Lecture notes for: Introduction to SAT solving, part 2

Agenda

- Recap of last time
- Grown-up SAT solving: the full CDCL algorithm
  - Making decisions
  - Learning from our bad decisions
  - Knowing when to give up, and when to back up and try again
- Implication graphs

Recap of last time

OK! So when we finished last time, we wrote down what I called an "elementary school" SAT solving algorithm. It looked 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
    return SAT
```

So the idea is that, when we were solving one of those logic puzzles from elementary school, we could generally solve them without making any guesses. In other words, we never had to use the "Decide" part of the CDCL algorithm, which is the part that makes guesses. We just considered what we currently knew, and then propagated that knowledge to extend the assignment until all variables were assigned.

As we've discussed, BCP is just *repeated application of the unit clause rule*. Recall that the unit clause rule says that whenever you have a unit clause, you must extend the current assignment with the binding that will satisfy that clause. So in this algorithm, you do BCP and keep on extending the assignment until you either come up with a conflicting clause, in which case the formula is unsatisfiable, or until you run out of unassigned variables, in which case you're done and the formula is satisfiable. If neither of those things happen, then you give up and complain to the teacher, "This problem is too hard!"

Grown-up SAT solving: the full CDCL algorithm

If we happen to have a formula that *can* be solved by just propagating what we know until every variable is assigned or until we come up with a conflicting clause, then the elementary school SAT solver is good enough. It would have been good enough for those elementary school puzzles, because those puzzles could be solved without guessing. But, more generally, we're going to want to solve formulas that may require some amount of guessing. So we want to do something here instead of "give up and complain".

Making decisions

Being a grown-up sometimes requires making decisions even when you don't have all the information, and so grown-up SAT solving will have to do that too.

Q: So, what do we have to do instead of give up and complain? Any suggestions?

A: Well, we're going to have to make a guess, right? We're going to have to pick a variable and say "suppose this one is true", or "suppose this one is false". And we'll have to keep track of the place where we made a decision and everything that followed from it, in case it comes to pass that we have to undo the decision later.

So now we need to use the concept of "decision level" that we introduced last time.

In the elementary-school SAT solver, we never actually had to make a guess. All variables were assigned at decision level 0. In other words, we didn't have any *decision variables*, which are variables that get their assignment as a result of a guess. But now we're going to have to start making decisions, and keeping track of where we made them. That's what the decision level is for. So we'll introduce a variable called `decisionLevel` for that.

And now, instead of giving up and complaining, we need to venture bravely into the unknown and make a decision about some variable in the formula. So, we're going to pick out an as-yet-unassigned variable, and we're going to extend the assignment with a binding for it. This will be our first decision variable.

How do we know which variable to assign to and whether to assign it true or

false? We don't. We make the choice using some heuristic. The choice of heuristic is orthogonal to the CDCL algorithm, although of course it has profound implications for the performance of the solver, and different heuristics are more appropriate for different kinds of formulas. We don't have time to get into decision heuristics now, though.

```
// given a formula F, returns SAT or UNSAT
CDCL(F):
    // initialize empty assignment
    A = {}, decisionLevel = 0
    // do BCP on the initial formula
    if BCP(F, A) = conflict
        then return UNSAT
    while notAllVariablesAssigned(F, A)
        decisionLevel++
        // add a new decision to the assignment
        // based on some heuristic
        A = A extended with DECIDE(F, A)
        // Do stuff (more to come here...)
    return SAT
```

Every time we make a decision, we increment the decision level. You can visualize this as a binary tree: if we first decide x_1, then that decision is made at decision level 1, and the next decision will be made at decision level 2, and so on. The decision level of a variable is the depth of the binary tree at which that variable was assigned.

Of course, we don't have to make a decision about every variable. For many variables, their value will be implied by BCP. We call these *implied variables*, as opposed to decision variables.

The decision level of an implied variable is the maximum of the decision levels of the other variables in its *antecedent clause*. (Recall that the antecedent clause of a variable is the unit clause that implied its value. Decision variables don't have antecedent clauses; implied variables do.)

And whenever we update an assignment with a new variable, either by doing BCP or by calling DECIDE, we keep track of the decision level of that variable, too.

Learning from our bad decisions

Once we've made a decision about a variable, there are two possibilities. Either it was a bad decision, meaning it will lead to a conflict, or it was a good decision, meaning it will lead to the formula being satisfied.

Q: How do we find out whether our decision leads to a conflict?

A: We can do BCP again! We've extended our assignment, right? So maybe that extended assignment will lead to a conflict, and we run BCP to find out.

Q: What do we do if BCP tells us that we have a conflict?

A: Well, up above, when we encountered a conflict, we just returned UNSAT right away, because we knew then that the formula was unsatisfiable. But now, we can't do that, because the conflict we encountered might not be because the formula is unsatisfiable; it might have merely been the result of us making a bad decision!

Q: How do we know which kind of conflict we have on our hands?

A: We can *analyze* the conflict! We do that by…calling ANALYZE-

CONFLICT, of course.

