CDM

Satisfiability

Klaus Sutner
Carnegie Mellon University

14-satisfiability 2017/12/15 23:21

CC BY NC SA

SAT Algorithms

There is an old, but surprisingly powerful satisfiability testing algorithm due to Davis and Putnam, originally published in 1960.

The Key Papers

P. C. Gilmore
A proof method for quantification theory: its justification and realization
IBM J. Research and Development, 4 (1960) 1: 28–35

M. Davis, H. Putnam
A Computing Procedure for Quantification Theory
Journal ACM 7 (1960) 3: 201–215.

M. Davis, G. Logemann, D. Loveland
A Machine Program for Theorem Proving
Communications ACM 5 (1962) 7: 394–397.

The Real Prey

Note the titles: the real target was an algorithm to establish validity in first-order logic. Recall that valid (aka provable) formulae in FOL are only semidecidable, but not decidable. So the challenge is to find computationally well-behaved methods that can identify at least some valid formulae.

Gilmore and Davis/Putnam exploit a theorem by J. Herbrand:

- To show that φ is valid, show that $\neg\varphi$ is inconsistent.
- Translate $\neg\varphi$ into a set of clauses Γ.
- Enumerate potential counterexamples based on Herbrand models, stop if one is found.

The last step requires what is now called a SAT solver.

Tiny Example

$$\varphi \equiv P(a) \wedge \forall x\,(P(x) \Rightarrow Q(f(x))) \Rightarrow Q(f(a))$$

$$\neg\varphi \equiv P(a) \wedge \forall x\,(P(x) \Rightarrow Q(f(x))) \wedge \neg Q(f(a))$$

$$\equiv \forall x\,\big(P(a) \wedge (P(x) \Rightarrow Q(f(x))) \wedge \neg Q(f(a))\big)$$

$$\Gamma = \{P(a), \neg P(x) \vee Q(f(x)), \neg Q(f(a))\}$$

Try substitution $x = a$:

$$\Gamma = \{P(a), \neg P(a) \vee Q(f(a)), \neg Q(f(a))\}$$

$$\Gamma_0 = \{p, \neg p \vee q, \neg q\}$$

Gilmore's Abstract

A program is described which can provide a computer with quick logical facility for syllogisms and moderately more complicated sentences. The program realizes a method for proving that a sentence of quantification theory is logically true. The program, furthermore, provides a decision procedure over a subclass of the sentences of quantification theory. The subclass of sentences for which the program provides a decision procedure includes all syllogisms. Full justification of the method is given.

A program for the IBM 704 Data Processing Machine is outlined which realizes the method. Production runs of the program indicate that for a class of moderately complicated sentences the program can produce proofs in intervals ranging up to two minutes.

The "Multiplication" Method

Unfortunately, Gilmore's method to check satisfiability of a propositional formula ψ comes down to this:

- Transform ψ into disjunctive normal form.
- Remove all conjuncts containing x and .
- If nothing is left, report success.

DOA.

The DPLL Idea

The basic idea of the DPLL solver is beautifully simple. Assume that the input is in conjunctive normal form.

- Repeatedly apply simple cleanup operations, until nothing changes.
- Bite the bullet: pick a variable and explicitly set it True and False, respectively.
- Backtrack.

As the authors point out, their method yielded a result in a 30 minute hand-computation, where Gilmore's algorithm running on an IBM 704 failed after 21 minutes. The variant presented below was first implemented by Davis, Logeman and Loveland in 1962 on an IBM 704.

Naive Algorithm

Here is the most basic recursive approach to SAT testing (in reality backtracking). We are trying to build a truth-assignment σ, initially σ is totally undefined.

- If every clause is satisfied, then return True.
- If some clause is false, then return False.
- Pick any unassigned variable x.
  - Set $\sigma(x) = 0$. If Γ now satisfiable, return True.
  - Set $\sigma(x) = 1$. If Γ now satisfiable, return True.
- Return False.

