Chapter 6:
Propositional Logic: SAT solvers

Boolean satisfiability, or SAT for short, is the decision problem of determining whether a given propositional-logic formula (represented as a set of clauses) is satisfiable. SAT was the first decision problem to be shown to be *NP-complete*.

Briefly (and informally), a decision problem is said to belong to the class NP if there exists a nondeterministic algorithm that 'solves' it in time bounded by a polynomial in the size of an efficient description of a particular instance of the problem. We know that a truth table has 2^n rows, where n is the number of atomic propositions in a formula. Thus, any *deterministic* algorithm for satisfiability based on an exhaustive check of the truth table runs in exponential – not polynomial – time. A *nondeterministic* algorithm could 'guess' an interpretation, check whether that specific interpretation satisfied all of the clauses in the set, and return 'satisfiable' if it did. This checking can be done – by a deterministic algorithm – in a number of steps that is polynomial in the 'size' of the set of clauses, and consequently, such a nondeterministic algorithm is considered to 'solve' SAT, and to do so in polynomial time. That means that SAT belongs to the class NP. This class can be thought of consisting of those problems for which any 'guessed' candidate solution can be deterministically *checked* in polynomial time.

But SAT is also *NP-hard*. That means that any problem in NP can be reduced to SAT in polynomial time by a deterministic algorithm. In a sense, this means that SAT is as difficult as any problem in NP. More specifically, if there were to exist a deterministic algorithm that solved SAT in polynomial time, the same would be true of all problems in NP.

When a problem belongs to NP and is NP-hard, it is called *NP-complete*. Despite decades of effort, no one has ever been able to show that any NP-complete problem belongs to P – the class of decision problems that can be solved in polynomial time by a *deterministic* algorithm. On the other hand, no one has ever been able to show that any NP-complete problem definitely cannot be solved in deterministic polynomial time. So P $\subseteq$ NP, but no one has been able to show whether or not the reverse inclusion holds.

A particularly great amount of effort has been directed toward solving SAT as efficiently as possible; and because SAT is NP-hard, any problem in NP can be reduced to SAT in deterministic polynomial time, so 'SAT-solving'

algorithms can in principle be applied to the solution of any problem in NP.

In fact, SAT solvers are applied to many real problems. For example, package installers, that check whether different sets of packages within an operating system distribution can be installed together without conflicts, employ SAT solvers; SAT solvers are currently a tool of choice for the formal verification of large computing systems, and are the basis of many artificial-intelligence systems. This chapter gives a brief introduction the the state of the art in SAT solving.

6.2 Davis-Putnam algorithm

One of the first algorithms proposed for efficiently deciding satisfiability was the Davis-Putnam (DP) algorithm, based on resolution. For simplicity and brevity, we shall pass directly to its improved version, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm.

6.3 DPLL algorithm

The DPLL algorithm is essentially a method of searching for a satisfying partial interpretation of a set of clauses.

Definition 6.18 Let S be a set of clauses and let $\mathscr{I}$ be a partial interpretation for S. For any clause $C \in S$, if $v_{\mathscr{I}}(C) = T$, the partial interpretation $\mathscr{I}$ *satisfies* C, and if $v_{\mathscr{I}}(C) = F$, C is a *conflict clause* for $\mathscr{I}$.

The existence of a conflict clause means that the given partial interpretation cannot possibly be extended to an interpretation that satisfies the entire set of clauses.

Example: Let $S = \{pqr, \overline{p}q, \overline{qr}, r\}$, and let $\mathscr{I}_{q\overline{r}}$ be the partial interpretation that assigns T to q and F to r. Then $\mathscr{I}_{q\overline{r}}$ satisfies the first three clauses in S, but the fourth, r, is a conflict clause for $\mathscr{I}_{q\overline{r}}$. □

In addition to search, *unit resolution* will be performed to infer necessary features of a satisfying interpretation. Suppose that the set of clauses S contains a unit clause l (a clause containing only the single literal l). Then any satisfying interpretation must assign T to l. That fact being noted, the unit clause l can be deleted, as can any instances of its complement l^c belonging to other clauses. Unit resolution is also referred to as *unit propagation*, or as *Boolean constraint propagation*.

