SAT solvers

Andrew W. Appel

Princeton
University

Based in part on lecture notes
by **Clark Barrett**

Outline

- Propositional logic
- The intractability of solving SAT
- The surprising tractability of SAT
- How SAT solvers work
- Modeling for SAT

Propositional Logic

A *SAT solver* solves the *Boolean satisfiability* problem.

Boolean satisfiability is phrased in the language of *propositional logic.*

Propositional symbols

$A ::= A_1 \mid A_2 \mid A_3 \mid \ldots$

Expressions (also called "well-formed formulas" wff)

$\alpha,\beta ::= A \mid \neg\alpha \mid \alpha\wedge\beta \mid \alpha\vee\beta \mid \alpha\rightarrow\beta \mid \alpha\leftrightarrow\beta \mid (\alpha)$

Semantics of propositional Logic

Syntax: *Propositional symbols A,* *Formulas* α

$A ::= A_1 \mid A_2 \mid A_3 \mid \ldots$

$\alpha,\beta ::= A \mid \neg\alpha \mid \alpha\wedge\beta \mid \alpha\vee\beta \mid \alpha\rightarrow\beta \mid \alpha\leftrightarrow\beta \mid (\alpha)$

Semantics:

Truth assignment (or "model") σ : $A\rightarrow Bool$

"Evaluate a proposition in a model"

eval(σ, α) : Bool :=
  match e with
  | $A_i \Rightarrow \sigma(A_i)$
  | $\neg\alpha \Rightarrow$ negb(eval(σ, α))
  | $\alpha\wedge\beta\Rightarrow$ andb(eval(σ, α), eval(σ, β))
  | . . . et cetera
  end

"Model satisfies a proposition" $\sigma \vDash \alpha$ when eval(σ, α)=**T**

Truth Tables

eval(σ, α) : Bool :=
 match e with
 | A_i $\Rightarrow$ σ (A_i)
 | $\neg$ α $\Rightarrow$ negb(eval(σ, α))
 | $\alpha \wedge \beta$ $\Rightarrow$ andb(eval(σ, α), eval(σ, β))
 | . . . et cetera
 end

α	$\neg\alpha$
T	F
F	T

α	β	$e_1 \vee \beta$
T	T	T
T	F	T
F	T	T
F	F	F

α	β	$\alpha \wedge \beta$
T	T	T
T	F	F
F	T	F
F	F	F

α	β	$\alpha \rightarrow \beta$
T	T	T
T	F	F
F	T	T
F	F	T

Complex Truth Tables

Truth tables can also be used to calculate all possible values of eval(σ, e)= for a given *wff*:

A	B	C	(A ∨ (B ∧ ¬C))
T	T	T	T T
T	T	F	T T
T	F	T	T F
T	F	F	T F
F	T	T	F T
F	T	F	F T
F	F	T	F F
F	F	F	F F

Complex Truth Tables

Truth tables can also be used to calculate all possible values of eval(σ, e)= for a given *wff*:

A	B	C	(A ∨ (B ∧ ¬C))
T	T	T	T T F
T	T	F	T T T
T	F	T	T F F
T	F	F	T F T
F	T	T	F T F
F	T	F	F T T
F	F	T	F F F
F	F	F	F F T

Complex Truth Tables

Truth tables can also be used to calculate all possible values of eval(σ, e)= for a given *wff*:

A	B	C	(A ∨	(B	∧	¬C))
T	T	T	T	T	F	F
T	T	F	T	T	T	T
T	F	T	T	F	F	F
T	F	F	T	F	F	T
F	T	T	F	T	F	F
F	T	F	F	T	T	T
F	F	T	F	F	F	F
F	F	F	F	F	F	T

Complex Truth Tables

Truth tables can also be used to calculate all possible values of eval(σ, e)= for a given *wff*:

A	B	C	(A	∨	(B	∧	¬C))
T	T	T	T	**T**	T	F	F
T	T	F	T	**T**	T	T	T
T	F	T	T	**T**	F	F	F
T	F	F	T	**T**	F	F	T
F	T	T	F	**F**	T	F	F
F	T	F	F	**T**	T	T	T
F	F	T	F	**F**	F	F	F
F	F	F	F	**F**	F	F	T

Gottlob Frege
1848-1925

Predicate Logic
Predicates and Relations
Universal ∀ and Existential ∃ quantifiers
Truth tables (sort of) and soundness

1879: Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens

(Concept-Script: A Formal Language for Pure Thought Modeled on that of Arithmetic)

"In effect, Frege invented axiomatic predicate logic, in large part thanks to his invention of quantified variables, which eventually became ubiquitous in mathematics and logic" -- Wikipedia

David Hilbert
1862 - 1943

one of the most influential and universal mathematicians of the 19th and early 20th centuries. … Hilbert and his students contributed significantly to establishing rigor and developed important tools used in modern mathematical physics. Hilbert is known as one of the founders of proof theory and mathematical logic, as well as for being among the first to distinguish between mathematics and metamathematics.

