Decision Procedures

Azadeh Farzan

CS410 - Fall 2020
A decision procedure is an algorithm that given a decision problem, terminates with a correct yes/no answer.

Play an important part in automated verification, theorem proving, compiler optimization, synthesis, and many areas of AI.

The problems are inherently difficult.

Procedures need to be a very efficient.
Quick Example

Are these two code fragments equivalent?

```
if(!a && !b) h();  
else               
  if(!a) g();      
  else f();       
```

```
if(a) f();  
else       
  if(b) g();  
  else h();  
```

It seems to be a **YES/NO** type of problem. Can we state this generally as a **decision problem**?

\[
(\text{if } x \text{ then } y \text{ else } z) \equiv (x \land y) \lor (\neg x \land z)
\]

\[
(\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \equiv a \land f \lor \neg a \land (b \land g \lor \neg b \land h).
\]
A propositional formula is **satisfiable** if there is an assignment (to propositions) that makes the formal to evaluate to true.

\[ A \land B \]

A formula is **valid** if all assignments make it to evaluate to true.

\[ A \Rightarrow A \]
A **theory** is (informally):

- A finite or infinite set of formulas, which are characterized by common grammatical rules, functions and predicates, and a domain of values.

---

<table>
<thead>
<tr>
<th>Theory name</th>
<th>Example formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional logic</td>
<td>$x_1 \land (x_2 \lor \neg x_3)$</td>
</tr>
<tr>
<td>Equality</td>
<td>$y_1 = y_2 \land \neg(y_1 = y_3) \implies \neg(y_1 = y_3)$</td>
</tr>
<tr>
<td>Linear arithmetic</td>
<td>$(2z_1 + 3z_2 \leq 5) \lor (z_2 + 5z_2 - 10z_3 \geq 6)$</td>
</tr>
<tr>
<td>Bit vectors</td>
<td>$((a &gt;&gt; b) &amp; c) &lt; c$</td>
</tr>
<tr>
<td>Arrays</td>
<td>$(i = j \land a[j] = 1) \implies a[i] = 1$</td>
</tr>
<tr>
<td>Pointer logic</td>
<td>$p = q \land *p = 5 \implies *q = 5$</td>
</tr>
<tr>
<td>Combined theories</td>
<td>$(i \leq j \land a[j] = 1) \implies a[i] &lt; 2$</td>
</tr>
</tbody>
</table>
More Definitions

A procedure for a decision problem is **sound** if when it answers "YES" to the validity/satisfiability question, the formula is valid/satisfiable.

A procedure for a decision problem is **complete** if it always terminates, and when formula is valid, it answers "YES" to the validity question.

A procedure is called a **decision procedure** for T if it is sound and complete with respect to every formula in T.

A theory is **decidable** if and only if there is a decision procedure for it.
Theories and Algorithms

A theory is interesting if:

- It is expressive enough to model a real decision problem.
- It is either decidable or semi-decidable, and more efficiently solvable than more expressive theories.

![Diagram showing the trade-off between expressiveness of theories and decidability.](image)

Many applications, however, require a more complex Boolean structure. In program analysis and verification, for example, disjunctions may appear in the program to be verified, either explicitly (e.g., \( x = y \lor z \)) or implicitly through constructs such as if and switch statements. Any reasoning system about such programs, therefore, must be able to deal with disjunctions. For example, verification conditions that arise in program verification (e.g., using Hoare logic), often have the form of an implication.

Example 1.25. Bounded model checking (BMC) of programs is a technique for verifying that a given property (typically given as an assertion by the user) holds for a program in which the number of loop iterations and recursive calls is bounded by a given number \( k \). The tests that the program can reach within this bound are represented symbolically by a formula, together with the negation of the property. If the combined formula is satisfiable, then there exists a path in the program that violates the property.

Consider the program in the left part of Fig. 1.4. The number of paths through this program is exponential in \( N \), as each of the \( a[i] \) elements can be either zero or nonzero. Despite the exponential number of paths through the program, its states can be encoded with a formula of size linear in \( N \), as demonstrated in the right part of the figure.

The formula on the right of Fig. 1.4 encodes the states of the program on its left, using the static-single-assignment (SSA) form. Briefly, this means that in each assignment of the form \( x = \text{exp} \), the left-hand side variable \( x \) is replaced with a new variable, say \( x_1 \), and a reference to \( x \) after this line and before \( x \) is assigned again is replaced with \( x_1 \). Such replacement is possible because there are no loops (recall that this is done in the context of BMC). After this transformation, the statements are conjoined. The resulting equation represents the states of the original program.
Propositional Logic
SAT Solvers

Given a propositional formula $F$, a SAT Solver decides if $F$ is satisfiable.
Propose a trivial SAT solving Algorithm.

The **satisfiability problem** for propositional logic is NP-Complete.
SAT Solving Techniques

- **DPLL-based**
  - Davis-Putnam-Loveland-Logemann
  - Traversing and backtracking on a binary tree, in which internal nodes represent partial assignments, and leaves present full assignments.