Algorithm 6.20: (DPLL algorithm)

Input: A formula A in clausal form.

Output: A report that A is unsatisfiable, or a report that A is satisfiable, together with a partial interpretation that satisfies A.

Call the procedure DPLL(B, $\mathscr{I}$) recursively. It takes two arguments – a formula B in clausal form, and a partial interpretation $\mathscr{I}$.

Initialize B as A and $\mathscr{I}$ as the empty partial interpretation.

$DPLL(B, \mathscr{I})$:

Unit resolution: Construct the set of clauses B' by performing unit resolution as much as possible. Construct $\mathscr{I}'$ by adding to $\mathscr{I}$ all the assignments made during unit resolution.

Check satisfaction: Evaluate B' under the partial interpretation $\mathscr{I}'$:

- if B' contains a conflict clause, return 'unsatisfiable';
- if B' is satisfied, return $\mathscr{I}'$;
- otherwise, continue.

Branching: Choose an atom p in B', and a truth value val as T or F; let $\mathscr{I}_1$ be the interpretation $\mathscr{I}'$ together with the assignment of val to p.

- $result \leftarrow DPLL(B', \mathscr{I}_1)$.
- If $result$ is not 'unsatisfiable,' return $result$.
- otherwise, continue.

Backtracking: $\mathscr{I}_2$ is the partial interpretation $\mathscr{I}'$, together with the assignment of the complement of val to p.

- $result \leftarrow DPLL(B', \mathscr{I}_2)$.
- return $result$.

Note: when DPLL calls itself, the set of clauses B' should be understood as having been simplified to reflect the assignment of val or its complement to the chosen atom p – specifically, if the truth value of p is T, then clauses containing p should be deleted, and occurrences of $\overline{p}$ should be deleted from clauses; and vice versa if the truth value of p is F.

An example of the DPLL algorithm

Consider an example that will be revisited when we discuss ways of improving the DPLL algorithm:

$$pq, qr, \overline{ps}t, \overline{p}su, \overline{p}\overline{t}u, \overline{p}s\overline{u}, \overline{psu} \ .$$

If this is A, then the recursive procedure DPLL will initially be called with this set of clauses and the empty partial interpretation.

To step through a possible execution of the algorithm, we keep track of the parameters used in the respective calls to DPLL. For each call, we list the last addition or modification to the partial interpretation, followed by the set of clauses. Indentation is used to reflect the depth of nesting of calls to the procedure:

$$\begin{array}{lr}
\emptyset: & pq, qr, \overline{ps}t, \overline{p}su, \overline{p}\overline{t}u, \overline{p}s\overline{u}, \overline{psu} \\
\quad p \mapsto T: & qr, \overline{s}t, su, \overline{t}u, s\overline{u}, \overline{su} \\
\quad\quad q \mapsto T: & \overline{s}t, su, \overline{t}u, s\overline{u}, \overline{su} \\
\quad\quad\quad s \mapsto T: & t, \overline{t}u, \overline{u} \\
\quad\quad\quad s \mapsto F: & u, \overline{t}u, \overline{u} \\
\quad\quad q \mapsto F: & r, \overline{s}t, su, \overline{t}u, s\overline{u}, \overline{su} \\
\quad\quad\quad s \mapsto T: & t, \overline{t}u, \overline{u} \\
\quad\quad\quad s \mapsto F: & u, \overline{t}u, \overline{u} \\
\quad p \mapsto F: & q, qr \\
\text{return } p \mapsto F, q \mapsto T &
\end{array}$$

It is unfortunate that the algorithm goes over the same ground after branching and after backtracking on q, the reason being that q is unrelated to the majority of the clauses. In section 6.5, we'll discuss ways of avoiding this behaviour.