⊨True *versus* ⊢Valid

⊨ **truth**: a property of statements,
i.e., that they are the case.

⊢ **validity**: a property of arguments,
i.e., that they have a good structure.

Definitions ⊨

A truth assignment ***satisfies*** a wff $\sigma \models \alpha$ when eval(σ, α)=**T**.

A wff is ***satisfiable*** if there exists a truth assignment that satisfies it.

Let Σ be a set of wffs. Then **Σ *tautologically implies* α**, $\Sigma \models \alpha$, if every truth assignment that satisfies all formulas in Σ also satisfies α.

- If $\varnothing \models \alpha$ then we say α is a ***tautology*** and we write $\models \alpha$.
- If Σ is unsatisfiable, then $\Sigma \models \alpha$ for every α.
- Let $\alpha \models \beta$ be shorthand for $\{\alpha\} \models \beta$. If $\alpha \models \beta$ and $\beta \models \alpha$ then α and α are ***tautologically equivalent.***

Definitions $\vdash$

The symbol $\vdash$ is not about "truth" or "model satisfies formula", it is about ***derivability*** using a set of inference rules.

Let Σ be a set of wffs. Then $\Sigma \vdash \alpha$ if and only if there is a derivation in (for example) Gentzen's natural deduction system concluding with that sequent.

If $\varnothing \vdash \alpha$ then we say α is a ***valid*** and we write $\vdash \alpha$.

Theorem (soundness): If $\vdash \alpha$ then $\models \alpha$.

Theorem (completeness): If $\models \alpha$ then $\vdash \alpha$.

Propositional logic (e.g., Gentzen's natural deduction) is sound and complete.

So α is ***valid*** if and only if α is a ***tautology.***

David Hilbert
1862 - 1943

one of the most influential and universal mathematicians of the 19th and early 20th centuries. … Hilbert and his students contributed significantly to establishing rigor and developed important tools used in modern mathematical physics. Hilbert is known as one of the founders of proof theory and mathematical logic, as well as for being among the first to distinguish between mathematics and metamathematics.

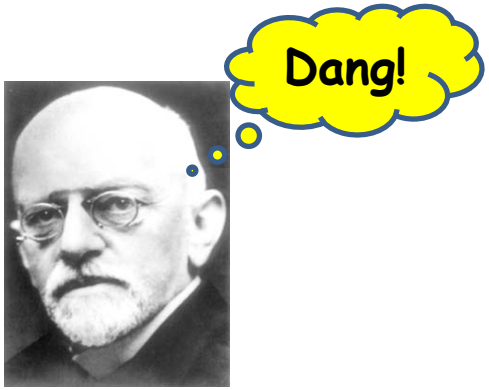
In 1920 he proposed explicitly a research project (in metamathematics, as it was then termed) that became known as Hilbert's program. He wanted mathematics to be formulated on a solid and complete logical foundation. He believed that in principle this could be done, by showing that:
(1) all of mathematics follows from a correctly chosen finite system of axioms; and
(2) that some such axiom system is provably consistent

Proved (1931) that

(1) no finite axiomatization can prove all true statements

(2) cannot prove consistency of a system in the system itself

Kurt Gödel
1906 – 1978
(lived 1934-1978 in Princeton, NJ)



Kurt Gödel
1906 – 1978

More specifically, he proved:

Any deductive system sufficiently powerful to have,

- propositional connectives (∧,∨,¬)
- quantifiers over the natural numbers (∀,∃)
- addition and multiplication (+,×)

cannot be both ***sound*** and ***complete.***

We generally choose ***soundness:*** if you can prove it, then it's true.

But fortunately, propositional calculus is so weak that it can be (and is) both sound and complete.

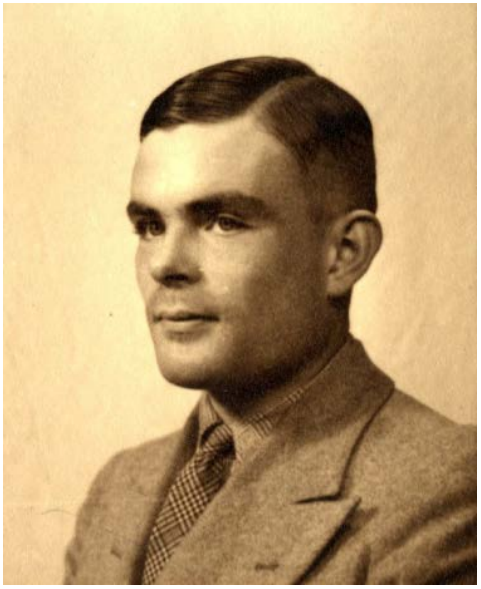
David Hilbert
1862 - 1943

Well, if we can't have an algorithm for deciding what's ***true,***

maybe we can at least,

- formalize what is an "algorithm", and

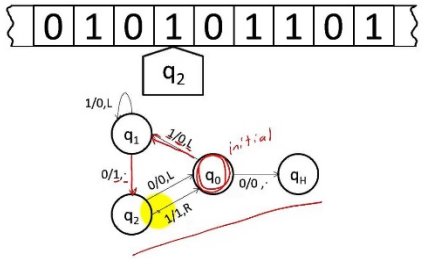
- Have an algorithm for deciding what's ***provable.***