- **Stochastic Search**
  - The solver guesses a full assignment, and if it doesn’t work, it starts flipping values based on some greedy heuristic.
Let’s see a reasonable SAT-solving algorithm ...
A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses

\[ C_1 \land C_2 \land \ldots \land C_n \]

each clause \( C \) is a disjunction of literals

\[ C = L_1 \lor \ldots \lor L_m \]

and each literal is either a plain variable \( x \) or a negated variable \( \overline{x} \).

**Example** \( (a \lor b \lor c) \land (\overline{a} \lor \overline{b}) \land (\overline{a} \lor \overline{c}) \)

For every Propositional formula there exists an *equisatisfiable* propositional formula in CNF which is at most *polynomially* larger.
A clause is **satisfied** if one or more of its literals are satisfied.

A clause is **conflicting** if all of its literals are assigned, but not satisfied.

A clause is **unit** if it is not satisfied and all but one of its literals are assigned.

A clause is **unresolved** if it is none of the above.

\[
\{ x_1 \mapsto 1, \ x_2 \mapsto 0, \ x_4 \mapsto 1 \} \\
(x_1 \lor x_3 \lor \neg x_4) \\
(\neg x_1 \lor x_2) \\
(\neg x_1 \lor \neg x_4 \lor x_3) \\
(\neg x_1 \lor x_3 \lor x_5)
\]
Simple Algorithm Overview

**Partial Evaluations:**
- We start with the **empty evaluation** (no variable has been assigned).
- We step by step **extend it** to all variables.

**Davis-Putnam-Logemann-Loveland basic structure:**

```plaintext
boolean DPLL(clause set N, partial valuation A) {
    if (all clauses in N are true under A) return true;
    elsif (some clause in N is false under A) return false;
    elsif (N contains unit clause P) return DPLL(N, A ∪ {P ↦ 1});
    elsif (N contains unit clause ¬P) return DPLL(N, A ∪ {P ↦ 0});
    else {
        let P be some undefined variable in N;
        if (DPLL(N, A ∪ {P ↦ 0})) return true;
        else return DPLL(N, A ∪ {P ↦ 1});
    }
}
```

Start with empty

Iterative Algorithm with smart backtracking

Not chosen randomly
The better algorithm takes shortcuts ...
Algorithm 2.2.1: DPLL-SAT

Input: A propositional CNF formula

Output: "Satisfiable" if the formula is satisfiable and "Unsatisfiable" otherwise

1. function DPLL
2. if $\text{BCP}() = \text{conflict}$ then return "Unsatisfiable";
3. while true do
4. if $\neg \text{Decide}()$ then return "Satisfiable";
5. else
6. while $\text{BCP}() = \text{conflict}$ do
7. backtrack-level := Analyze-Conflict();
8. if backtrack-level < 0 then return "Unsatisfiable";
9. else BackTrack(backtrack-level);

Fig. 2.4. DPLL-SAT: high-level overview of the Davis-Putnam-Loveland-Logemann algorithm. The variable $dl$ is the decision level to which the procedure backtracks.

2.2.3 BCP and the Implication Graph

We now demonstrate Boolean constraints propagation (BCP), reaching a conflict, and backtracking. Each assignment is associated with the decision level at which it occurred. If a variable $x_i$ is assigned 1 (true) (owing to either a decision or an implication) at decision level $dl$, we write $x_i@dl$. Similarly, $\neg x_i@dl$ reflects an assignment of 0 (false) to this variable at decision level $dl$. Where appropriate, we refer only to the true assignment, omitting the decision level, in order to make the notation simpler.
Each assignment is associated with a decision level, the level at which a decision or implication was made. BCP is performed over an implication graph. The graph represents a current partial assignment and the reason for each implication.

\[
\begin{align*}
  c_1 &= (\neg x_1 \lor x_2), \\
  c_2 &= (\neg x_1 \lor x_3 \lor x_5), \\
  c_3 &= (\neg x_2 \lor x_4), \\
  c_4 &= (\neg x_3 \lor \neg x_4), \\
  c_5 &= (x_1 \lor x_5 \lor \neg x_2), \\
  c_6 &= (x_2 \lor x_3), \\
  c_7 &= (x_2 \lor \neg x_3), \\
  c_8 &= (x_6 \lor \neg x_5).
\end{align*}
\]
A labeled directed acyclic graph \( G(V,E) \) where:

- \( V \) represents the literals of the current assignment. Each node is labeled with a literal and its decision level.

- \( E \) represents the connection between literals. \((u,v)\) belongs to the graph if the assignment to \( u \) plays a role in what value should be assigned to \( v \) (they both belong to a unit clause).

- \( G \) can also contain a single conflict node labeled with \( k \) (now the incoming edges are all from the literals that are part of a conflict clause with \( k \)).

- The root nodes correspond to decisions and the internal nodes correspond to implications made by propagation.

The graph may be partial; only referring to a certain decision level.
Assume that at decision level 3 the decision was

The root nodes of an implication graph correspond to decisions, and the inter-

levels lower than a partial graph represent assignments (not necessarily decisions) at decision

and the order of clauses in this list can be sensitive to the order of clauses in

A partial implication graph for decision level 6, corresponding to the

in (2.7), after a decision

Decision

x_1@6

x_2@6

x_3@6

x_4@6

κ

¬x_5@3

x_1@6

is indeed the clause generated.