6.4 An extended example of the DPLL algorithm

See the textbook for the 'four-queens' example. It illustrates in particular:

- the encoding of a constraint-satisfaction problem as an instance of SAT;
- the extent to which the number of clauses may drop as truth values are assigned to atoms.

6.5 Improving the DPLL algorithm

Branching heuristics

Various heuristics have been developed to guide the choice of which atom to "branch on." We'll focus on other ways of improving the algorithm.

The pure-literal rule

Suppose that a set of clauses contains occurrences only of a literal l, and not its complement, l^c. Then there is no point in considering the case where l has the value F. But if l has the value T, any clause containing l can be deleted.

Such a literal is called a *pure literal* (in the context of the given set of clauses), and the deletion of clauses that contain it is called an application of the *pure-literal rule*. The pure-literal rule can be applied immediately after the unit-resolution step, and the assignment $\mathscr{I}'$ can then include also the assignment that makes the pure literal true.

Non-chronological backtracking and clause learning

On finding the current set of clauses unsatisfiable under the current partial interpretation, the DPLL algorithm always backtracks to the nearest ancestor in the search tree at which an atom was chosen to "branch on," and only one of its possible truth values has so far been considered.

But in some cases it can be determined that it is pointless to try the other possible truth value of that atom, and that the algorithm could more efficiently jump further back.

Suppose that the set of clauses B' is

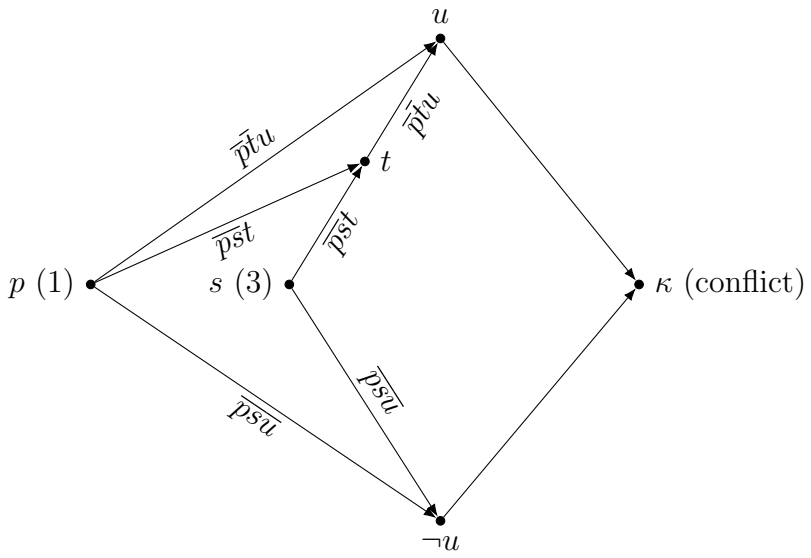
$$pq, qr, \overline{p}\overline{s}t, \overline{p}su, \overline{p}\overline{t}u, \overline{p}s\overline{u}, \overline{p}\overline{s}\overline{u} \ .$$

As was seen in our earlier analysis of this example, this set of clauses is satisfied by any partial interpretation that assigns F to p and T to q. But suppose that the truth value of p is set to T; then, in the next call to $DPLL$, when B' is evaluated with respect to the corresponding partial interpretation, the first of the above clauses is deleted, and $\overline{p}$ is deleted from the remaining clauses:

$$qr, \overline{s}t, su, \overline{t}u, s\overline{u}, \overline{s}\overline{u} \ .$$

This set of clauses is unsatisfiable – because, in particular, the subset of clauses involving the atoms s, t, and u is unsatisfiable. So whatever truth values might be assigned to q or r, the resulting sets of clauses will be found to be unsatisfiable. But the DPLL algorithm will not recognize that fact, and may choose to branch on q and r, in which case it will end up backtracking and effectively determining multiple times that the subset of clauses in s, t, and u is unsatisfiable. This was illustrated in our earlier discussion.