```
// given a formula F, returns SAT or UNSAT
CDCL(F):
    //initialize empty assignment
    A = {}, decisionLevel = 0
    // do BCP on the initial formula
    if BCP(F, A) = conflict
        return UNSAT
    while notAllVariablesAssigned(F, A)
        decisionLevel++
        // add a new decision to the assignment
        // based on some heuristic
        A = A extended with DECIDE(F, A)
        if BCP(F, A) = conflict
            // ANALYZE-CONFLICT returns a backtracking
            // level and a new conflict clause
            (bl, C) = ANALYZE-CONFLICT()
            // add newly learned conflict clause to F
            F = F extended with C
            // Do stuff (more to come here...)
    return SAT
```

ANALYZE-CONFLICT looks at the current conflicting clause in the formula and tells us two things: a *backtracking level*, which is the decision level that we need to go back to in order to undo our mistake, and a *conflict clause*, which is the *newly learned clause* that we need to add to our formula.

Let's talk about the conflict clause first. By the way, there's almost a naming collision here, between "conflict clause" and "conflicting clause". These are not the same thing!

Q: What's a conflicting clause? We talked about this last time.

A: A clause is conflicting if all its literals are assigned and it is not satisfied by the current partial assignment. It's a state that a clause can be in during the process of SAT solving, depending on what the current partial assignment is. So, the process of BCP can result in a clause becoming conflicting.

On the other hand, a *conflict clause*, which we first talked about way back for the first reading assignment, is a newly learned clause that you add to the original formula during solving. A conflict clause can be thought of as summarizing information that was already in the formula. The purpose of adding a conflict clause to a formula is to help us solve the formula faster by avoiding assignments that won't work out. A conflict clause is produced as a *result* of a conflict that arises during the process of BCP, but a conflict clause is not the same thing as a conflicting clause! This is a really unfortunate almost-collision of terminology.

By the way, in real CDCL implementations, instead of just tacking the conflict clause onto the end of the formula, we would have a *clause database* that we would add the new clause to. We would probably also have some sort of fancy efficient data structure for representing the formula. We could spend weeks on this topic alone if we had time.

OK, so that's the conflict clause, but ANALYZE-CONFLICT also returned something else: a backtracking level. The backtracking level is the decision level that we need to go back to in order to undo our mistake. Essentially, it's a number that tells you how far in the decision tree you need to back up.

The backtracking level tells us something important about the conflict we encountered: whether it was because the formula is actually unsatisfiable, or merely the result of making a bad decision. If ANALYZE-CONFLICT returns, say, decision level 2, then we need to undo any decisions made at

higher levels (and any assignments that happened as a result of those decisions), and try again. If ANALYZE-CONFLICT returns decision level 0, that means we need to go back to the very first decision made.

Knowing when to give up, and when to back up and try again

Q: What if ANALYZE-CONFLICT returns -1 as the backtracking level?

A: Then it's telling us to back up further than it's possible to back up. In other words, a backtracking level of -1 means that the conflicts we're encountering aren't the fault of our bad decisions — it's just that the formula is unsatisfiable! So, if *bl* is less than 0, we just return UNSAT.

Otherwise, if the backtracking level is 0 or higher, it means that we have a place to back up to, and we need to undo the decision that led to the conflict. To do this, we call BACKTRACK with the backtracking level and the current assignment A. BACKTRACK updates the assignment to *remove* any bindings that it added as a result of the bad decision that we want to undo. (How does it know which ones those were? Because we've been recording the decision level of every assigned variable.)

Finally, we update the current decision level to whatever that new backtracking level is, and we go back up to the top of the "while notAllVariablesAssigned(F, A)" loop and keep going from the point that we backtracked to. The process continues until we either return UNSAT, or we assign every variable and BCP doesn't discover any conflicts, in which case we fall out the bottom of the while loop and return SAT. And now we've finally written down the whole CDCL algorithm!