In the case of learning a unary clause, the solver backtracks to the ground level.

Analyze-Conflict

While

κ

asserting clause

for some conflicting clause

\[ c = (x_5 \lor \lnot x_1) \]

to our formula.

Therefore, we can safely add a conflict clause:

\[ c_9 = (x_5 \lor \lnot x_1) \]

generally called learning.

backtrack to the highest decision level before the current one.
We backtrack to level 3 after learning the conflict clause.

We erase all the decisions and implications made after that level, including assignments to $x_1, x_2, x_3, x_4$.

$$c_9 = (x_5 \lor \neg x_1)$$

\begin{align*}
c_1 &= (\neg x_1 \lor x_2), \\
c_2 &= (\neg x_1 \lor x_3 \lor x_5), \\
c_3 &= (\neg x_2 \lor x_4), \\
c_4 &= (\neg x_3 \lor \neg x_4), \\
c_5 &= (x_1 \lor x_5 \lor \neg x_2), \\
c_6 &= (x_2 \lor x_3), \\
c_7 &= (x_2 \lor \neg x_3), \\
c_8 &= (x_6 \lor \neg x_5).
\end{align*}

Clause $c_9$ was a special kind of conflict clause, called asserting clause which forced an immediate implication after backtracking.
Conflict Resolution

How are conflict clauses generated? (specifically, asserting clauses)
Conflict Resolution

Binary resolution inference rule:

\[
\frac{(a_1 \lor \ldots \lor a_n \lor \beta) \quad (b_1 \lor \ldots \lor b_m \lor \neg \beta)}{(a_1 \lor \ldots \lor a_n \lor b_1 \lor \ldots \lor b_m)}
\]

An inference system based on the above rule for propositional logic is sound and complete.

Unsatisfiability of a CNF formula is decided through finitely many applications of the resolution rule.
Example

\[\begin{align*}
c_1 &= (\neg x_4 \lor x_2 \lor x_5) \\
c_2 &= (\neg x_4 \lor x_{10} \lor x_6) \\
c_3 &= (\neg x_5 \lor \neg x_6 \lor \neg x_7) \\
c_4 &= (\neg x_6 \lor x_7) \\
\vdots
\end{align*}\]

Implication Order in BCP:

\[x_4, x_5, x_6, x_7\]
Example

$c_1 = (\neg x_4 \lor x_2 \lor x_5)$
$c_2 = (\neg x_4 \lor x_{10} \lor x_6)$
$c_3 = (\neg x_5 \lor \neg x_6 \lor \neg x_7)$
$c_4 = (\neg x_6 \lor x_7)$

... 

Implication Order in BCP:

$x_4, x_5, x_6, x_7$

The first UIP

An asserting clause which contains the negation of $x_4$.

Stopping criteria: negation of first UIP is the only literal from the current decision level.
Conflict Resolution

What is the stopping condition for binary resolution steps?

- Given a partial graph for a decision level, a unique implication point (UIP) is a node (other than the conflict node) that is on all paths from the decision node to the conflict node.
- First UIP is a UIP that is closest to the conflict node.

Why first? It is faster and one backtracks to the lowest level.
Termination

Why does this terminate? How do we know that a partial assignment cannot be repeated forever?

**Theorem:** It is never the case that the solver enters decision level $dl$ again with the same partial assignment.

2.2 SAT Solvers

of them after a while to prevent the formula from growing too much. The reason is the following.

**Theorem 2.8.** It is never the case that the solver enters decision level $dl$ again with the same partial assignment.

**Proof.** Consider a partial assignment up to decision level $dl - 1$ that does not end with a conflict, and assume falsely that this state is repeated later, after the solver backtracks to some lower decision level $dl - \beta (0 \leq \beta < dl)$.

Any backtracking from a decision level $dl + \alpha (\alpha \geq 0)$ to decision level $dl - \beta$ adds an implication at level $dl - \beta$ of a variable that was assigned at decision level $dl + \alpha$. Since this variable has not so far been part of the partial assignment up to decision level $dl$, once the solver reaches $dl$ again, it is with a different partial assignment, which contradicts our assumption.

The (hypothetical) progress of a SAT solver based on this strategy is illustrated in Fig. 2.7. More details of this graph are explained in the caption.

![Diagram illustrating the progress of a SAT solver](image.png)

**Fig. 2.7.** Illustration of the progress of a SAT solver based on conflict-driven backtracking. Every conflict results in a conflict clause (denoted by $c_1, \ldots, c_5$ in the drawing). If the top left decision is $x = 1$, then this drawing illustrates the work done by the SAT solver to refute this wrong decision. Only some of the work during this time was necessary for creating $c_5$, refuting this decision, and computing the backtracking level. The “wasted work” (which might, after all, become useful later on) is due to the imperfection of the decision heuristic.

2.2.4 Conflict Clauses and Resolution

Now consider Analyze-Conflict (Algorithm 2.2.2). The description of the algorithm so far has relied on the fact that the conflict clause generated is...