Three-Valued Logic

During the execution of the algorithm variables are either unassigned, true or false; they change back and forth between these values.

Strictly speaking, this is best expressed in terms of a three-valued logic with values $\{0, 1, ?\}$.

One has to redefine the Boolean operations to deal with unassigned variables. For example

$\wedge$	0	?	1
0	0	0	0
?	0	?	?
1	0	?	1

In practice, no one bothers.

Streamlining Things

Obviously, it is a bad idea to pick x blindly for the recursive split.

Moreover, one should do regular cleanup operations to keep Γ small.

There are two simple yet surprisingly effective methods:

- Unit Clause Elimination
- Pure Literal Elimination

Unit Clauses

A clause is a unit clause iff it contains just one literal.

Clearly, if $\{x\} \in \Gamma$ any satisfying truth-assignment σ must have $\sigma(x) = 1$. But then we can remove clause $\{x\}$ and do a bit of surgery on the rest, without affecting satisfiability.

This is also called Boolean constraint propagation (BCP). SAT solvers spend a lot of time dealing with constraint propagation.

Unit Clause Elimination (UCE)

Unit Subsumption: delete all clauses containing x, and

Unit Resolution: remove from all remaining clauses.

This process is called unit clause elimination.

Let $\{x\}$ be a unit clause in Γ and write Γ' for the resulting set of clauses after UCE for clause $\{x\}$.

Proposition

Γ and Γ' are equisatisfiable.

Pure Literal Elimination (PLE)

Here is another special case that is easily dispatched.

A pure literal in Γ is a literal that occurs only directly, but not negated. So the formula may either contain a variable x or its negation , but not both.

Clearly, we can accordingly set $\sigma(x) = 1$ (or $\sigma(x) = 0$) and remove all the clauses containing the literal.

This may sound pretty uninspired but turns out to be useful in the real world. Note that in order to do PLE efficiently we need to keep counters for the number of occurrences of both x and .

More on PLE

Here is a closer look at PLE. Let Γ be a set of clauses, x a variable. Define

- Γ: the clauses of Γ that contain x positively,
- Γ: the clauses of Γ that contain x negatively, and
- Γ: the clauses of Γ that are free of x.

So we have the partition

$$\Gamma = \Gamma \cup \Gamma \cup \Gamma$$

Note that UCE produces $\Gamma' = \{\, C- \mid C \in \Gamma \,\} \cup \Gamma$.

PLE lemma

Proposition

If Γ or Γ is empty, then Γ and Γ are equisatisfiable.

Since Γ is smaller than Γ (unless x does not appear at all), this transformation simplifies the decision problem.

But note that PLE flounders once all variables have positive and negative occurrences. If, in addition, there are no unit clauses, we are stuck.

The DPLL Algorithm

- Unit Clause Elimination: do UCE until no unit clauses are left.
- Pure Literal Elimination: do PLE until no pure literals are left.
- If an empty clause has appeared, return False.
- If all clauses have been eliminated, return True.
- Splitting: otherwise, cleverly pick one of the remaining literals, x. Backtrack to test **both**

  $$\Gamma, \{x\} \qquad \text{and} \qquad \Gamma, \{\}$$

  for satisfiability.

  Return True if at least one of the branches returns True; False otherwise.

For Glass-Half-Empty People

Note that UCE may well produce more unit clauses as well as pure literals, so the first two steps hopefully will shrink the formula a bit.

Still, thanks to Splitting, this looks dangerously close to brute-force search.

The algorithm still succeeds beautifully in the Real World, since it systematically exploits all possibilities to prune irrelevant parts of the search tree.