Alan Turing
1912 - 1954

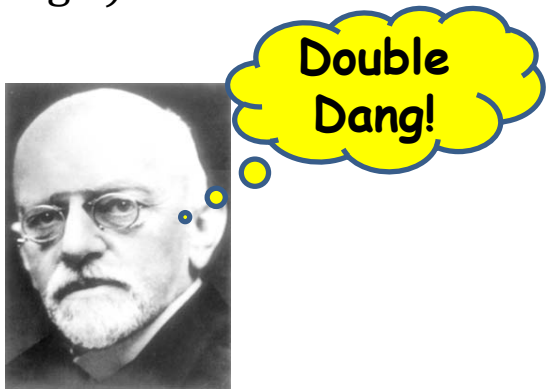
Good news! (1936)

I can formalize an “algorithm”



Bad news!

No algorithm can **decide** which statements are provable (in first-order logic)



Examples

- $(A \vee B) \wedge (\neg A \vee \neg B)$ is ***satisfiable*** but not ***valid***
- $(A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \leftrightarrow B)$ is ***unsatisfiable***
- $A, A \rightarrow B \models B$ $\quad (A \wedge (A \rightarrow B)) \wedge \neg B$
- $A, \neg A \models A \wedge \neg A$ $\quad (A \wedge \neg A) \wedge \neg (A \wedge \neg A)$
- $\neg(A \wedge B)$ is ***tautologically equivalent*** to $\neg A \vee \neg B$

$\neg(\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B))$

Suppose you had an algorithm SAT that would take a wff α as input, and return True if α is satisfiable and False otherwise.

How would you use this algorithm to verify each of the claims made above?

Some tautologies

Associative and commutative laws for $\wedge, \vee, \leftrightarrow$

e.g., $A \wedge (B \wedge C) \leftrightarrow (A \wedge B) \wedge C$ $\qquad$ $A \wedge B \leftrightarrow B \wedge A$

Distributive laws

$A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$

$A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$

De Morgan's laws

$\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$

$\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$

Implication $(A \rightarrow B) \leftrightarrow (\neg A \vee B)$

The intractability of SAT

What we knew in the 20th century

Determining satisfiability using truth tables

Example

$A \wedge ((B \vee \neg A) \wedge (C \vee \neg B))$

A	B	C	A	∧	((B	∨	¬A)	∧	(C	∨	¬B))
F	F	F	F		T	T	T		T	T	
F	F	T	F		T	T	T		T	T	
F	T	F	F		T	T	F		F	F	
F	T	T	F		T	T	T		T	F	
F	F	F	F		F	F	F		T	T	
F	F	T	F		F	F	F		T	T	
F	T	F	F		T	F	F		F	F	
F	T	T	T		T	F	T		T	F	

Determining satisfiability using truth tables

What is the complexity of this algorithm?

2^n where n is the number of propositional symbols

Can we do better?

SAT was the first problem to be shown NP-complete; all of the problems in the class NP can be solved by reducing them (in polynomial time) to SAT.

So, if we could build a *fast* solver for SAT, it could be used to solve lots of other problems.

Stephen Cook
1939 –

U. of Toronto 1970—
Turing Award 1982

Proved (1971) that

It is NP-complete to decide whether a boolean formula is satisfiable

(also, invented NP-completeness)

Trivia factoids:

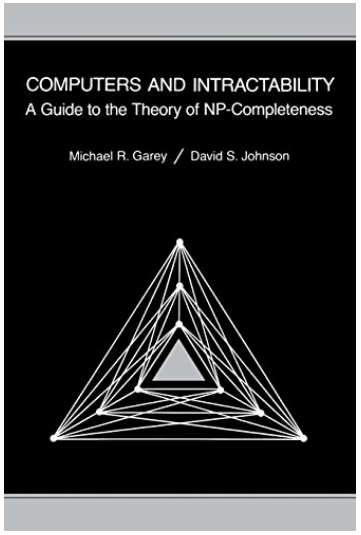
- In 1970, the Berkeley Math department denied him tenure.

- Stephen Cook was Professor Mark Braverman's PhD adviser (2008)

$P \stackrel{?}{=} NP$

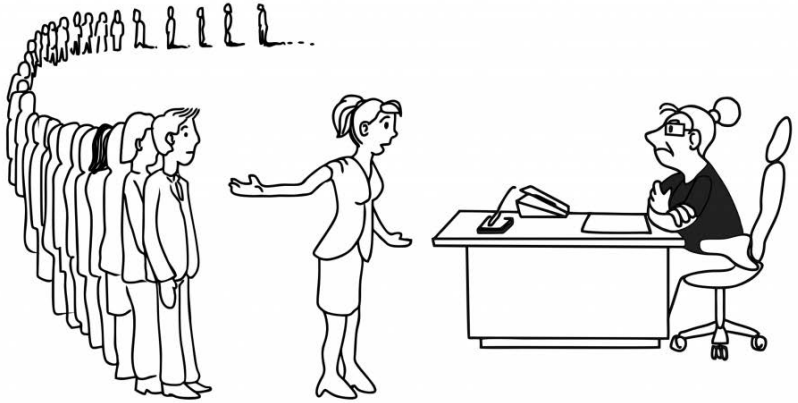
- No algorithm is known for SAT (or any other NP-complete problem) that is polynomial-time in the worst case.

