15-414: Bug Catching: Automated Program Verification

Lecture Notes on SAT Solvers & DPLL

Matt Fredrikson

Carnegie Mellon University

Lecture 16
March 18, 2025

1 Introduction

In this lecture we will continue our exploration of algorithmic techniques for proving formulas automatically. Such algorithms are called **decision procedures**, because given a formula in some logic they attempt to decide their validity after a finite amount of computation.

Until now, we have gradually built up from proving properties about formulas in propositional logic, to doing so for first-order dynamic logic. As we begin discussing decision procedures, we will return to propositional logic so that the techniques applied by these algorithms can be more clearly understood. Decision procedures for propositional logic are often referred to as SAT solvers, as they work by exploiting the relationship between validity and satisfiability, and directly solve the latter problem. Later on, we will see that these same techniques underpin decision procedures for richer logics, and are able to automatically prove properties about programs.

2 Review: Propositional Logic and Unit Resolution

Recall that our grammar for propositions is:

$$Q, P \quad ::= \quad p \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid \neg P \mid \top \mid \bot$$

A truth assignment M assigns either true or false to every propositional variable (or atom, as we say). This is analogous to the *state* ω in dynamic logic that assigns an integer to every variable. So, if you like, you can think of propositional theorem proving as deciding the (quantifier-free) theory of Booleans. We write $M \models P$ if the formula P

is true given the assignment M. This is defined exactly as we did in dynamic logic on these connectives with one additional clause:

$$M \models p \quad \text{iff} \quad M(p) = true$$

A convenient way to present a truth assignment is by giving a list of p if $M(p) = true$ and $\neg p$ if $M(p) = false$. In principle, this list would have to be infinite, but since every formula contains only finitely many atoms we can use such a finite representation.

$$(p \land q \to r) \land (p \to q) \to (p \to r) \tag{1}$$

We use some additional terminology to refer to formulas that evaluate to $\top$ under some or all possible interpretations.

Definition 1 (Validity and Satisfiability)**.** A formula F is called *valid* iff it is true in all interpretations, i.e. $I \models F$ for all interpretations I. Because any interpretation makes valid formulas true, we also write $\vDash F$ iff formula F is valid. A formula F is called *satisfiable* iff there is an interpretation ω in which it is true, i.e. $I \models F$. Otherwise it is called *unsatisfiable*.

Satisfiability and validity are duals of each other. That is, a formula F is valid if and only if $\neg F$ is unsatisfiable.

$$F \text{ is valid} \leftrightarrow \neg F \text{ is unsatisfiable} \tag{2}$$

Importantly, this means that we can decide whether a formula is valid by reasoning about the satisfiability of its negation. A proof of validity for F from the unsatisfiability of $\neg F$ is called a *refutation*. Most efficient decision procedures use this approach, and therefore attempt to directly prove the satisfiability of a given formula. These tools are called SAT solvers, referring to the propositional SAT problem. If a SAT solver finds no satisfying interpretation for F, then we can conclude that $\neg F$ is valid.

In Lecture 14 we discussed *resolution*, an important inference rule for propositional logic that was central to early SAT solvers and the development of *refutation-based* decision procedures. We highlighted a particular case of resolution known as *unit resolution*, summarized by the following rule:

$$\frac{p \lor C \quad \neg p}{C}$$

Today we will return to this rule, and see how it is used today in modern SAT solvers.

3 A Simple Procedure

Conceptually, SAT is not a difficult problem to solve. Each atom in the formula corresponds to a binary choice, and there are a finite number of them to deal with. Recall from the second lecture how we used truth tables to determine the validity of a formula:

1. Enumerate all possible interpretations of the atoms in F.
2. Continue evaluating all subformulas until the formula is a Boolean constant.
3. F is valid iff it is $true$ under all interpretations.

We can modify this procedure to decide satisfiability in the natural way.

1. Enumerate all possible assignments of the atoms in F.
2. Continue evaluating all subformulas until the formula is a Boolean constant.
3. F is satisfiable iff it is $true$ under at least one interpretation.