Example

After three UCE steps (no PLE) and one split on d we get the answer "satisfiable":

```
1    {a,b,c} {a,!b} {a,!c} {c,b} {!a,d,e} {!b}
2    {a,c}           {a,!c} {c}   {!a,d,e}
3                     {a}         {!a,d,e}
4                                 {d,e}
```

We could also have used PLE (on d):

```
1    {a,b,c} {a,!b} {a,!c} {c,b} {!a,d,e} {!b}
2    {a,b,c} {a,!b} {a,!c} {c,b}           {!b}
3                           {c,b}           {!b}
4                           {c}
```

Finding an Assignment

This algorithm also solves the search problem: we only need to keep track of the assignments made to literals. In the example, the corresponding assignment is

$$\sigma(b) = 0, \sigma(c) = \sigma(a) = \sigma(d) = 1$$

The choice for e does not matter.

Note that we also could have chosen $\sigma(e) = 1$ and ignored d.

Exercise

Implement a version of the algorithm that returns a satisfying truth assignment if it exists.

How about all satisfying truth assignments?

Correctness

Claim

The Davis/Putnam algorithm is correct: it returns true if, and only if, the input formula is satisfiable.

Proof.

We already know that UCE and PLE preserve satisfiability. Let x be any literal in φ. Then by Boole-Shannon expansion

$$\varphi(x, \boldsymbol{y}) \equiv (x \wedge \varphi(1, \boldsymbol{y})) \vee (\wedge\varphi(0, \boldsymbol{y}))$$

But splitting checks exactly the two formulae on the right for satisfiability; hence φ is satisfiable if, and only if, at least one of the two branches returns true.

Termination is obvious.

□

Bad News

One can think of DPLL as a particular kind of resolution (see next section). Unfortunately, it inherits potentially exponential running time as shown by Tseitin in 1966.

Intuitively, this is not really surprising: too many splits will kill efficiency, and DPLL has no clever mechanism of controlling splits.

And there is the Levin-Cook theorem that shows that SAT is $\mathbb{NP}$-complete, so one should not expect algorithmic miracles.

Good News

In practice, though, Davis/Putnam is usually quite fast, even for huge formulae.

It is not entirely understood why formulae that appear in real-world problems tend to produce something like polynomial running time when tackled by DPLL.

Take the restriction to "real world" problems here with a grain of salt. For example, in algebra, DPLL has been used to solve problems in the theory of so-called quasi groups (cancellative groupoids). In a typical case, there are n^3 Boolean variables and about n^4 to n^6 clauses; n might be 10 or 20.

Example: Exactly One

Neither UCE nor PLE applies here, so the first step is a split.

```
{{!a,!b},{!a,!c},{!a,!d},{!a,!e},{!b,!c},{!b,!d},{!b,!e},{!c,!d},
  {!c,!e},{!d,!e},{a,b,c,d,e}}

{{!a},{!a,!b},{!a,!c},{!a,!d},{!a,!e},{!b,!c},{!b,!d},{!b,!e},
  {!c,!d},{!c,!e},{!d,!e},{a,b,c,d,e}}

{{!b},{!b,!c},{!b,!d},{!b,!e},{!c,!d},{!c,!e},{!d,!e},{b,c,d,e}}

{{!c},{!c,!d},{!c,!e},{!d,!e},{c,d,e}}

{{d},{d,e},{!d,!e}}

True
```

Of course, this formula is trivially satisfiable, but note how the algorithm quickly homes in on one possible assignment.

The Real World

If you want to see some cutting edge problems that can be solved by SAT algorithms (or can't quite be solved at present) take a look at

`http://www.satcompetition.org`

`http://www.satlive.org`

Try to implement DPLL yourself, you will see that it's hopeless to get up to the level of performance of the programs that win these competitions.

Bookkeeping

We pretended that literals are removed from clauses: in reality, they would simply be marked False. In this setting, a unit clause has all but one literals marked False.

Similarly, if every clause has true literal, then the algorithm returns True. And, if some clause has only false literals, then it returns False.

So one should keep count of non-false literals in each clause. And one should know where a variable appears positively and negatively.

At present, it seems that lean-and-mean is the way to go with SAT solvers. Keeping track of too much information gets in the way.

