15-414: Bug Catching: Automated Program Verification

Lecture Notes on Satisfiability & DPLL

Matt Fredrikson

Carnegie Mellon University

Lecture 13

1 Introduction

In this lecture we will switch gears a bit from proving logical theorems "by hand", to algorithmic techniques for proving them 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

We'll focus on automating the decision problem for Boolean satisfiability. Let's start by refreshing ourselves on the fundamentals of propositional logic. The formulas F, G of propositional logic are defined by the following grammar (where p is an atomic proposition, or *atom*):

$$F ::= \top \mid \bot \mid p \mid \neg F \mid F \land G \mid F \lor G \mid F \rightarrow G \mid F \leftrightarrow G$$

When it comes to the semantics, recall that the meaning of formulas is given by an interpretation I that gives the truth value for each atom. Given an interpretation, we can assign values to formulas constructed using the logical operators.

Definition 1 (Semantics of propositional logic)**.** The propositional formula F is true in interpretation ω, written $I \models F$, as inductively defined by distinguishing the shape of formula F:

1. $I \models \top$ for all interpretations I.
2. $I \not\models \bot$ for all interpretations I.
3. $I \models p$ iff $I(p) = \mathit{true}$ for atoms p.
4. $I \models F \land G$ iff $I \models F$ and $I \models G$.
5. $I \models F \lor G$ iff $I \models F$ or $I \models G$.
6. $I \models \neg F$ iff $I \not\models F$.
7. $I \models F \to G$ iff $I \not\models F$ or $I \models G$.

Our notation for interpretations is essentially a list of all atoms that are true. So, the interpretation:

$$I = \{p, q\}$$

assigns the value true to p and q, and *false* to all others. For example, the formula in Equation 1 below would evaluate to *false* under I, because $I(p) = \mathit{true}, I(q) = \mathit{true}, I(r) = \mathit{false}$ so $I \models p \land q$ and $I \not\models p \land q \to r$.

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

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

Definition 2 (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.

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?

4 Normal Forms and Simplification

Before continuing, we should address a major uncertainty about our naive sat procedure. We assumed that the function subst "simplifies" the formula after making a substitution, but what exactly does this mean and how can we do it efficiently? In practice, most SAT procedures require that formulas be provided in a *normal form*, which makes this step easier.

Basic Identities. When a formula contains the constants $\top$ and $\bot$, a number of simplifications follow directly from the semantics of propositional logic. We will ignore the implication operator, as it can be rewritten in terms of negation and disjunction. For negation, conjunction, and disjunction, we can use the following equivalences to rewrite formulas containing constants in simpler terms:

$$\neg\top \leftrightarrow \bot \leftrightarrow F \wedge \bot \tag{3}$$

$$\neg\bot \leftrightarrow \top \leftrightarrow F \vee \top \tag{4}$$

$$F \vee \bot \leftrightarrow F \leftrightarrow F \wedge \top \tag{5}$$

$$\tag{6}$$

Repeatedly applying these simplifications to a formula containing *only* constants will eventually lead to either $\top$ or $\bot$. However, practical SAT solvers use additional strategies to further reduce the space of interpretations that need to be considered by the decision procedure.

Negation Normal Form and Pure Literal Elimination. The term "normal form" in this context refers to a class of formulas that have a particular syntactic property. When writing programs to efficiently process formulas, it is common to assume that formulas will be given to the program in a normal form that reduces the number of cases to be considered and admits uniform reasoning techniques. One such example is called negation normal form (NNF).

Definition 3 (Negation Normal Form (NNF))**.** A formula is in negation normal form if negation occurs only over atoms and the only Boolean operators are $\land, \lor$, and $\neg$.