Implementing this procedure is fairly straightforward. The only part that might be tricky is enumerating the valuations, making sure that *i)* we don't miss any, and *ii)* we don't enumerate any of them more than necessary, potentially leading to nontermination.

One natural way to do this is to use recursion, letting the stack implicitly keep track of which valuations have already been tried. We will rely on two helper functions to do this, which are outlined informally below.

- `choose_atom: formula -> atom`. This function takes a formula argument and returns an arbitrary atom appearing in it.
- `subst: formula -> atom -> bool -> formula`. Takes a formula, and atom appearing in the formula, and a Boolean value, and returns a new formula with all instances of the atom replaced by the Boolean value. It also simplifies it as much as possible, attempting to reduce the formula to a constant.

The function `sat` is given below. At each recursive step, the function begins by comparing the formula to the constants `true` and `false`, as a final decision can be made immediately in either case. Otherwise, it proceeds by selecting an arbitrary atom `p` from `F`, and creating two new formulas `Ft` and `Ff` by substituting `true` and `false`, respectively, and simplifying them as much as possible so that if there are no unassigned atoms in the formula then they are reduced to the appropriate constant. `sat` then makes two recursive calls on `Ft` and `Ff`, and if either return `true` then `sat` does as well.

```
let rec sat (F:formula) : bool =
  if F = true then true
  else if F = false then false
  else begin
    let p = choose_atom(F) in
    let Ft = (subst F p true) in
    let Ff = (subst F p false) in
    sat Ft || sat Ff
  end
```

Intuitively, we can think of this approach as exhaustive case splitting. The procedure chooses an atom p, splits it into cases p and $\neg p$, and recursively applies itself to the cases. If either is satisfiable, then the original is as well. We know this will terminate because each split eliminates an atom, and there are only a finite number of atoms in a formula.

We now have a basic SAT solver. We know that SAT is a hard problem, and more precisely that it is NP-Complete, and a bit of thought about this code should convince that this solver will experience the worst-case runtime of 2^n much of the time. There is a chance that we might get lucky and conclude that the formula is satisfiable early, but certainly for unsatisfiable formulas `sat` won't terminate until it has exhausted all of the possible variable assignments. Can we be more clever than this?

Conjunctive Normal Form and Unit Resolution. We have been assuming that formulas are given in clausal form, which is a disjunction of conjunctions of literals. Consider the following CNF formula:

$$\underbrace{(p_1 \lor \neg p_3 \lor \neg p_5)}_{C_1} \land \underbrace{(\neg p_1 \lor p_2)}_{C_2} \land \underbrace{(\neg p_1 \lor \neg p_3 \lor p_4)}_{C_3} \land \underbrace{(\neg p_1 \lor \neg p_2 \lor p_3)}_{C_5} \land \underbrace{(\neg p_4 \lor \neg p_2)}_{C_6} \tag{3}$$

Suppose that `sat` begins by choosing to assign p_1 to $true$. This leaves us with:

$$\begin{aligned}
&(p_1 \lor \neg p_3 \lor \neg p_5) \land (\neg p_1 \lor p_2) \land (\neg p_1 \lor \neg p_3 \lor p_4) \land (\neg p_1 \lor \neg p_2 \lor p_3) \land (\neg p_4 \lor \neg p_2) \\
\leftrightarrow\ &(\top \lor \neg p_3 \lor \neg p_5) \land (\bot \lor p_2) \land (\bot \lor \neg p_3 \lor p_4) \land (\bot \lor \neg p_2 \lor p_3) \land (\neg p_4 \lor \neg p_2) \\
\leftrightarrow\ &\top \land p_2 \land (\neg p_3 \lor p_4) \land (\neg p_2 \lor p_3) \land (\neg p_4 \lor \neg p_2) \\
\leftrightarrow\ &p_2 \land (\neg p_3 \lor p_4) \land (\neg p_2 \lor p_3) \land (\neg p_4 \lor \neg p_2)
\end{aligned}$$

Notice the clause C_2, which was originally $\neg p_1 \lor p_2$, is now simply p_2. It is obvious that any satisfying interpretation must assign p_2 $true$, so there is really no choice to make given this formula. We say that p_2 is a *unit literal*, which simply means that it occurs in a clause with no other literals.

