UCT

Satisfiability

Klaus Sutner

Carnegie Mellon University

Spring 2022

Where Are We?

- We have some machinery to compare the complexity of decision problems:
  complexity classes, reductions, hardness, completeness.

- In the classical theory, we have a nice hierarchy of increasingly unsolvable problems.

- We are currently thinking of $\mathbb{P}$ as a first-order approximation to the notion of "feasible computation."

- We would like to build a similar complexity hierarchy around $\mathbb{P}$.

Again, the trick will be to compare problems to each other, rather than direct lower bounds.

> We may ask, of a given problem P,
>
> *If we could solve P, what else could we solve?*
>
> And, we may ask,
>
> *The solutions to which problems would also furnish solutions to P?*

M. Davis 1958

Entscheidungsproblem to the Rescue

Recall the Entscheidungsproblem? A decision algorithm for all of math? Another great source for difficult problems.

> In a sense, [the Entscheidungsproblem] is the most general problem of mathematics.
>
> J. Herbrand

Right, the most general and, as we now know, absolutely, hopelessly, mind-numbingly unsolvable.

But: we could try to turn adversity into an asset, and use some tamer version of the Entscheidungsproblem as a template of a hard problem.

The original Entscheidungsproblem would have included arbitrary first-order questions about number theory. This would indeed be very difficult, a problem not even located in arithmetical hierarchy.

We need something far less ambitious and closer to real algorithms.

So where would be look for hard problems, something that is eminently decidable but appears to be outside of $\mathbb{P}$? And, we'd like the problem to be practical, not some monster from CRT.

The Circuit Value Problem is a good indicator for the right direction: evaluating Boolean expressions is polynomial time, but relatively difficult within $\mathbb{P}$.

So can we push CVP a little bit to force it outside of $\mathbb{P}$, just a little bit? Say, up into EXP_1?

Scaling Back

Taking a clue from CVP, how about asking questions about propositional logic, rather than first-order logic?

Probably the most natural question that comes to mind here is

> Is $\varphi(x_1, \ldots, x_n)$ a tautology?

where φ is a Boolean formula, with variables $x_1, \ldots, x_n$.

For example, $p \wedge (p \Rightarrow q) \Rightarrow q$ is a tautology.

Close, but no cigar.

For technical reasons, it is better to ask the very similar question

> Is $\varphi(x_1, \ldots, x_n)$ satisfiable?

Obviously, φ is a tautology iff $\neg\varphi$ fails to be satisfiable, so nothing is lost.

But, as we will see, satisfiability is slightly better behaved than tautology if one is concerned about resource bounds. This has to do with convenient normal forms: φ may be in normal form (such as CNF), but $\neg\varphi$ is not.

Problem:	**Satisfiability (SAT)**
Instance:	A Boolean formula $\varphi(x_1, \ldots, x_n)$.
Question:	Is φ satisfiable?

The difficulty here comes from the fact that there are 2^n possible truth assignments $\sigma : \mathsf{Var} \to \mathbf{2}$. Even though we can evaluate $\varphi[\sigma]$ in linear time, any algorithm using brute-force search will be exponential, something like $2^n \mathrm{poly}(n)$.

Of course, that does not mean that there is no better algorithm, brute-force is just the most obvious line of attack. It would also work for Tautology.

If you think of a Boolean formula as the kind to little thingy you encountered in 151, they might seem pretty feeble. For example

$$(p \Rightarrow q) \Rightarrow (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$$

is clearly a tautology (the infamous cut rule). It is quite useful as a logical axiom in a formal system, though. In fact, all the logical axioms in any of the standard systems are obviously tautologies.

But: what if the formula has 10000 variables, and takes a megabyte of memory just to write it down? Your intuition will tell you zip about a monster like that. Unfortunately, big formulae are where the action is.

Example 1: VC to SAT

As a warm-up exercise showing off the expressiveness of SAT, we will show how to translate the Vertex Cover problem into a satisfiability problem.

