Lecture notes for: Introduction to SAT solving, part 1

Agenda

- Motivating examples: probems encodable as SAT problems
- Implementing SAT solvers
    - BCP recap
    - An elementary-school SAT solver (no guessing)

Motivating examples: problems encodable as SAT problems

Q: Questions left over from last time?

There was one question last time that I want to try to address. So we talked about how we are for the most part only going to be interested in quantifier-free theories.

And Will asked, "But you really need quantifiers to be able to talk about a lot of interesting things, don't you? Why can we get away without them?"

I didn't have a great answer offhand. All I could think of is "Well, in this course we're interested in decision procedures that can actually run on a real computer, so we need our theories to be decidable, and for that I think we mostly need to avoid quantifiers. And the obvious success of SMT solving with quantifier-free theories demonstrates its usefulness." That answer is fine as far as it goes, but it didn't really answer the question. So: why *can* we do so much with quantifier-free theories?

I guess one way to answer that question is to say, well, we know that just solving plain ol' SAT (without quantifiers allowed in the formulas) is NP-complete, therefore any NP problem can be encoded as a SAT problem, and so we already know that you don't actually need quantifiers to express (most?) computationally interesting problems. But your ability to exploit that fact depends on your ability to come up with a clever encoding in a quantifier-free theory. (And *that*, the machine can't do for you!)

But there's yet another point to make, which is that when we ask a question of an SMT solver about a formula, the questions we're asking *do* implicitly contain quantifiers. If you're asking if the formula is satisfiable, then you're asking if there exists a satisfying assignment. And if you're asking if the formula is valid, the you're asking if, for all assignments, the formula is true. So there's an implicit quantification happening.

So instead of thinking of most theories as "first-order logics minus the quantifiers", it might make sense to think of them as "first-order logics where you can quantify over assignments, and you have to do it around the outside of the whole formula".

But enough about SMT; today we're going back to talking about SAT! And before we jump back into CDCL, maybe it'll be useful to start out by looking at a problem that can be encoded as SAT.

So, say we're the FCC and we're assigning frequencies to radio stations. We have n radio stations, $s_1, ...s_n$, and we have k transmission frequencies, where $k < n$. So some stations need to share a frequency, but we want them to not be too close to each other. For now we're not going to consider *relative* distance, and we're just going to say, okay, some pairs of stations are above a threshold for being too close, and other pairs are not. So we know all the pairs of stations that are too close. Say, maybe stations

3 and 4 are too close to each other to share a frequency. We have a set of all such pairs.

By the way, the book chose to frame this problem as radio station frequency allocation, but the problem is very similar to the problem of register allocation in a compiler. Say you have to compile a program to run on some hardware. The program has some number of variables; the hardware has some number of registers, which might be fewer than the number of variables. So you need to figure out which variables conflict with each other. There's a notion of variables being "live" during some part of a program's run, and you need to first figure out which variables have overlapping live ranges. That's analogous to a pair of radio stations being too close. Then you need to make sure not to assign them to the same register.

It's not quite the same problem, though, because in the case of the radio stations, if you don't have enough transmission frequencies, then the problem is just unsatisfiable.

Q: But what does the compiler do if it doesn't have enough registers?

A: It just allocates things in memory instead of in registers. So register allocation is really more of an optimization problem rather than a decision problem. If you don't have enough registers, your compiler usually isn't gonna be like, "Sorry! UNSAT!"

Q: Actually, this is an interesting question. Under what conditions would you actually *want* the compiler to say "Sorry, UNSAT"?

I think you might want that if you were compiling code to run on an embedded system that had some extremely limited amount of memory.

I asked around about this on Twitter this morning, about whether there are compilers that ensure that compiled programs don't run out of memory at runtime, and I got a lot of interesting replies. I'll send a Canvas announcement pointing to the Twitter thread in case any of you want to peruse it.

Anyway, it turns out that it's really straightforward to represent the input to this problem as a graph, and then the problem becomes how to color the graph. In the case of the radio stations, we say that every station is a node in the graph, and for every two stations that are too close together to share a frequency, we draw an edge between them.

Likewise for the register allocation, we say that every variable is a node in the graph, and for every two variables that have overlapping live ranges, we draw an edge between them.

Then the problem becomes to color the nodes, where the number of colors that we have is the number of slots that we have to allocate things into. So, frequencies in the case of radio stations, registers in the case of program variables. And the idea is to come up with a coloring so that adjacent nodes are assigned different colors.

This is the graph colorability problem: does such a coloring exist? But we can also describe it as a SAT problem.

The setup is: Define a set of Boolean variables x_{ij} where i ranges from 1 to n and j ranges from 1 to k. The idea is that variable x_ij is set to true iff radio station i is assigned the frequency j.