We can immediately set p_2 to the value that satisfies its literal, and apply equivalences to remove constants from the formula.

$$\begin{aligned}
&\top \land (\neg p_3 \lor p_4) \land (\neg \top \lor p_3) \land (\neg p_4 \lor \neg \top) \\
\leftrightarrow\ &(\neg p_3 \lor p_4) \land (\bot \lor p_3) \land (\neg p_4 \lor \bot) \\
\leftrightarrow\ &(\neg p_3 \lor p_4) \land p_3 \land \neg p_4
\end{aligned}$$

After simplifying, we again have two unit literals p_3 and $\neg p_4$. We can continue by picking p_3, assigning it a satisfying value, and simplifying.

$$\begin{aligned}
&(\neg \top \lor p_4) \land \top \land \neg p_4 \\
\leftrightarrow\ &(\bot \lor p_4) \land \neg p_4 \\
\leftrightarrow\ &p_4 \land \neg p_4
\end{aligned}$$

Now all clauses are unit, and it is clear that the formula is not satisfiable. Notice that once we assigned p_1 $true$, we were able to determine that the resulting formula was unsatisfiable without making any further decisions. All of the resulting simplifications were a logical consequence of this original choice. The process of carrying this to its conclusion is called *Boolean constraint propagation* (BCP), or sometimes *unit propagation* for short.

4 DPLL

Using an implementation that resembles the one above for large problems would not yield good results in practice. One immediate problem is that the formula is copied multiple times and mutated in-place with each recursive call. While this makes it easy to keep track of which variables have already been assigned or implied via propagation, even through backtracking, it is extremely slow and cumbersome.

To address this, we will make use of a *partial interpretation*, which only assigns values to some of the variables in the formula. Rather than copying modified formulas around, we can directly mutate the partial interpretation. Because it remains *partial* throughout most of the execution, parts of the formula cannot be evaluated fully to a constant, but are instead *unresolved*.

Definition 2 (Status of a clause under partial interpretation)**.** Given a partial interpretation I, a clause is:

- Satisfied, if one or more of its literals is satisfied
- Conflicting, if all of its literals are assigned but not satisfied
- Unit, if it is not satisfied and all but one of its literals are assigned
- Unresolved, otherwise

For example, given the partial interpretation $I = \{p_1, \neg p_2, p_4\}$:

$(p_1 \lor p_3 \lor \neg p_4)$ is satisfied

$(\neg p_1 \lor p_2)$ is conflicting

$(p_2 \lor \neg p_4 \lor p_3)$ is unit

$(\neg p_1 \lor p_3 \lor p_5)$ is unresolved

Another missed opportunity is that the previous algorithm does not take advantage of BCP, which in the example above allowed us to conclude that the remaining formula, which origionally had five variables, was unsatisfiable with just one recursive call instead of the 2^5 that would have been necessary in our original naive implementation. This is a big improvement! Let's add it to our decision procedure and have a look at the consequences.

at the beginning of the procedure, before `F` is further inspected and any choices are made. This will ensure that if we are given a formula that is already reducible to a constant through BCP, then we won't do any unnecessary work by deciding values that don't matter. The resulting procedure is called the David-Putnam-Loveland-Logemann or DPLL procedure, as it was introduced by Martin Davis, Hilary Putnam, George Logemann, and Donald Loveland in the 1960s.

To see how DPLL might be implemented, we'll assume two new helper functions.

- `set_and_propagate`. This function takes a literal, a partial interpretation, and a CNF formula. It sets the corresponding literal in the partial valuation, and then uses BCP to propagate the consequences of this assignment until no further unit clauses can be found. It returns a list of literals that were assigned during this process, and a boolean indicating whether a conflict was detected.
- `backtrack`. This function takes a list of literals and a partial interpretation, and undoes the assignments made by the literals in the list.