Problem:	**Vertex Cover**
Instance:	A ugraph G, a bound k.
Question:	Does G have a vertex cover of size k?

For any translation to SAT, it is critical to interpret the Boolean variables the right way.

Let's assume G looks like $\langle [n], E \rangle$. It seems natural to introduce n Boolean variables

$$p_x \qquad 1 \leq x \leq n,$$

one for each vertex x.

The idea is simply that

$\sigma \models p_x \iff x$ is in the alleged cover

So the truth assignment σ is just a bitvector for the set
$C_\sigma = \{\, x \mid \sigma \models p_x \,\} \subseteq V$.

We need to construct a formula $\Phi_{G,k}$ that enforces this interpretation. Let's ignore the cardinality part for the moment. Every edge needs to have at least one endpoint in the alleged cover C_σ:

$$\bigwedge_{(u,v) \in E} p_u \vee p_v$$

This conjunction has size $O(n^2)$, so we are good.

Counting

We also need to make sure that $|C_\sigma| = k$.

Write $\mathsf{CNT}_{r,s}(p_1, p_2, \ldots, p_r)$ for a formula that is true under σ iff exactly s of the r variables are true.

We could simply add $\mathsf{CNT}_{n,k}(p_1, p_2, \ldots, p_n)$ to Φ_G and be done.

Easy, what could possibly go wrong?

Problems with Reductions

To establish a reduction from A to B we need to avoid three possible errors:

- Logical correctness: we must have $x \in A \Leftrightarrow f(x) \in B$.
- Computational simplicity: f must be easy to compute.
- Size constraint: $f(x)$ must not be too long.

In the heat of battle, it's quite possible to screw up one of these issues.

The standard way to get a counting formula is to use threshold functions.

Definition

A threshold function thr^n_m, $0 \leq m \leq n$, is an n-ary Boolean function defined by

$$\mathrm{thr}^n_m(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \#(i \mid x_i = 1) \geq m, \\ 0 & \text{otherwise.} \end{cases}$$

So $\mathrm{thr}^n_m(\boldsymbol{x})$ simply means that at least m of the n variables are true.

Lots of Boolean functions can be defined in terms of threshold functions.

thr_0^n is the constant tt.

thr_1^n is n-ary disjunction.

thr_n^n is n-ary conjunction.

$\mathrm{thr}_k^n(\boldsymbol{x}) \wedge \neg \mathrm{thr}_{k+1}^n(\boldsymbol{x})$ is the counting function: $\mathsf{CNT}_{n,k}(x_1, x_2, \ldots, x_n)$, "exactly k out of n."

For example, $\mathsf{CNT}_{n,2}(\boldsymbol{x})$ looks like

$$\bigvee_{i<j}(x_i \wedge x_j) \quad \wedge \quad \neg \bigvee_{i<j<k} (x_i \wedge x_j \wedge x_k)$$

What would the formula $\mathsf{CNT}_{n,k}(x_1, x_2, \ldots, x_n)$ look like?

$$\bigvee_{I \in [n]_k} \bigwedge_{i \in I} x_i \quad \wedge \quad \neg \bigvee_{J \in [n]_{k+1}} \bigwedge_{i \in J} x_i$$

Here $[n]_k$ denotes all subsets of $[n]$ of cardinality k.

The notation used above makes it easy to write down the general counting formula, but it does obscure the actual size a bit. To determine size (say, the number of connectives) we need to expand out the disjunctions and conjunctions, we can only use the binary versions plus unary negation.

But that means that $\mathrm{CNT}_{n,k}$ has size something like $\binom{n}{k+1}$. This is not polynomial in n for variable k.

To be sure, it would work for fixed k, but that is not what the vertex cover problem asks.