Splitting

There are several strategies to choose the next variable in a split. Note that one also needs to determine which truth value to try first.

- Hit the most (unsatisfied) clauses.
- Use most frequently occurring literal.
- Focus on small clauses.
- Do everything at random.

Watch Pointers

Here is a clever hack that minimizes the number of times a clause needs to be inspected (after the algorithm has performed one of its basic steps).

- For every clause, keep pointers to two unassigned literals.
- For each variable, keep track of watched clauses (positive and negative).

The key idea: examine a clause only when one of its watched literals is assigned False.

Watching

Suppose literal is assigned True and let C be a clause on the watch list for x.

$k = \#$ literals in C $\quad \ell = \#$ false literals in C $\quad \ell' = \#$ true literals in C

- If $\ell = k$: return False.
- If $0 < \ell'$: return True.
- If $\ell < k$: check for UCE.
- Otherwise: update pointers and watch lists.

The additional bookkeeping is more than compensated for by cutting down on the number of inspected clauses.

Looking for Contradictions

One key step in the Davis/Putnam approach is to translate validity into un-satisfiability of the negation of a given formula. In the context of propositional logic, this leads to the question of how one can identify contradictions easily (obviously, within complexity theoretic constraints).

Challenge: How does one check for contradictions?

Of course, all these problems are equivalent in a sense, but we are looking for practical algorithms.

Refutation

J. A. Robinson
A Machine-Oriented Logic Based on the Resolution Principle
Journal ACM, 12 (1965) 1: 23–41.

As before we assume that Γ is given in CNF. Then we try to show that these clauses are inconsistent.

Resolvents

Suppose x is a variable that appears in clause C and appears negated in clause C', where $C, C' \in \Gamma$:

$$C = \{x, y_1, \ldots, y_k\} \qquad C' = \{, z_1, \ldots, z_l\}$$

Then we can define an new clause, a resolvent of C and C' by

$$D = \{y_1, \ldots, y_k, z_1, \ldots, z_l\}$$

Proposition

$C \wedge C'$ is equivalent to $C \wedge C' \wedge D$.

We write $(C, C') = D$, but note that there may be several resolvents.

Resolution

The CNF formula

$$\varphi = \{\{x, , \}, \{, \}, \{, z\}, \{y\}\}$$

admits the following ways to compute resolvents:

$$\begin{aligned} (\{x, , \}, \{, \}) &= \{, \} \\ (\{, z\}, \{y\}) &= \{z\} \\ (\{, \}, \{z\}) &= \{\} \\ (\{y\}, \{\}) &= \emptyset \end{aligned}$$

This iterative computation of resolvents is called resolution.

The last resolvent corresponds to the empty clause, indicating that the original formula is not satisfiable.

(Bad) Notation

It is a near sacred principle that in the context of resolution methods one writes the empty clause thus:

$$\square$$

Yup, that's a little box, like the end-of-proof symbol or the necessity operator in modal logic.

As we will see shortly, $\square$ is a resolvent of a set of clauses Γ if, and only if, the corresponding CNF formula is a contradiction.

Resolution, Defined

More precisely, given a collection of clauses Γ, let (Γ) be the collection of all resolvents of clauses in Γ, including Γ itself. Set

$$(\Gamma) = \bigcup_n^n (\Gamma)$$

so (Γ) is the least fixed point of the resolvent operator applied to Γ.

We will show that

Lemma

Γ is a contradiction if, and only if, $\square \in (\Gamma)$.

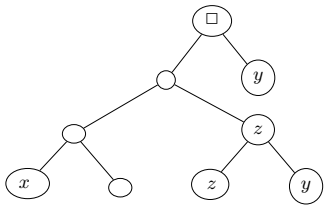
DAG Perspective

One often thinks of a resolution proof for the un-satisfiability of Γ as a directed, acyclic graph G whose nodes are clauses. The following degree conditions hold:

- The clauses of Γ have indegree 0.
- Each other node has indegree 2 and corresponds to a resolvent of the two predecessors.
- There is one node with outdegree 0 corresponding to the empty clause.