Combining these elements, partial interpretations and BCP, we can write the DPLL procedure as seen below. The core of the procedure is the `choose` function, which is responsible for making decisions and backtracking. The `set_and_propagate` function is used to handle the consequences of these decisions, and the `eval_cnf` function is used to determine if the formula is satisfied under the current partial interpretation when there are no remaining unassigned literals. A few important details are worth noting:

1. If a satisfying assignment is found, then an exception is raised to terminate early, returning control to the outer `dpll` function.
2. If `set_and_propagate` detects a conflict, then the `backtrack` function is called to undo the assignments that led to the conflict. Subsequently, `choose` either tries the other assignment for the same variable, or terminates to the context from which it was called. At the topmost level, this will cause the first `choose` call to terminate without an exception, which means that conflicts were detected for every possible assignment of the variables, and the formula is unsatisfiable.

```
let dpll (cnf : cnf) : option valuation =
  let pval = Array.make cnf.nvars None in
  let rec choose (remaining: list int) =
    match remaining with
      | Nil -> if eval_cnf pval cnf then raise Sat_found
      | Cons v vs ->
        match pval[v] with
        | None ->
          let conflict_detected, diff =
            set_and_propagate {var=v; sign=false} pval cnf in
          if not conflict_detected then choose vs ;
          backtrack diff pval ;

          let conflict_detected, diff =
            set_and_propagate {var=v; sign=true} pval cnf in
          if not conflict_detected then choose vs ;
          backtrack diff pval ;
        | Some _ -> choose vs
        end
    end in

  let remaining = range 0 cnf.nvars in
  try choose remaining ; None
  with Sat_found -> Some (extract_sat_valuation pval cnf) end
```

Remarkably, although DPLL was introduced over 50 years ago, it still forms the basis of most modern SAT solvers. Much has changed since the 1960's, however, and the scale of SAT problems that are used in practice has increased dramatically. It is not uncommon to encounter instances with millions of atomic propositions and hundreds of thousands of clauses, and in practice it is often feasible to solve such instances.

As we discussed earlier, when a clause C is unit under partial interpretation I, I must be extended so that C's unassigned literal ℓ is satisfied. There is no need to backtrack on ℓ before the assignments in I that made C unit have already changed, because ℓ's value was implied by those assignments. Rather, backtracking can safely proceed to the *most recent decision*, erasing any assignments that arose from unit propagation in the meantime. Implementing this backtracking optimization correctly is essential to an efficient SAT solver, as it is what allows DPLL to avoid explicitly enumerating large portions of the search space in practice.

Learning conflict clauses. Consider the following CNF:

$$\underbrace{(\neg p_1 \lor p_2)}_{C_1} \land \underbrace{(\neg p_3 \lor p_4)}_{C_2} \land \underbrace{(\neg p_6 \lor \neg p_5 \lor \neg p_2)}_{C_3} \land \underbrace{(\neg p_5 \lor p_6)}_{C_4} \land \underbrace{(p_5 \lor p_7)}_{C_5} \land \underbrace{(\neg p_1 \lor p_5 \lor \neg p_7)}_{C_6}$$

And suppose we make the following decisions and propagations.

1. Decide p_1
2. Propagate p_2 from clause C_1

3. Decide p_3
4. Propagate p_4 from clause C_2
5. Decide p_5
6. Propagate p_6 from clause C_4
7. Conflicted clause C_3

At this point C_3 is conflicted. We should take a moment to reflect on our choices, and how they influenced this unfortunate outcome. We know that some subset of the decisions contributed to a partial assignment that cannot be extended in a way that leads to satisfiability, but which ones?

Tracing backwards, the implication p_6 was chronologically the most direct culprit, as it was incidental to the conflict in C_3. This was a consequence of our decision to set p_5, so we could conclude that this to blame and proceed backtracking to this point and change the decision. However, C_3 would not have been conflicting, even with p_5 and p_6, if not for p_2. Looking back at the trace, p_2 was a consequence of our decision to set p_1.