For example, $(\neg p \lor \neg q) \land r$ is in NNF, but $\neg(p \land q) \land r$ is not because a negation occurs on the conjunction. Any propositional formula can be transformed to an equivalent NNF formula in linear time by rewriting implications and applying De Morgan's laws.

$$P \to Q \leftrightarrow \neg P \lor Q \tag{7}$$

$$\neg(P \lor Q) \leftrightarrow \neg P \land \neg Q \tag{8}$$

$$\neg(P \land Q) \leftrightarrow \neg P \lor \neg Q \tag{9}$$

When a formula is in NNF, we know that negations will only occur over atoms. We call an atom or its negation a *literal*, and say that a literal is *positive* if it does not have a negation, and that a literal is *negative* if it does have a negation.

This already allows us to simplify formulas in some cases. For example, consider the formula:

$$(p \lor q \lor \neg r) \land (\neg p \lor s) \land (\neg r \lor (\neg q \land \neg s)) \tag{10}$$

In this formula, the atom r only occurs in negative literals. A bit of thought should convince us that in this case, there is no point in assigning the value $true$ to r in an interpretation – this would cause both literals containing r to evaluate to $false$, and therefore result in any conjunction containing r to also evaluate $false$.

Any atom that only appears in either positive or negative literals is called *pure*, and their corresponding atoms must always be assigned in a way that makes the literal $true$. Thus, they do not constrain the problem in a meaningful way, and can be assigned without making a choice. This is called *pure literal elimination*, and is one type of simplification that can be applied to NNF formulas. In the former example, we would have:

$$\begin{aligned}
(p \lor q \lor \neg r) \land (\neg p \lor s) \land (\neg r \lor (\neg q \land \neg s)) &\leftrightarrow \\
(p \lor q \lor \neg\bot) \land (\neg p \lor s) \land (\neg\bot \lor (\neg q \land \neg s)) &\leftrightarrow \\
(p \lor q \lor \top) \land (\neg p \lor s) \land (\top \lor (\neg q \land \neg s)) &\leftrightarrow \\
\top \land (\neg p \lor s) \land \top &\leftrightarrow \\
(\neg p \lor s)&
\end{aligned}$$

In practice, pure literal elimination can significantly reduce the complexity of propositional formulas, and so it is sometimes used as a pre-processing simplification before handing the formula to a solver. However, oftentimes there are no pure literals in a formula. Introducing more structure into the normal form can lead to more opportunities.

Conjunctive Normal Form and Unit Resolution. A further restriction on NNF formulas is called conjunctive normal form (CNF).

Definition 4 (Conjunctive Normal Form (CNF))**.** A formula F is in conjunctive normal form if it is a conjunction of disjunctions of literals, i.e., it has the form:

$$\bigwedge_i \left(\bigvee_j l_{ij} \right)$$

where l_{ij} is the jth literal in the ith *clause* of F.

Every formula can be converted into CNF using the same approach as for NNF, but this may cause the size of the formula to increase exponentially. However, it is possible to transform any propositional formula into an *equisatisfiable* one in linear time. Two formulas F and G are equisatisfiable when F is satisfiable if and only if G is as well. We will not cover the details of such transformations, but more information is availble in Chapter 1 of [BM07].

Consider the following CNF formula:

$$\underbrace{(p_1 \lor \neg p_3 \lor \neg p_5)}_{C_1} \land \underbrace{(\neg p_1 \lor p_2)}_{C_2} \land \underbrace{(\neg p_1 \lor \neg p_3 \lor p_4)}_{C_3} \land \underbrace{(\neg p_1 \lor \neg p_2 \lor p_3)}_{C_5} \land \underbrace{(\neg p_4 \lor \neg p_2)}_{C_6} \quad (11)$$

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 if we assign p_1 to $true$ then resulting formula is not satisfiable. Notice that once we assigned p_1 to $true$, 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.

5 DPLL

BCP allowed us to conclude that the remaining formula, which originally 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.