Counting in Style

To keep the cardinality formula small, we introduce new variables

$$q_{i,j} \qquad 0 \leq i \leq n, 0 \leq j \leq k+1$$

with the intent that

$$q_{i,j} \Leftrightarrow \mathrm{thr}^i_j(p_1, \ldots, p_i)$$

We can determine the $q_{i,j}$ in a dynamic programming style, very much like an instance of CVP.

$$q_{i,0} = 1 \qquad i = 0, \ldots, n$$

$$q_{0,j} = 0 \qquad j = 1, \ldots, k+1$$

$$q_{i+1,j} = q_{i,j} \vee (q_{i,j-1} \wedge p_{i+1})$$

$$q_{n,k} \wedge \neg q_{n,k+1}$$

Total Damage

We get a formula $\Phi_{G,k}$ of size $O(n^2)$ (at least with uniform size function) that clearly can be constructed from G and k in polynomial time. A closer look shows that we can actually get away with just logarithmic space: all we need is a few loops over variables.

Moreover,

$$\sigma \models \Phi_{G,k} \iff C_\sigma \text{ is a vertex cover of size } k$$

and we have our translation to SAT.

Done.

Example 2: HC to SAT

Here is another translation, from Hamiltonian Cycle problem to Satisfiability.

Problem:	**Hamiltonian Cycle**
Instance:	A ugraph G.
Question:	Does G have a Hamiltonian cycle?

Again, it is critical to interpret the Boolean variables the right way.

As always, assume G looks like $\langle [n], E \rangle$. We introduce $n(n+1)$ Boolean variables

$$p_{t,x} \qquad 0 \leq t \leq n, 1 \leq x \leq n.$$

Think of t as time, and of x as location.

A Big Formula

The Idea: the Hamiltonian path we are looking for touches node x at time t iff $\sigma(p_{t,x}) = 1$ for a satisfying truth assignment σ.

So we need to construct a (large) Boolean formula Φ_G that enforces the following:

$$\sigma \models \Phi_G \iff \sigma \text{ codes a Hamiltonian cycle in } G$$

Then Φ_G is satisfiable iff G has a Hamiltonian cycle and we are done.

Of course, there is the constraint that Φ_G needs to be constructible from G in polynomial time.

Otherwise we could cheat and define $\Phi_G = \bot$ or $\Phi_G = \top$;-)

Φ_G is a conjunction with 4 parts as follows:

$$\bigwedge_t \mathsf{CNT}_{n,1}(p_{t,1}, p_{t,2}, \ldots, p_{t,n})$$

$$\bigwedge_{x \neq 1} \mathsf{CNT}_{n,1}(p_{1,x}, p_{2,x}, \ldots, p_{n,x})$$

$$p_{0,1} \wedge p_{n,1}$$

$$\bigwedge_{t,x} p_{t,x} \Rightarrow \bigvee_{y \in \Gamma_x} p_{t+1,y}$$

Here $\Gamma_x = \{\, y \in [n] \mid xy \in E \,\}$ denotes the neighborhood of x in G.

Since $\mathsf{CNT}_{m,1}(x_1, \ldots, x_m)$ has size $\Theta(m^2)$ the size of Φ_G is $\Theta(n^3)$ and thus polynomial in the size of the graph.

Moreover, Φ_G can be constructed in a straightforward manner from G, there is a polynomial time computable function that does the job.

Even better, with a little effort we see that the function is log-space computable: we only need to keep track of a few vertices, and those require $\log n$ bits each (recall that we don't charge for the input/output tapes).

Suppose G has a Hamiltonian cycle. We may think of this cycle as a sequence v_t, $0 \leq t \leq n$ of vertices where $v_0 = v_n = 1$.

Set $\sigma(p_{t,x}) = 1$ iff $v_t = x$.

It is easy to check that σ satisfies Φ_G.

In the opposite direction, suppose σ satisfies Φ_G. By part 1 there is a sequence of vertices v_t, $0 \leq t \leq n$: let v_t be the unique x such that $\sigma \models p_{t,x}$.

By part 2 every vertex appears on this list. Also, by part 3, $v_0 = v_n = 1$ so that all other vertices must appear exactly once by counting.