Thus, we learn from this outcome that $\neg p_1 \lor \neg p_5$ is logically entailed by our original CNF. The process that we used to arrive at this clause uses the resolution rule that we covered in Lecture 14. We can use this to quickly derive a proof that the clauses in our formula imply $\neg p_1 \lor \neg p_5$. In the following, let F be our original formula.

$$\begin{array}{ll}
\neg p_1 \lor p_2 & C_1 \\
\neg p_3 \lor p_4 & C_2 \\
\neg p_6 \lor \neg p_5 \lor \neg p_2 & C_3 \\
\neg p_5 \lor p_6 & C_4 \\
p_5 \lor p_7 & C_5 \\
\neg p_1 \lor p_5 \lor \neg p_7 & C_6 \\
\hline
\neg p_5 \lor \neg p_2 & C_7 = C_3 \bowtie_p C_4 \\
\neg p_1 \lor \neg p_5 & C_8 = C_1 \bowtie_r C_7
\end{array}$$

Clauses derived in this way are called *conflict clauses*, and they are useful in pruning the search space. In the current example, suppose that we added the conflict clause $\neg p_1 \lor \neg p_5$ to our set. Then any partial interpretation with p_1 makes this clause unit, implying the assignment $\neg p_5$.

5. Backtrack to p_5
6. Learn clause $C_7 \leftrightarrow \neg p_1 \lor \neg p_5$
7. Propagate $\neg p_5$ from clause C_7
8. ...

Without this, if we eventually backtrack past p_5 to change the assignment to p_3, then when the procedure revisits p_5 it will attempt both assignments p_5 and $\neg p_5$, encountering the same conflict again.

To summarize, the procedure for finding a conflict clause under partial assignment I is as follows.

```
1. Let $C$ be a conflicting clause under $I$
2. While $C$ contains implied literals, do:
3.     Let $\ell$ be the most recent implied literal in $C$
4.     Let $C'$ be the clause that implied $\ell$ by unit propagation
5.     Update $C$ by applying resolution to $C$ and $C'$ on $\ell$
```

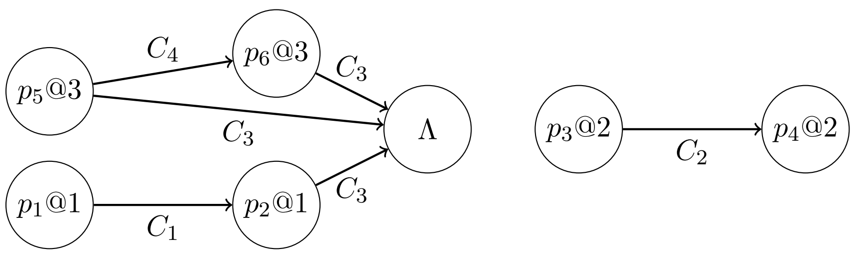
This procedure terminates when all of the literals in C correspond to decisions made by `dpll`. However, the conflict clause produced in this way is by no means the only sound or useful such clause that can be derived. The most efficient way to find others is to construct an *implication graph*.

Definition 3 (Implication graph)**.** An implication graph for partial assignment I is a directed acyclic graph with vertices V and edges E, where:

- Each literal ℓ_i in I corresponds to a vertex $v_i \in V$.
- Each edge $(v_i, v_j) \in E$ corresponds to an implication brought about by unit propagation. That is, if ℓ_j appears in I because of a unit propagation, and ℓ_i appears in the corresponding unit clause that brought about this propgation, then $(v_i, v_j) \in E$.
- V contains a special *conflict vertex* Λ, which only has incoming edges $\{(\ell, \Lambda) | \ell \in C\}$ for each literal appearing in a conflicting clause C.

The implication graph is a data structure maintained by many efficient implementations of DPLL. As assignments are added to a partial interpretation, the graph is updated with new nodes and edges to keep track of the relationship between decisions and their implied consequences. Likewise, nodes and edges are removed to account for backtracking.

The implication graph for our running example is shown below.