The natural place to insert this optimization is 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 [DP60, DLL62].

```
let rec dpll (F:formula) : bool =
  let Fp = BCP F in
  if Fp = true then true
  else if Fp = false then false
  else begin
    let p = choose_atom(Fp) in
    let Ft = (subst Fp p true) in
    let Ff = (subst Fp p false) in
    dpll Ft || dpll Ff
  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.

Using an implementation that resembles the one above for such 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.

Modern solvers address this by using imperative loops rather than recursive calls, and mutating an interpretation rather than the formula itself. The interpretation remains *partial* throughout most of the execution, which means that parts of the formula cannot be evaluated fully to a constant, but are instead *unresolved*.

Definition 5 (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

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

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

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:

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

2. Propagate p_2 from clause C_1

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

3. Decide p_3

$$\underbrace{(\bot \vee \top)}_{C_1} \wedge \underbrace{(\bot \vee p_4)}_{C_2} \wedge \underbrace{(\neg p_6 \vee \neg p_5 \vee \bot)}_{C_3} \wedge \underbrace{(\neg p_5 \vee p_6)}_{C_4} \wedge \underbrace{(p_5 \vee p_7)}_{C_5} \wedge \underbrace{(\bot \vee p_5 \vee \neg p_7)}_{C_6}$$

4. Propagate p_4 from clause C_2

$$\underbrace{(\bot \vee \top)}_{C_1} \wedge \underbrace{(\bot \vee \top)}_{C_2} \wedge \underbrace{(\neg p_6 \vee \neg p_5 \vee \bot)}_{C_3} \wedge \underbrace{(\neg p_5 \vee p_6)}_{C_4} \wedge \underbrace{(p_5 \vee p_7)}_{C_5} \wedge \underbrace{(\bot \vee p_5 \vee \neg p_7)}_{C_6}$$

5. Decide p_5

$$\underbrace{(\bot \vee \top)}_{C_1} \wedge \underbrace{(\bot \vee \top)}_{C_2} \wedge \underbrace{(\neg p_6 \vee \bot \vee \bot)}_{C_3} \wedge \underbrace{(\bot \vee p_6)}_{C_4} \wedge \underbrace{(\top \vee p_7)}_{C_5} \wedge \underbrace{(\bot \vee \top \vee \neg p_7)}_{C_6}$$

6. Propagate p_6 from clause C_4

$$\underbrace{(\bot \vee \top)}_{C_1} \wedge \underbrace{(\bot \vee \top)}_{C_2} \wedge \underbrace{(\bot \vee \bot \vee \bot)}_{C_3} \wedge \underbrace{(\bot \vee \top)}_{C_4} \wedge \underbrace{(\top \vee p_7)}_{C_5} \wedge \underbrace{(\bot \vee \top \vee \neg p_7)}_{C_6}$$

7. Conflicted clause C_3

At this point C_3 is conflicted. We should take a moment to reflect on how we got here. We know that some subset of the decisions contributed to a partial assignment that led to conflict, 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 and the presence of C_4, 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, and the fact of C_1 being in our formula.

Thus, we learn from this outcome that $\neg p_1 \vee \neg p_5$ is logically entailed by our original CNF. The process that we used to arrive at this clause is called *resolution*, and corresponds to repeated application of the binary resolution rule.

$$(\text{res})\ \frac{\Gamma \vdash P, \Delta \quad \Gamma \vdash \neg P, \Delta}{\Gamma \vdash \Delta}$$

Indeed, we can use this to quickly derive a proof that the clauses in our formula imply $\neg p_1 \vee \neg p_5$. We get this by applying res to the clauses involved in the conflict: first C_3 and C_4, and then C_1.

In the following, let F be our original formula.

$$\text{res}\dfrac{\text{res}\dfrac{\vee\text{L}\dfrac{\vdots}{F \vdash \neg p_6, \neg p_5, \neg p_2} \qquad \vee\text{L}\dfrac{\vdots}{F \vdash \neg p_5, p_6}}{F \vdash \neg p_5, \neg p_2} \qquad \vee\text{L}\dfrac{\vdots}{F \vdash \neg p_1, p_2}}{F \vdash \neg p_1, \neg p_5}$$

In the above proof, we did not close out any branches, but they can be seen to follow immediately from (going left to right) the presence of C_3, C_4, and C_1 in F.

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 $\neg p_1 \vee \neg p_5$ 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 $C_7 \leftrightarrow \neg p_1 \vee \neg p_5$
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 would attempt both assignments p_5 and $\neg p_5$. Because p_1 has not changed, and we have now proved that $F \rightarrow (\neg p_1 \vee \neg p_5)$, this would lead to the very 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 $\ell$
```

