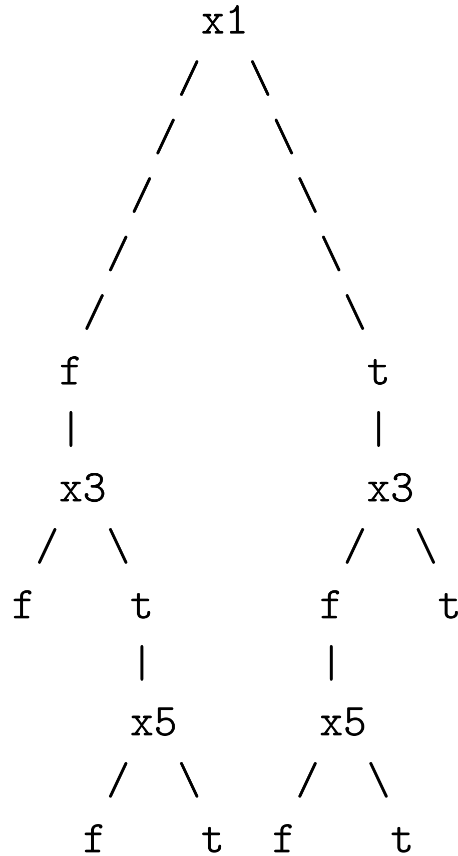
It says that x_4 has to be true, because x_1, x_4, and x_5 are in a clause together. And it says that x_6 has to be true, because x_1, x_5, and x_6 are in a clause together.

So then x_2 has to be false, because x_2, x_4, and x_6 are in a triple together. and since x_5 is false, and x_2, x_5, and x_7 are in a triple together, x_7 has to be true. But now x_3 is true, x_4 is true, and x_7 is true, and they're in a triple together.

So we've just ruled out all assignments where x_1 is false, x_3 is true, and x_5 is false. We already know that one of x_1 and x_3 has to be true. So now we know that x_1 and x_5 can't be false at the same time, because if x_1 is false, then x_3 has to be true.

So we've learned another conflict clause: $(x_1 \vee x_5)$.

And so on. It turns out we can keep applying similar reasoning to this, and we end up only having to consider six partial assignments to show that the formula is unsatisfiable.



So let's back up a second. That worked out pretty well for us, but the whole reason why it worked out so well was because we looked at x_1 first, and then looked at x_3, and so on. If we had looked at the variables in a different order, we might not have been able to rule out huge chunks of the search space so fast.

Q: So how did I know to look at x_1 first, and then x_3, and so on?

A: Using a simple heuristic — the "Maximum Occurrences in clauses of Minimal Size" heuristic, known affectionately as "MOMS".

When we start, all the clauses are of the same size, and x_1 occurs the most. So we start with x_1. Then, after simplification, x_3 occurs the most in clauses of minimal size, and so on. It turns out that this simple heuristic is enough to cut the number of partial assignments that need to be checked down to 6.

They point out that for the original problem that they talked about, the Pythagorean Triples problem, this simple heuristic isn't enough, and in fact the authors of this paper had to design a special, fancy heuristic just for solving that problem. They cite their own paper about that.

Who watches the watchmen?

Do you trust a SAT solver that claims a problem is unsatisfiable?

If a solver says a formula is satisfiable, that's one thing. The solver will provide a satisfying assignment, and it's easy to check that the assignment actually does satisfy the formula.

But, if a solver says a formula is *unsatisfiable*, like in the Boolean Schur triples problem instance that we just talked about, how do you know it's correct? Especially if it's a solver using some special, fancy heuristic, how do you know you can trust it?

Well, you could formally verify the SAT solver itself. But that's really hard to do. And most people are going to want to use their fast SAT solver of choice, not the not-necessarily-state-of-the-art verified one.

The other idea is to have the solver produce a "certificate" of an unsatisfiability claim, or a *proof of unsatisfiability*. It then needs to be possible to quickly and automatically *validate* the proof, that is, make sure that it actually does certify unsatisfiability.

What does such a proof of unsatisfiability look like? The article discusses one technique for creating them. The idea is this: to determine unsatisfiability of the formula, we had to deduce a whole bunch of these conflict clauses, right? So, we should be able to make sure that the solver did the right thing

by making sure that each conflict clause was deduced correctly. This is called a *clausal proof* of unsatisfiability.

Adding a new clause to the forumla you're trying to determine the satisfiability or unsatisfiability of is a risky thing, right?

Q: What property do you want to have be true when you add a clause?

A: Well, you want to make sure that you don't *change* the satisfiability of the formula by adding a clause. In other words, you want the addition of a clause to be "solutions-preserving": any satisfying assignments that there were for the previous formula should also satisfied by the new formula with the added clause.

It turns out that this is actually pretty easy to check, though. If you have a formula F and you want to know that a new clause C is solutions-preserving, just take the partial assignment that makes all literals in C false. Then do BCP on F with that assignment. If you get a conflict, then you know that the clause is solutions-preserving, since you now know that it's not possible to falsify C and also satisfy F.

So, to check a proof of unsatisfiability, we just need a list of all the clauses that were learned and added, and then we use BCP to check the proof. It's surprisingly straightforward!

We don't have time to go into more detail about this, but the citation for this if you want to know more is "Verification of Proofs of Unsatisfiability for CNF Formulas" by Goldberg and Novikov (2003). The paper is only six pages long.

There's more stuff in the article about what it means to be solutions-preserving modulo a particular variable, but we don't have time to get into that.