Strictly speaking, the graph is a DAG, not a tree, but we can duplicate nodes to unfold it into a tree.

Example



The Logical System

We can think of resolution as a proof system based on rules of inference:

$$\text{resolution} \quad \frac{x \vee \varphi \qquad \vee \psi}{\varphi \vee \psi}$$

$$\text{contraction} \quad \frac{\varphi \vee \varphi}{\varphi}$$

$$\text{subsumption} \quad \frac{\varphi \vee \psi \quad \varphi}{\varphi}$$

We are ignoring associativity, commutativity, substitutivity.

This system is complete in the sense that it derives $\square$ from any contradiction.

Analyzing Resolution

As usual, there are three issues we have to address:

Correctness: any formula with resolvent $\square$ is a contradiction.

Completeness: any contradiction has $\square$ as a resolvent.

Effectiveness: a resolution proof can be generated effectively.

Note that for a practical algorithm the last condition is actually a bit too weak: we would like a reasonable performance guarantee. Alas ...

On the upside, note that we do not necessarily have to compute all of (Γ): if $\square$ pops up we can immediately terminate.

Correctness

Lemma

For any truth-assignment σ we have

$$\sigma(C) = \sigma(C') = 1 \quad \textit{implies} \quad \sigma((C, C')) = 1$$

Proof.

If $\sigma(y_i) = 1$ for some i we are done, so suppose $\sigma(y_i) = 0$ for all i.

Since σ satisfies C we must have $\sigma(x) = 1$. But then $\sigma() = 0$ and thus $\sigma(z_i) = 1$ for some i.

Hence σ satisfies (C, C'). □

It follows by induction that if σ satisfies Γ it satisfies all resolvents of Γ.

Hence resolution is correct: only contradictions will produce □.

Completeness

Theorem

Resolution is complete.

Proof.

We have to show that if Γ is a contradiction, then $\square \in (\Gamma)$.

Proof is by induction on the number n of variables in Γ.

$n = 1$: Then $\Gamma = \{x\}, \{\}$.

In one resolution step we obtain □. Done.

Proof contd.

Assume $n > 1$ and let x be a variable.

Let Γ_0 and Γ_1 be obtained by performing UCE for $\Gamma, \{x\}$ and $\Gamma, \{\}$, respectively.

Note that both Γ_0 and Γ_1 must be contradictions.

Hence, by induction hypothesis, $\square \in (\Gamma_i)$.

Now the crucial step: by repeating the "same" resolution proof with Γ rather than Γ_i, $i = 0, 1$, we get $\square \in (\Gamma)$ if this proof does not use any of the mutilated clauses.

Otherwise, if mutilated clauses are used in both cases, we must have

- $\{x\} \in (\Gamma)$ from Γ_1, and
- $\{\} \in (\Gamma)$ from Γ_0.

Hence $\square \in (\Gamma)$. □

A Prototype Algorithm

It is clear that we would like to keep the number of resolvents introduced in the resolution process small. Let's say that clause ψ subsumes clause φ if $\psi \subseteq \varphi$: ψ is at least as hard to satisfy as φ.

We keep a collection of "used" clauses U which is originally empty. The algorithm ends when Γ is empty.

- Pick a clause ψ in Γ and move it to U.
- Add all resolvents of ψ and U to Γ except that:
  - Tautology elimination: delete all tautologies.
  - Forward subsumption: delete all resolvents that are subsumed by a clause.
  - Backward subsumption: delete all clauses that are subsumed by a resolvent.

Efficiency

So how large does a resolution proof have to be, even one that uses clever tricks like the subsumption mechanism?

Can we find a particular problem that is particularly difficult for resolution?