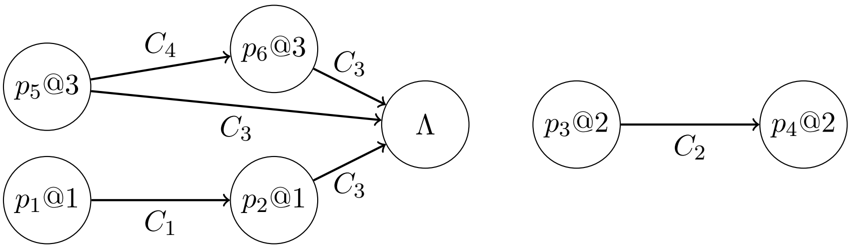
This procedure terminates when all of the literals in C correspond to decisions made by `dpll`. However, the conflict clause produced in this way is by no means the only sound or useful such clause that can be derived. The most efficient way to find others is to construct an *implication graph*.

Definition 6 (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 propagation, 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.



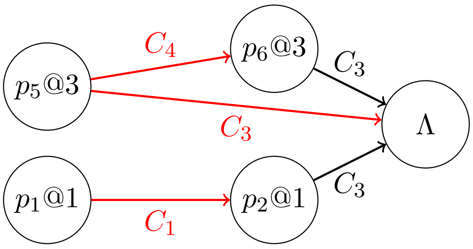
The three decisions we made correspond to roots of the graph, and implications are internal nodes. We also keep track of at which *decision level* each vertex appeared, with the $@$ notation. Recall that we began (decision level 1) by deciding p_1, which implied p_2 by unit propagation. The responsible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

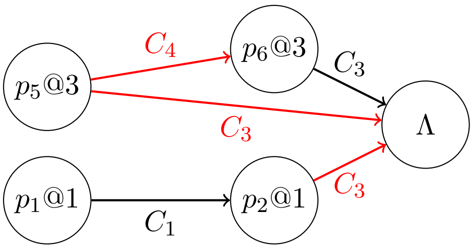
Focusing only on the subgraph connected to the conflict node, the correspondence between the roots and the conflict clause we obtained via resolution, $\neg p_1 \vee \neg p_5$, is immediate. This is not an accident, and in fact is the entire reason for building an implication graph in the first place.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

Definition 7 (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 immediately 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 strategy, such choices are best validated empirically to identify those that yield the best results on important problems that arise in practice.

6 Summary

- **Decision procedure** is an algorithm that, given a decision problem, terminates with a correct yes/no answer. In this lecture we presented a decision procedure for propositional logic, also known as SAT solvers.
- Duality between **satisfiability** and **validity**: F is valid $\leftrightarrow$ $\neg$ F is unsatisfiable.
- To perform efficient formula simplification, we must have our formula in a **normal form** such as **negation normal form (NNF)** or **conjunctive normal form (CNF)**.
- **Boolean constraint propagation (BCP)** (also known as unit propagation) is the process of applying the **unit resolution** until a fix point.
- **DPLL** is the basis of most modern SAT solvers and performs BCP at each branching step.

- **Status of a clause**: satisfied, conflicting, unit or unresolved.
- Learned clauses can be derived by using the **resolution** rule.
- In practice, learned clauses are derived from the **implication graph** of the current interpretation.

References

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.