Q: So, what constraints do we have that affect what our satisfying assignment can be?

A: First, every station gets at least one frequency. So how do we encode that?

Suppose there are 4 stations and 3 frequencies. Then the formula for expressing that each station gets at least one frequency would be:

$(x_{11} \vee x_{12} \vee x_{13}) \wedge (x_{21} \vee x_{22} \vee x_{33}) \wedge (x_{31} \vee x_{32} \vee x_{33}) \wedge (x_{41} \vee x_{42} \vee x_{43})$

In general, for any number of n stations and k frequencies, the formula is a conjunction of clauses where each clause is the disjunction of $x_{i1}, ..., x_{ik}$.

So that's making sure that every station gets at least one.

Secondly, we have to make sure each station gets at *most* one. This is a little bit harder to encode!

But let's try to do it. Again, to make it concrete, let's still say there are 4 stations and 3 frequencies.

The encoding of this one is kind of hairy. But, in general, if a station is assigned to a frequency you want to make sure that it isn't assigned to any other frequency. So if x_{11} is true, then that means station 1 is assigned to frequency 1, so you then need to make sure that station 1 isn't assigned to frequencies 2 or 3.

$$((x_{11} \implies (\neg x_{12} \wedge \neg x_{13})) \wedge (x_{12} \implies \neg x_{13}))$$
$$\wedge$$
$$((x_{21} \implies (\neg x_{22} \wedge \neg x_{23})) \wedge (x_{22} \implies \neg x_{23}))$$
$$\wedge$$
$$((x_{31} \implies (\neg x_{32} \wedge \neg x_{33})) \wedge (x_{32} \implies \neg x_{33}))$$
$$\wedge$$
$$((x_{41} \implies (\neg x_{42} \wedge \neg x_{43})) \wedge (x_{42} \implies \neg x_{43}))$$

Thirdly, you need the set of pairs of stations that are close to each other. So for every pair (i, j) in that set,

add an implication $x_{it} \implies \neg x_{jt}$ for every $t < k$.

So, if stations 3 and 4 happen to be close, we have

$$(x_{31} \implies \neg x_{41}) \land (x_{32} \implies \neg x_{42}) \land (x_{33} \implies \neg x_{43})$$

Q: Do you also need the other way around? That is, do you also need:

$$(x_{41} \implies \neg x_{31}) \land (x_{42} \implies \neg x_{32}) \land (x_{43} \implies \neg x_{33})$$

A: It turns out you don't, because $p \implies q$ desugars to $q \lor \neg p$. So the above becomes:

$$(\neg x_{31} \lor \neg x_{41}) \land (\neg x_{32} \lor \neg x_{42}) \land (\neg x_{33} \lor \neg x_{43})$$

So the SAT encoding of the problem consists of the conjunction of all these pieces.

The book doesn't actually discuss how the solver operates on this particular problem; it merely gives it as an example of an encoding of a real-world problem as a SAT problem to help motivate the need for SAT solving.

By the way, I thought it would be interesting to try to encode the problem of assigning papers to presenters in this class as a SAT problem (or maybe an SMT problem). But I didn't manage to come up with an encoding in the limited time I had to think about it, although I'm certain that it's possible. Might be something fun to try on your own if you want to get some practice encoding problems as SAT.

Implementing SAT solvers

Q: Questions about the example we just went over?

OK, let's talk about implementing SAT solvers some more!

We're going to assume from now on that the input to the solver is a Boolean formula in CNF. We can do this with no loss of generality because you can efficiently convert any SAT formula to CNF. We didn't go over it in class, but there's a standard algorithm for it called the Tseytin transformation. (Not Satan, as in the Prince of Darkness, but Tseytin.) The details of that conversion are in Chapter 1 of the book.

So, given one of these CNF formulas, the goal of the solver is to find a satisfying assignment or determine that there is no satisfying assignment, and it does this by building up an assignment one variable at a time and then backtracking if it discovers it made a mistake.

We're going to be talking about conflict-driven clause learning SAT solvers. As we talked about last Monday, you can think of there being three phases to the CDCL algorithm:

- simplify: simplify the formula you have, update the current assignment with anything you can infer (BCP happens here)
- decide: make a decision about a variable and its value, using some heuristic to choose which variable and which value to try next (we'll talk about heuristics laser)
- learn: add a new clause to the formula summarizing information learned from a conflict, and backtrack to where you were before you made the decision that caused the conflict

with the "simplify" and "decide" phases iterating until you encounter a conflict, in which case you go to the "learn" phase.