```
// given a formula F, returns SAT or UNSAT
CDCL(F):
    //initialize empty assignment
    A = {}, decisionLevel = 0
    // do BCP on the initial formula
    if BCP(F, A) = conflict
        return UNSAT
    while notAllVariablesAssigned(F, A)
        decisionLevel++
        // add a new decision to the assignment
        // based on some heuristic
        A = A extended with DECIDE(F, A)
        if BCP(F, A) = conflict
            // ANALYZE-CONFLICT returns a backtracking
            // level and a new conflict clause
            (bl, C) = ANALYZE-CONFLICT()
            // add conflict clause to formula
            F = F extended with C
            // if the backtracking level returned by
            // ANALYZE-CONFLICT was less than 1,
            // we know the formula is unsatisfiable
            if bl < 0
                return UNSAT
            else
                // undo our mistake; go back to the last
                // decision level before the mistake
                BACKTRACK(A, bl)
                decisionLevel = bl
    return SAT
```

By the way, this way of expressing the algorithm is a bit different what Kroening and Strichman show in chapter 2 (although it amounts to the same thing). It is closer to what appears in chapter 4 of the *Handbook of Satisfiability*. It's also a lot like the version that Emina Torlak uses in her slides. (In particular, as in Emina's slides, I explicitly show extending the formula with the learned conflict clause as part of the main CDCL algorithm, instead of making that a part of ANALYZE-CONFLICT.)

During solving, the decision level may move up and down. It gets incremented every time we make a new decision, but it may jump backward by one or more based on what ANALYZE-CONFLICT returns.

We've seen the overall structure of the algorithm, but we glossed over how DECIDE works and how ANALYZE-CONFLICT works. We don't have time to talk about either of these now, but we'll introduce the concept of an *implication graph*, which helps us understand how ANALYZE-CONFLICT can construct conflict clauses.

Implication graphs

Suppose we have the following formula:

$$(x_1 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_3 \vee x_2 \vee x_4)$$

When solving this formula with CDCL, we start by trying to do BCP, but there are no unit clauses. So we go on to picking a variable to decide. Suppose we pick x_4 and assign it false.

We write "$\neg x_4 @ 1$" to denote that x_4 was assigned false at decision level 1.

Q: Now we can do BCP again. Does this result in any new implied variables?

A: No. So we need to decide another variable. Suppose we pick x_1 and assign it false as well. So we have $\neg x_1@2$.

Q: And now we do BCP yet again. Any new implied variables this time?

A: Yes! This time we have $x_3@2$ and $x_2@2$ (denoting that x_3 and x_2, respectively, were assigned true at decision level 2). Notice that the decision level for both of these is the maximum decision level of the other variables in their antecedent clauses. The antecedent clause for x_3 was $(x_1 \vee x_3)$, and the antecedent clause for x_2 was $(\neg x_3 \vee x_2 \vee x_4)$.

There are no more unassigned variables, so we've determined that the formula is satisfiable. (That was an easy one — we didn't have any conflicts.)