Recall that there is a Boolean formula ${}_k(x_1, \ldots, x_k)$ of size $\Theta(k^2)$ such that σ satisfies ${}_k(x_1, \ldots, x_k)$ iff σ makes exactly one of the variables $x_1, \ldots, x_k$ true.

$${}_k(x_1, \ldots, x_k) = (x_1 \vee x_2 \ldots \vee x_k) \wedge \bigwedge_{1 \le i < j \le k} \neg(x_i \wedge x_j)$$

Note that formula is essentially in CNF.

Das Dirichletsche Schubfachprinzip

Lemma (Pigeonhole Principle)

There is no injective function from $[n+1]$ to $[n]$.

This is, of course, totally obvious, but a formal proof in a system like Peano arithmetic requires a bit of work. The classical proof is by induction on n.

Alternatively, we could translate the PHP into a Boolean formula (for any particular value of n, not in the general, parametrized version).

Idea: Variable x_{ij} is true iff pigeon i sits in hole j (or, in less ornithological language, $f(i) = j$).

Pigeonhole Principle

We need a formula that expresses PHP in terms of these variables.

Given variables x_{ij} where $1 \leq i \leq m$ and $1 \leq j \leq n$ define

$$\Gamma_{mn} = \bigwedge_{i \leq m} {}_n(x_{i1}, x_{i2}, \ldots, x_{in})$$

Then Γ_{mn} is satisfiable iff $m \leq n$.

In particular we ought to be able to use resolution to show that $\Gamma_{n+1,n}$ is a contradiction.

Exponential Lower Bound

By completeness there must be a resolution proof showing that $\Gamma_{n+1,n}$ is a contradiction.

But:

Theorem

Every resolution proof for the contradiction $\Gamma_{n+1,n}$ has exponential length.

Proof is quite hairy.

Easy Cases

One might wonder if there is perhaps a special class of formulae where a resolution type approach is always fast.

We can think of a clause

$$\{{}_1, {}_2, \ldots, {}_r, y_1, y_2, \ldots, y_s\}$$

as an implication:

$$x_1 \wedge x_2 \wedge \ldots \wedge x_r \Rightarrow y_1 \vee y_2 \vee \ldots \vee y_s$$

When $s = 1$ these implication are particularly simple.

Horn Formulae

Definition

A formula is a Horn formula if it is in CNF and every clause contains at most one un-negated variable.

Example:

$$\varphi = \{\{, , z\}, \{, \}, \{x\}\}$$

or in implicational notation

$$\varphi = \{x \wedge y \Rightarrow z, y \wedge z \Rightarrow \bot, x\}$$

Horn Clauses

In other words, a Horn formula has only Horn clauses, and a Horn clause is essentially an implication of the form

$$C = x_1 \wedge x_2 \wedge \ldots \wedge x_n \Rightarrow y$$

where we allow $y = \bot$.

We also allow single un-negated variables (if you like: $\top \Rightarrow x$).

Note that if we have all unit clauses x_i, then a resolvent of these and C will be y.

This gives rise to the following algorithm.

Marking Algorithm

Testing Horn formulae for Satisfiability

- Mark all variables x in unit clauses $\{x\}$.
- If there is a clause $x_1 \wedge x_2 \wedge \ldots \wedge x_n \Rightarrow y$ such that all the x_i are marked, mark y. Repeat till a fixed point is reached.
- If $\bot$ is ever marked, return No.
- Otherwise, return Yes.

Hallelujah, it's DFS (actually: graph exploration).

Note that the Marking Algorithm is linear in the size of Γ.

We can read off a satisfying truth-assignment if the formula is satisfiable:

$$\sigma(x) = \begin{cases} 1 & x \text{ is marked,} \\ 0 & \text{otherwise.} \end{cases}$$

Then $\sigma(\Gamma) = 1$.

Moreover, $\tau(\Gamma) = 1$ implies that

$$\forall\, x\, (\tau(x) \leq \sigma(x))$$

so σ is the "smallest" satisfying truth-assignment.