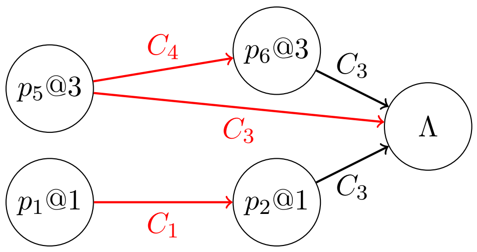
The three decisions we made correspond to roots of the graph, and implications are internal nodes. We also keep track of at which *decision level* each vertex appeared, with the @ notation. Recall that we began (decision level 1) by deciding p_1, which implied p_2 by unit propagation. The responsible clause, in this case C_1, labels the edge that reflects this implication.

Visually, the implication graph makes the relevant facts quite obvious. First, notice the subgraph containing vertices $p_3@2$ and $p_4@2$. The decision to assign p_3 ended up being irrelevant to the eventual conflict in C_3, and this is reflected in the fact that the subgraph is disconnected from the conflict node. When analyzing a conflict, we can simply ignore subgraphs disconnected from the conflict node.

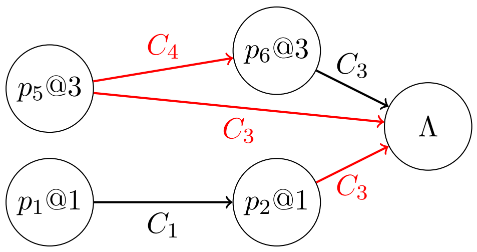
Focusing only on the subgraph connected to the conflict node, the correspondence between the roots and the conflict clause we obtained via resolution, $\neg p_1 \vee \neg p_5$, is immediate. This is not an accident, and in fact is the entire reason for building an implication graph in the first place. We can use this data structure to generalize on the resolution-based procedure outlined above by identifying *separating cuts* in the implication graph.

Definition 4 (Separating cut)**.** A separating cut in an implication graph is a minimal set of edges whose removal breaks all paths from the roots to the conflict nodes.

The separating cut partitions the implication graph into two sides, which we can think of as the "reason" side and the "conflict" side. Importantly, any set of vertices on the "reason" side with at least one edge to a vertex on the "conflict" side corresponds to a sufficient condition for the conflict. We obtain a conflict clause by negating the literals that correspond to these vertices. In the example from earlier, we chose the following edges highlighted in red for our conflict clause.



However, we could have just as well chosen the following, which would have led to the clause $\neg p_5 \vee \neg p_2$.



Any conflict clause corresponding to such a cut is derivable using the resolution rule, and is safe to add to the clause set. Different procedures have various ways of selecting cuts. Some choose to compute several cuts, aggressively adding multiple conflict clauses to further constrain the search. Most modern solvers aim to find a single effective cut that corresponds to an *asserting clause*, which forces an implication immedately after backtracking. Because SAT is a hard problem, these are heuristic choices that may or may not improve performance on different classes of instances. For any sound startegy, such choices are best validated empirically to identify those that yield the best results on important problems that arise in practice.

5 Worked example.

To get a better sense of how DPLL with clause learning works, we will illustrate its full application to an example formula, given by the set of CNF clauses below.

$p \vee q$	C_1
$\neg p \vee q$	C_2
$\neg r \vee \neg q$	C_3
$r \vee \neg q$	C_4

The following trace of DPLL with clause learning demonstrates the unsatisfiability of these clauses.

1. Decide p
2. Propagate q from C_2
3. Propagate $\neg r$ from C_3
4. Conflict on C_4
5. Backtrack to p
6. Learn $\neg p$ by resolving C_4 with C_3 (yielding $\neg q$), and then with C_2
7. Propagate q from C_1
8. Propagate $\neg r$ from C_3
9. Conflict on C_4
10. Return `unsat`

Notice that the procedure ended after encountering the second conflict on C_4. After adding the learned clause $\neg p$ to the formula and applying BCP, this conflict arose without having made any more decisions. In other words, the original clauses entailed the unit clause $\neg p$ via resolution, and when the resulting formula is simplified by applying unit propagation, C_4 is simplified to $\bot$. This means that the formula must be unsatisfiable, so the procedure can stop and return that result.