So the book has a pictorial overview of the CDCL algorithm, and we can label parts of this figure with those three phases of simplify, decide, and learn.

The “simplify” phase is BCP. The “decide” phase is “Decide”, obviously. And the “learn” phase is the “AnalyzeConflict” and “Backtrack”

BCP recap

So, as CDCL solving proceeds, there are various states that a clause in a formula can be in:

- **satisfied** if at least one of its literals is satisfied by the current partial assignment. (Recall that a clause is a disjunction.)
- **conflicting** if all its literals are assigned and it is not satisfied by the current partial assignment.
- **unit** if *all but one* of its literals are assigned and it is not satisfied. These are the ones that we said are hanging on the edge of being conflicting.

It’s the unit clauses that are particularly interesting, because they allow us to extend the assignment. In fact, they *require* us to extend the assignment! For instance, if we have the assignment $\{x_1 = \text{true}, x_2 = \text{false}, x_4 = \text{true}\}$, and we have the clause $(\neg x_1 \vee \neg x_4 \vee x_3)$, then we know we need to extend the assignment with $x_3 = \text{true}$.

This is called the **unit clause rule** – whenever you have a unit clause, you must extend the current assignment with the binding that will satisfy that clause.

Recall how the “Science of Brute Force” paper spoke of *unit clause propagation*, or UCP. That’s what we’re going to call Boolean constraint propagation, or BCP. But it’s the same thing as that paper’s UCP. And BCP is simply *repeated application of the unit clause rule*.

Q: What's the name for a unit clause that *causes* a particular literal to become assigned?

A: We call it the **antecedent clause** of that literal. So in the example I just gave, $(\neg x_1 \vee \neg x_4 \vee x_3)$ is the antecedent clause of x_3.

As CDCL runs, decisions get made about the values of variables, and we can think of these decisions as forming a binary tree. So for instance, if we decide the value of x_1 first, then that decision is made at depth 1 in the tree. Therefore the decision about $x_1 was made at **decision level** 1. Any other decisions that *have* to get made as a result of that decision also happen at level 1.

One important thing to keep in mind is that decisions are *guesses*, whereas when we add bindings to the assignment as a result of BCP, those aren't guesses. When you run CDCL, it might actually be the case that you don't have to make any guesses! Maybe you have a formula where it's already the case that you have a unit clause (because you had a unary clause in the formula, so you have a literal that you know has to be true). You don't want to make guesses unless it's necessary.

So this suggests that you should also do BCP right away when you start CDCL, even before you make any decisions! And, indeed, that's what this picture looks like.

An elementary-school SAT solver (no guessing)

When I was a kid in elementary school, from time to time we had to do these logic puzzles that would be something along the lines of, there are five people, and they all have favorite colors, and they all have favorite sports, and they all have favorite animals, and your job would be to fill in some table assigning people to their favorite colors or sports or animals or what have you. And you'd get a list of constraints, which would be something like "the person whose favorite sport is basketball hates the color red", and "the person whose favorite animal is the zebra plays either soccer or volleyball" and so on. And one of these constraints would give you a place to start from that would allow you to propagate information to further constrain the search.

As I recall, these puzzles were always designed in such a way that you would never have to make guesses and then backtrack if your guess was wrong. You would instead be able to figure out the entire assignment by propagating what you already knew. In other words, all decisions would be made at decision level 0, in the initial BCP phase.

By the way, those logic puzzles were usually always designed in such a way that they *had* a solution. But it is possible that during this initial BCP phase, you find out that the formula does in fact not have a solution. So, if you implemented the elementary school logic puzzle version of a SAT solver, it would look like this:

```
// Elementary-school SAT solver
// given a formula F, returns SAT or UNSAT
CDCL(F):
    // initialize empty assignment
    A = {}
    // do BCP on the initial formula
    if BCP(F, A) = conflict
        return UNSAT
    while notAllVariablesAssigned(F, A)
        // Give up and complain to the
        // teacher that the problem is underconstrained
    return SAT
```

Of course, we're not doing the elementary school logic puzzle version of CDCL, we're doing the version of CDCL that actually has to deal with having to make guesses. What the CDCL algorithm gives you is a framework for making good guesses and then gracefully backing out of our guesses when it turns out we made bad ones.

So, something is going to have to go here in place of "give up and complain to the teacher", right?

```
// given a formula F, returns SAT or UNSAT
CDCL(F):
    // initialize empty assignment
    A = {}
    // do BCP on the initial formula
    if BCP(F, A) = conflict
        return UNSAT
    while notAllVariablesAssigned(F, A)
        // Do stuff
    return SAT
```

Next time, we'll talk about what we have to do in place of "Do stuff" there.