Garey and Johnson 1979



"I can't find an efficient algorithm, I guess I'm just too dumb."

"I can't find an efficient algorithm, because no such algorithm is possible!"



"I can't find an efficient algorithm, but neither can all these famous people."

SAT in the 20th century

By 1979, it was well understood
by computer scientists that:

- You prove problem *X* is NP-complete by *reducing SAT to X*

- If you *reduce X to SAT* then obviously you don't know what you're doing

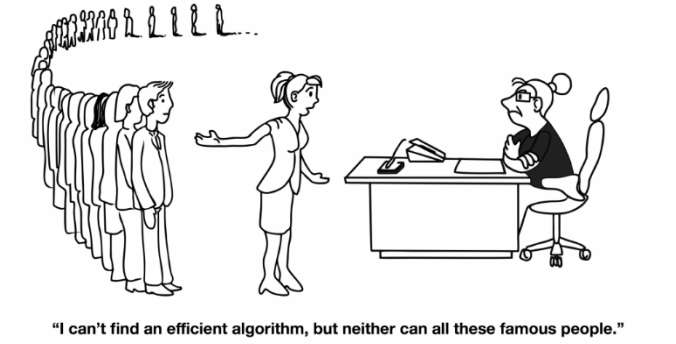
A conversation in 1981

Andrew: Hey, Philip, what are you working on?
Philip: My University of Illinois master's thesis in Computer Science

Andrew: What's it about?
Philip: An analyzer to test whether a concurrent program can deadlock

Andrew: Cool, how's it work?
Philip: I'll just reduce the Petri Net to "testing the equivalence of Boolean formulas"

Andrew: That'll never work! Don't you know about the theory of NP completeness?



"I can't find an efficient algorithm, but neither can all these famous people."

Philip was 30 years ahead of his time . . .

Philip: No problem, I already got into Yale Medical School . . .

SAT in the 21st century

. . . is very different!

Proceedings of the 38th annual Design Automation Conference, 2001

33.1

Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz
Department of EECS
UC Berkeley
moskewcz@alumni.princeton.edu

Conor F. Madigan
Department of EECS
MIT
cmadigan@mit.edu

Ying Zhao, Lintao Zhang, Sharad Malik
Department of Electrical Engineering
Princeton University
{yingzhao, lintaoz, sharad}@ee.princeton.edu

Princeton Undergrads

PhD students

Professor

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the development of several SAT packages, both proprietary and in the public domain (e.g. GRASP, SATO) which find significant use in both research and industry. Most existing complete solvers are variants of the Davis-Putnam (DP) search algorithm. In this paper we describe the development of a new complete solver, Chaff, which achieves significant performance gains through careful engineering of all aspects of the search – especially a particularly efficient implementation of Boolean constraint propagation (BCP) and a novel low overhead decision strategy. Chaff has been able to obtain one to two orders of magnitude performance improvement on difficult SAT benchmarks in comparison with other solvers (DP or otherwise), including GRASP and SATO.

Categories and Subject Descriptors

J6 [**Computer-Aided Engineering**]: Computer-Aided Design.

General Terms

Algorithms, Verification.

Keywords

developed, most employing some combination of two main strategies: the Davis-Putnam (DP) local search. Heuristic local searc to be complete (i.e. they are not assignment if one exists or prov complete SAT solvers (includ exclusively on the DP search algor

1.1 Problem Specificati

Most solvers operate on prob conjunctive normal form (*CNF*). AND of one or more *clauses*, wh one or more *literals*. The *liter* logical unit in the problem, being or its complement. (In this paper ¬.) All Boolean functions can b The advantage of *CNF* is that in (*sat*), each individual *clause* must

1.2 Basic Davis-Putnam

We start with a quick revie backtrack search. This is described in the following pseudo-code fragment:

```
while (true) {
  if (!decide()) // if no unassigned vars
    return(satisifiable);
  while (!bcp()) {
    if (!resolveConflict())
```

CNF

Most algorithms for SAT first convert the formula into ***conjunctive normal form (CNF).***

- A ***literal*** is a propositional variable or its negation
- A ***clause*** is a disjunction of literals
- A ***formula*** is in CNF if it is a conjunction of clauses
- A propositional symbol occurs ***positively*** if it occurs unnegated in a clause
- A propositional symbol occurs ***negatively*** if it occurs negated in a clause

CNF

Examples

- Literals: P_i, $\neg P_i$
- Clauses: $(P_1 \vee \neg P_3 \vee P_5)$, $(P_2 \vee \neg P_2)$