We can use the notion of antecedent clauses to define a directed acyclic graph that connects assigned variables to one another. Every assigned variable is a node in the graph. The edges in the graph come from antecedent clauses: for each variable x, there is an edge from each variable in x's antecedent clause to x. (x also occurs in its own antecedent clause, but we omit the edge from x to itself.)

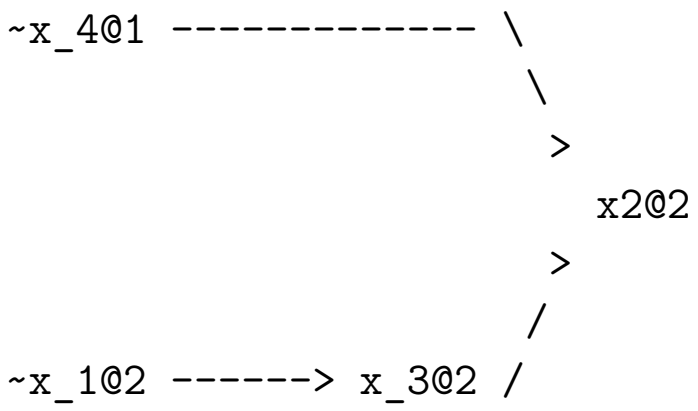
Only implied variables have antecedent clauses, so only implied variables can have incoming edges. Decision variables have no antecedent clauses, so they cannot have incoming edges, only outgoing edges. That is, they are *source* nodes in the graph.

This graph is called the **implication graph**. The convention when drawing these implication graphs is to start with decision variables the left and have directed edges go to the right.

For example, an implication graph for the series of assignments we just did would be as follows: on the left we have two decision variables, $\neg x_4@1$ and $\neg x_1@2$. There's no edge between these two; they've both decision variables, so neither has any incoming edges.

Then from BCP we get $x_3@2$. Its antecedent was $(x_1 \vee x_3)$, so we add an edge from x_1 to x_3. And we also have $x_2@2$. Its antecedent was $(\neg x_3 \vee x_2 \vee x_4)$, so it has incoming edges from both x_3 and x_4.

The implication graph looks like this:



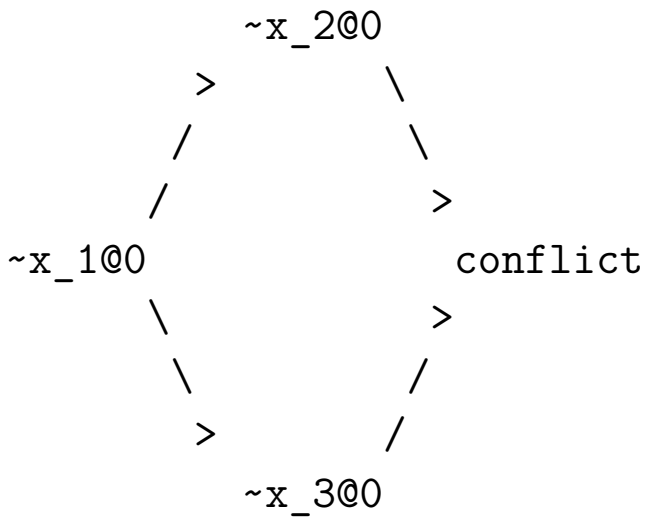
If unit propagation results in a clause being conflicting, then we add a special node called a conflict node to the implication graph, and we add edges from the nodes representing the variables of the conflicting clause to the conflict node. The conflict node is always a *sink* node in the graph.

For instance, suppose that our formula is the following:

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge \cdots$$

Suppose we begin by assigning $\neg x_1@0$. Then from BCP we get $\neg x_2@0$ and $\neg x_3@0$, making the third clause in the formula conflicting.

The implication graph looks like this:



For implication graphs that have a conflict node, the source nodes that the conflict node can be reached from represent the decision variables that played a role in the conflict. So, one way to construct a conflict clause is to take the *disjunction of the negation* of those assignments. For example, for the above simple formula, the conflict clause is the unary clause (x_1), because the source node in the implication graph set x_1 to false.

Does the unary clause (x_1) tell us anything that the original formula didn't? No. But that's the point: conflict clauses don't add new information, they just summarize existing information. Had the clause (x_1) been a part of the formula from the beginning, we never would have tried assigning $x_1 =$ false because we would have been forced to assign $x_1 =$ true during the initial BCP. By adding the conflict clause, we avoid exploring any other assignments that set $x_1 =$ false.

There are more sophisticated ways of constructing conflict clauses, but it works pretty well to just look at the implication graph, find all the literals associated with source nodes from which the conflict node can be reached, and then take the disjunction of their negations!