But the algorithm can be modified with the use of an *implication graph*, which makes explicit the causes of conflicts:



Here, nodes without incoming edges are labelled by literals assigned the value T during branching or backtracking; the number in parentheses indicates the *decision level* at which that assignment occurred. (The decision level corresponds to the level of indentation used in our earlier description of a run of the algorithm.)

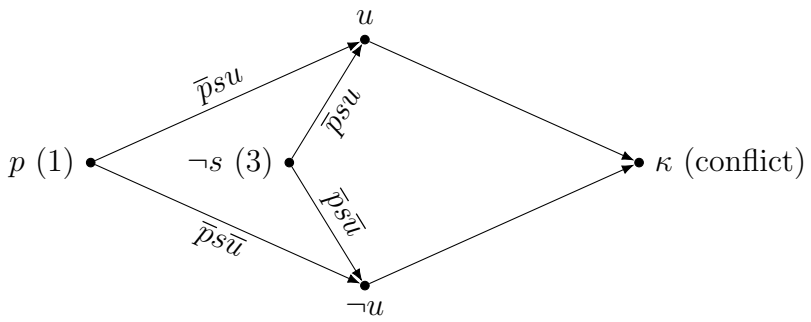
Given the truth values of those literals, the satisfaction of the various clauses implies the truth of other literals. Edges lead to nodes labelled by those literals, the edges themselves labelled by the clauses involved in the implication.

This particular implication graph shows that a conflict follows from the assignment of T to p at decision level 1, and the assignment of T to s two

branching steps later, at decision level 3; and that intervening branching on q or r (at level 2) does not enter into the conflict.

In order to avoid encountering the same conflict again, a suitable clause can be *learned* by the SAT solver: the negation of $p \wedge s$ is, by De Morgan's laws, $\neg p \vee \neg s$; by adding this clause to the set of clauses, it will be ensured that there is a conflict clause for the partial interpretation that gives rise to the conflict.

Because of the conflict, the DPLL algorithm will backtrack to the latest decision level (3), and assign the value F to s. That assignment results in a new implication graph:



Here again, a conflict arises. At this point, the DPLL algorithm would normally backtrack to decision level 2, and try a new truth value for q or r, but this implication graph shows that the conflict does not involve q or r, but is rather a consequence of the assignment of T to p at decision level 1; indeed, the learned clause $\overline{ps}$ ensures that that assignment to p also determines the assignment of F to s, and ensures the existence of the conflict. On the basis of the implication graph, a SAT solver can therefore add the further learned clause $\neg(p \wedge \neg s) \equiv (\neg p \vee s) \equiv \overline{p}s$ to the set of clauses, and also jump directly back to decision level 1, and assign the value F to p. Such backtracking by more than a single decision level is called *backjumping*, or *non-chronological backtracking*. It prevents the SAT solver from wasting time trying different assignments to the atoms q and r that don't play any role in the conflict.

Once F is assigned to p, unit resolution quickly results in the empty set of clauses, showing the original set to be satisfiable.

SAT solvers that incorporate clause learning and non-chronological backtracking are called *Conflict-Driven Clause-Learning (CDCL)* SAT solvers.

Restarts and forgets

Learned clauses accumulate over time, so some SAT solvers incorporate *forgetting* in order to limit the amount of memory that they occupy.

Solvers may sometimes spend inordinate amounts of time seeking solutions, by focusing on 'hard' parts of the search space. Many have a *restarting* feature that can curtail long searches, at the cost of no longer being guaranteed to find a solution.

Free SAT solvers

Free, open-source SAT solvers include MiniSat and MicroSat – the latter consisting of less than 250 lines of C code. Z3, from Microsoft Research, is more general: it incorporates a SAT solver, but is also a theorem-prover for fragments of predicate logic. It can be found on the website `rise4fun.com`.