- CNF: $(P_1 \vee \neg P_3) \wedge (\neg P_2 \vee P_3 \vee P_5)$
- P_1 occurs positively and P_2 occurs negatively

The wrong way to convert to CNF

Step 1: push negations to the leaves. Repeat:

$$\neg(\alpha \wedge \beta) \rightarrow (\neg\alpha \vee \neg\beta)$$
$$\neg(\alpha \vee \beta) \rightarrow (\neg\alpha \wedge \neg\beta)$$
$$\neg\neg\alpha \rightarrow \alpha$$

Step 2: bottom up, repeat:

Let $\alpha=(A_1 \wedge \ldots \wedge A_m)$, $\beta=(B_1 \wedge \ldots \wedge B_n)$ be CNFs, with A_i, B_i *literals*

$$\alpha \wedge \beta \rightarrow (A_1 \wedge \ldots \wedge A_m \wedge B_1 \wedge \ldots \wedge B_n)$$

$$\alpha \vee \beta \rightarrow (A_1 \vee B_1) \wedge (A_1 \vee B_2) \ldots \wedge (A_1 \vee B_n) \wedge$$
$$(A_2 \vee B_1) \wedge (A_2 \vee B_2) \ldots \wedge (A_2 \vee B_n) \wedge$$
$$\ldots$$
$$(A_m \vee B_1) \wedge (A_m \vee B_2) \ldots \wedge (A_m \vee B_n)$$

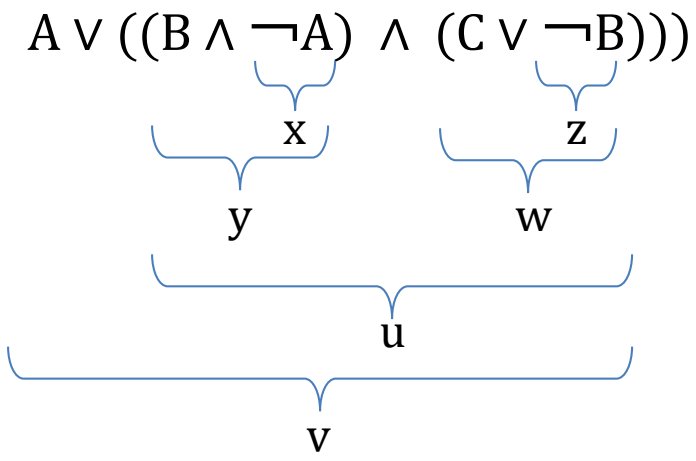
(blows up the formula exponentially!)

The right way to convert to CNF

Step 1: convert formula to ***administrative-normal form (ANF)***

e ::= let x=y∧z in e | let x=y∨z in e | let x=¬y in e | return x

(there is one "let", and one new variable, for every operator in the original expression)



let x = ¬A in
let y = B ∧ x in
let z = ¬B in
let w = C ∨ z in
let u = y ∧ w in
let v = A ∨ u in
return v

The right way to convert to CNF

Step 1: convert formula to *administrative-normal form (ANF)*

e ::= let x=y∧z in e | let x=y∨z in e | let x=¬y in e | return x

Step 2: turn each line of ANF into CNF

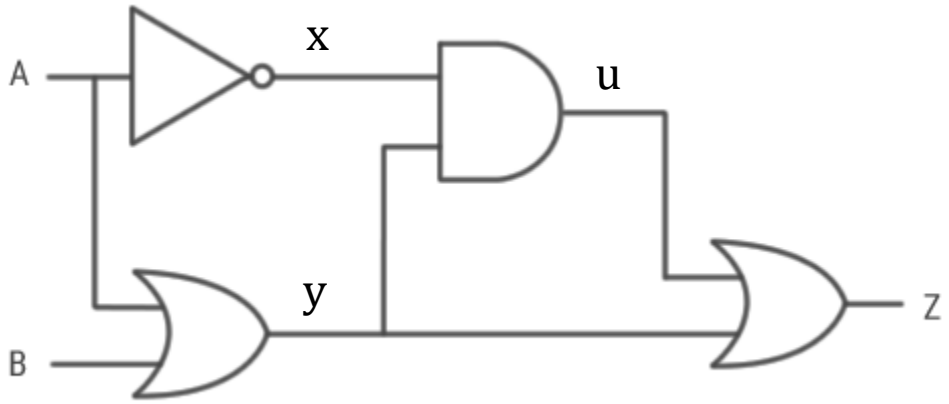
let x=y∧z in	⇒	x↔(y∧z)	⇒ (¬x ∨ y) ∧ (¬x ∨ z) ∧ (¬y ∨¬z ∨ x)
let x=y∨z in	⇒	x↔(y∨z)	⇒ (¬x ∨ y ∨ z) ∧ (¬y ∨x) ∧ (¬z ∨ x)
let x=¬y in	⇒	x ↔¬y	⇒ (¬x ∨ ¬y) ∧ (x ∨ y)
return x	⇒	x	

then **and** all those conjuncts together.