Lastly, by part 4, $v_t v_{t+1}$ is an edge.

Hence G has a Hamiltonian cycle – which can be read off directly from the satisfying truth assignment.

The same holds true for lots of other combinatorial problems that fit exactly the same pattern.

Exercise

Express Independent Set and Clique as a Satisfiability problem.

Exercise

Express Subset Sum as a Satisfiability problem:

Problem:	***Subset Sum***
Instance:	*A list of natural numbers* $a_1, \ldots, a_n, b$.
Question:	*Is there a subset* $I \subseteq [n]$ *such that* $\sum_{i \in I} a_i = b$?

Who Cares?

If we could find a fast algorithm for SAT, then we could use it to solve all these other problems.

Of course, this might not be the best strategy, it might be much better to work in the original domain (e.g., try to find a vertex cover directly). But as a general strategy this is fine.

More important for us is the opposite direction: if we cannot figure out good algorithms for any of these other problems, then there cannot be a good algorithm for SAT either. SAT is in a way the toughest nut to crack.

The brute force approach to SAT is to try all possible truth assignments, leading to a $m\,2^m$ algorithm where m is the number of variables.

Using this as a back-end to handle, say, Vertex Cover is completely useless: we only need to consider $\binom{n}{k}$ sets of vertices, not a formula with $m = \Theta(n^2)$ variables.

Big Question:
Are there any algorithms for SAT that are fast at least some/most of the time?

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.

Note the "quantification theory" in the titles: the real target was a method to establish validity in first-order logic (which can in some sense be translated into propositional logic, see Herbrand structures.

This is really the afterglow of the failure of Hilbert's program.

Thanks to Gödel we know that there cannot be a consistent and complete Hilbert system that contains a bit of arithmetic. And, according to Church and Turing, the Entscheidungsproblem is also unsolvable.

But: there still might be interesting partial answers, things still might work out in some limited circumstances. This turned out to be indeed the case.

FOL to Propositional

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

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

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

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

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

Try substitution $x = a$:

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

$$\mathit{\Gamma}_0 = \{p, \neg p \lor q, \neg q\}$$

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.

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 $\overline{x}$.
- If nothing is left, report success.

DOA.

The DPLL Idea

The basic idea of the DPLL solver is beautifully simple. Assume that the input Γ is in conjunctive normal form: Γ is a conjunction of disjunctions of literals:

$$\Gamma = \{C_1, C_2, \ldots, C_k\}$$

where each clause C_i is a disjunction of literals.

Of course, in an actual algorithm this would be a list of lists (say, of integers where i denotes x_i and $-i$ denotes $\overline{x_i}$).

- 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, Logemann 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 for a set of clauses Γ, 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.

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 $\overline{x}$ 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 $\overline{x}$, 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 $\overline{x}$.

More on PLE

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

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

So we have the partition

$$\Gamma = \Gamma_x^+ \cup \Gamma_x^- \cup \Gamma_x^*$$

Note that UCE produces $\Gamma' = \{ C - \overline{x} \mid C \in \Gamma_x^- \} \cup \Gamma_x^*$.

PLE lemma

Proposition

If Γ_x^+ or Γ_x^- is empty, then Γ and Γ_x^* are equisatisfiable.

Since Γ_x^* 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 variables, x. Backtrack to test **both**

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

  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 often succeeds beautifully in the RealWorld™, since it systematically exploits all possibilities to prune irrelevant parts of the search tree.

Silly Example

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

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

We could also have used PLE (on `d`, `a`, `c`):

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

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.

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

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

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.

$\square$

One can think of DPLL as a particular kind of a method called resolution. 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 (which we will prove soon) that shows that SAT is $\mathbb{NP}$-complete, so one should not expect algorithmic miracles. In a sense, the theorem supports experience: the algorithms exhibit exponential blowup, on occasion.

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 RealWorld 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.

Tens of thousands of variables and millions of clauses can often be handled.

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.

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

satcompetition

satlive

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