Resulting term has size ***linear*** in the size of the original, but has extra variables.
The new formula is **not** tautologically equivalent to the original! (Why?)
But it is ***equisatifiable:*** the new formula is satisfiability iff the original is.

A-Normal Form, digital circuits

let x = ¬A in
let y = A ∨ B in
let u = x ∧ y in
let Z = u ∨ y in
return Z



An ANF expression is equivalent to a combinational Boolean circuit.
Either notation can comfortably express ***common subexpressions.***

Each line of ANF is one logic gate of the circuit.
Each line of ANF, each logic gate, can be expressed as a constant-size set of CNF conjuncts.

CNF can model digital circuits as conveniently as it can model Boolean formulas.

How to use a SAT-solver

1. Express your verification problem as a Boolean formula
   - Usually you are interested in proving this formula is ***valid.***
2. Convert the negation of your formula to Conjunctive Normal Form
3. Run a SAT solver to prove UNSAT
   - (or if not UNSAT, to find a counterexample, that is, a satisfying assignment)

Proceedings of the 38th annual Design Automation Conference, 2001

33.1

Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz
Department of EECS
UC Berkeley
moskewcz@alumni.princeton.edu

Conor F. Madigan
Department of EECS
MIT
cmadigan@mit.edu

Ying Zhao, Lintao Zhang, Sharad Malik
Department of Electrical Engineering
Princeton University
{yingzhao, lintaoz, sharad}@ee.princeton.edu

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the development of several SAT packages, both proprietary and in the public domain (e.g. GRASP, SATO) which find significant use in both research and industry. Most existing complete solvers are variants of the Davis-Putnam (DP) search algorithm. In this paper we describe the development of a new complete solver, Chaff, which achieves significant performance gains through careful engineering of all aspects of the search – especially a particularly efficient implementation of Boolean constraint propagation (BCP) and a novel low overhead decision strategy. Chaff has been able to obtain one to two orders of magnitude performance improvement on difficult SAT benchmarks in comparison with other solvers (DP or otherwise), including GRASP and SATO.

Categories and Subject Descriptors

J6 [**Computer-Aided Engineering**]: Computer-Aided Design.

General Terms

Algorithms, Verification.

Keywords

developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a result, complete SAT solvers (including ours) are based almost exclusively on the DP search algorithm.

1.1 Problem Specification

Most solvers operate on problems for which f is specified in conjunctive normal form (*CNF*). This form consists of the logical AND of one or more *clauses*, which consist of the logical OR of one or more *literals*. The *literal* comprises the fundamental logical unit in the problem, being merely an instance of a variable or its complement. (In this paper, complement is represented by ¬.) All Boolean functions can be described in the *CNF* format. The advantage of *CNF* is that in this form, for f to be satisfied (*sat*), each individual *clause* must be *sat*.

1.2 Basic Davis-Putnam Backtrack Search

We start with a quick review of the basic Davis-Putnam backtrack search. This is described in the following pseudo-code fragment:

```
while (true) {
  if (!decide()) // if no unassigned vars
    return(satisifiable);
  while (!bcp()) {
    if (!resolveConflict())
```

Journal of the ACM, July 1960

A Computing Procedure for Quantification Theory*

MARTIN DAVIS

Rensselaer Polytechnic Institute, Hartford Division, East Windsor Hill, Conn.

AND

HILARY PUTNAM

Princeton University, Princeton, New Jersey

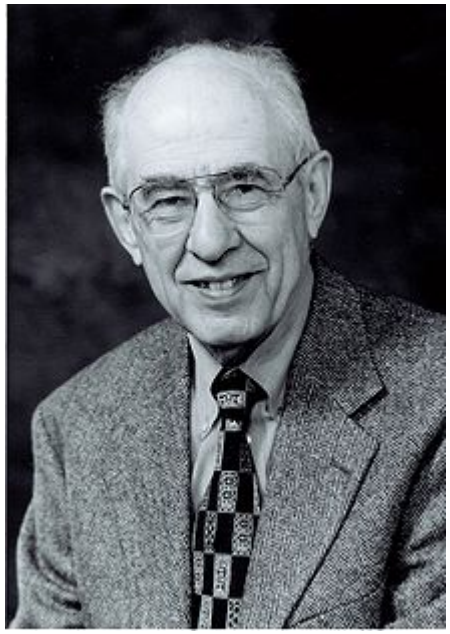
The hope that mathematical methods employed in the investigation of formal logic would lead to purely computational methods for obtaining mathematical theorems goes back to Leibniz and has been revived by Peano around the turn of the century and by Hilbert's school in the 1920's. Hilbert, noting that all of classical mathematics could be formalized within quantification theory, declared that the problem of finding an algorithm for determining whether or not a given formula of quantification theory is valid was the central problem of mathematical logic. And indeed, at one time it seemed as if investigations of this "decision" problem were on the verge of success. However, it was shown by Church and by Turing that such an algorithm can not exist. This result led to considerable pessimism regarding the possibility of using modern digital computers in deciding significant mathematical questions. However, recently there has been a revival of interest in the whole question. Specifically, it has been realized that while no *decision procedure* exists for quantification theory there are many proof procedures available—that is, uniform procedures which will ultimately locate a proof for any formula of quantification theory which is valid but which will usually involve seeking "forever" in the case of a formula which is not valid—and that some of these proof procedures could well turn out to be feasible for use with modern computing machinery.

Martin Davis (1928 -)

winner of the 2009 Pioneering Achievement Award from ACM SIG Design Automation, " For his fundamental contributions to algorithms for solving the Boolean Satisfiability problem, which heavily influenced modern tools for hardware and software verification, as well as logic circuit synthesis."

PhD Princeton 1950 (adviser: Alonzo Church)
Professor, New York University

Coinventor of Davis-Putnam algorithm (1960) for resolution theorem proving



Hilary Putnam, 1926-2016

PhD (Philosophy) 1951, UCLA
Professor:
Northwestern, Princeton, MIT (1951-1965)
Harvard (1965-2000)

Main interests

Philosophy of mind
Philosophy of language
Philosophy of science
Philosophy of mathematics
Metaphilosophy
Epistemology

Notable ideas

Multiple realizability
Functionalism
Causal theory of reference
Semantic externalism
Brain in a vat · Twin Earth
Internal realism

Davis-Putnam Algorithm

From now on, unless otherwise indicated, we assume formulas are in CNF, *or, equivalently, that we have a set of clauses to check for satisfiability (i.e. the conjunction is implicit).*

The first algorithm to try something more sophisticated than the truth-table method was the *Davis-Putnam (DP)* algorithm, published in 1960.

It is often confused with the later, more popular algorithm presented by Davis, Logemann, and Loveland in 1962, which we will refer to as *Davis-Putnam-Logemann-Loveland (DPLL)*.

We first consider the original DP algorithm.

Davis-Putnam Algorithm

There are three satisfiability-preserving transformations in DP.

- The 1-literal rule
- The affirmative-negative rule
- The rule for eliminating atomic formulas

The first two steps reduce the number of literals in the formula.

The last step reduces the number of variables in the formula.

By repeatedly applying these rules, eventually we obtain a formula containing an empty clause, indicating unsatisfiability, or a formula with no clauses, indicating satisfiability.

Davis-Putnam Algorithm

The 1-literal rule also called ***unit propagation.***

Suppose (p) is a unit clause (clause containing only one literal). Let −p denote the negation of p where double negation is collapsed (i.e. −¬q ≡ q).

- Remove all instances of −p from clauses in the formula (shortening the corresponding clauses).

- Remove all clauses containing p (including the unit clause itself).

Davis-Putnam Algorithm

The affirmative-negative rule

also called the ***pure literal rule.***

- If a literal appears *only positively*
  or *only negatively,*
  delete all clauses containing that literal.

Why does this preserve satisfiability?

Davis-Putnam Algorithm

The resolution rule

- Choose a propositional symbol p which occurs positively in at least one clause and negatively in at least one other clause.
- Let P be the set of all clauses in which p occurs positively.
- Let N be the set of all clauses in which p occurs negatively.
- Replace the clauses in P and N with those obtained by resolution on p using all pairs of clauses from P and N.
- For each pair of clauses,

  $(p \vee l_1 \vee \cdots \vee l_m)$ and $(\neg p \vee k_1 \vee \cdots \vee k_n)$,

  resolution on p forms the new clause

  $(l_1 \vee \cdots \vee l_m \vee k_1 \vee \cdots \vee k_n)$.

Warning! The number of new clauses is $|P| \cdot |N|$,
so *each* resolution can cause quadratic blowup,
and a series of resolutions can cause exponential blowup.

DPLL Algorithm

Martin Davis, George Logemann, Donald Loveland 1962

In the worst case, the resolution rule can cause a quadratic expansion every time it is applied.

For large formulas, this can quickly exhaust the available memory.

The DPLL algorithm replaces resolution with a *splitting rule*.

- Choose a propositional symbol *p* occuring in the formula.
- Let Δ be the current set of clauses.
- Test the satisfiability of $\Delta \cup \{(p)\}$.
- If satisfiable, return *True*.
- Otherwise, return the result of testing $\Delta \cup \{(\neg p)\}$ for satisfiability.

Proceedings of the 38th annual Design Automation Conference, 2001

33.1

Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz
Department of EECS
UC Berkeley
moskewcz@alumni.princeton.edu

Conor F. Madigan
Department of EECS
MIT
cmadigan@mit.edu

Ying Zhao, Lintao Zhang, Sharad Malik
Department of Electrical Engineering
Princeton University
{yingzhao, lintaoz, sharad}@ee.princeton.edu

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in A
deve
publ
both
varia
we
whic
engi
effic
and
to
impr
othe

In this paper, we describe the development of a new complete solver, Chaff, which achieves significant performance gains through careful engineering of all aspects of the search—especially a particularly efficient implementation of Boolean constraint propagation and a novel low-overhead decision strategy.

Categories and Subject Descriptors

J6 [**Computer-Aided Engineering**]: Computer-Aided Design.

General Terms

Algorithms, Verification.

Keywords

developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a result, complete SAT solvers (including ours) are based almost

cified in
e logical
al OR of
damental
variable
ented by
7 format.
satisfied

ch

We start with a quick review of the basic Davis-Putnam backtrack search. This is described in the following pseudo-code fragment:

```
while (true) {
  if (!decide()) // if no unassigned vars
    return(satisifiable);
  while (!bcp()) {
    if (!resolveConflict())
```

Some Experimental Results

Problem	truth table	DP	DPLL
prime 3	0.00	0.00	0.00
prime 4	0.02	0.06	0.04
prime 9	18.94	2.98	0.51
prime 10	11.40	3.03	0.96
prime 11	28.11	2.98	0.51
prime 16	>3600.00	out of memory	9.15
ramsey 3 3 5	0.03	0.06	0.02
ramsey 3 3 6	5.13	8.28	0.31
adder 3 2	>>3600.00	6.50	7.34
adder 4 2	>>3600.00	22.95	46.86
adder 5 2	>>3600.00	44.83	170.98
adder 6 3	>>3600.00	out of memory	1186.4
adder 7 3	>>3600.00	out of memory	3759.9

John Harrison, *Introduction to Logic and Automated Theorem Proving*, 2009

Abstract DPLL

Abstract DPLL uses *states* and *transitions* to model the progress of the algorithm.

- Most states are of the form M ‖ F, where
  - M is a *stack* of annotated *literals* denoting a partial truth assignment, and
  - F is the CNF formula being checked, represented as a *set of clauses*.

- The *initial state* is ∅ ‖ F, where F is to be checked for satisfiability.

- Transitions between states are defined by a set of *conditional transition rules*.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006

Abstract DPLL

- Most states are of the form M ‖ F, where
  - M is a *stack* of annotated *literals* denoting a partial truth assignment, and
  - F is the CNF formula being checked, represented as a *set of clauses*.
- The *initial state* is ∅ ‖ F, where F is to be checked for satisfiability.
- Transitions between states are defined by a set of *conditional transition rules*.

The *final state* is either:

- a special fail state: fail , if F is unsatisfiable, or
- M ‖ G, where G is a CNF formula equisatisfiable with the original formula F, and M satisfies G

We write M ⊨ C to mean that for every truth assignment v,

$$v(M) = \textit{True} \text{ implies } v(C) = \textit{True}.$$

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006

Abstract DPLL rules

UnitProp

$M \parallel F, C \vee l \implies M\ l \parallel F, C \vee l$ if $\begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$

PureLiteral

$M \parallel F \implies M\ l \parallel F$ if $\begin{cases} l \text{ occurs in some clause of } F \\ -l \text{ occurs in no clause of } F \\ l \text{ is undefined in } M \end{cases}$

Decide

$M \parallel F \implies M\ l^{d} \parallel F$ if $\begin{cases} l \text{ or } -l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$

Backtrack

$M\ l^{d}\ N \parallel F, C \implies M \neg l \parallel F, C$ if $\begin{cases} M\ l^{d}\ N \models \neg C \\ N \text{ contains no decision literals} \end{cases}$

Fail

$M \parallel F, C \implies \mathit{fail}$ if $\begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$

Example

$\emptyset$	$\| 1 \vee \bar{2}, \bar{1} \vee \bar{2}, 2 \vee 3, \bar{3} \vee 2, 1 \vee 4$	$\Longrightarrow$	(PureLiteral)
4	$\| 1 \vee \bar{2}, \bar{1} \vee \bar{2}, 2 \vee 3, \bar{3} \vee 2, 1 \vee 4$	$\Longrightarrow$	(Decide)
$4\ 1^d$	$\| 1 \vee \bar{2}, \bar{1} \vee \bar{2}, 2 \vee 3, \bar{3} \vee 2, 1 \vee 4$	$\Longrightarrow$	(UnitProp)
$4\ 1^d\ \bar{2}$	$\| 1 \vee \bar{2}, \bar{1} \vee \bar{2}, 2 \vee 3, \bar{3} \vee 2, 1 \vee 4$	$\Longrightarrow$	(UnitProp)
$4\ 1^d\ \bar{2}\ 3$	$\| 1 \vee \bar{2}, \bar{1} \vee \bar{2}, 2 \vee 3, \bar{3} \vee 2, 1 \vee 4$	$\Longrightarrow$	(Backtrack)
$4\ \bar{1}$	$\| 1 \vee \bar{2}, \bar{1} \vee \bar{2}, 2 \vee 3, \bar{3} \vee 2, 1 \vee 4$	$\Longrightarrow$	(UnitProp)
$4\ \bar{1}\ \bar{2}\ \bar{3}$	$\| 1 \vee \bar{2}, \bar{1} \vee \bar{2}, 2 \vee 3, \bar{3} \vee 2, 1 \vee 4$	$\Longrightarrow$	(Fail)
	fail		

Boolean constraint propagation

UnitProp

$$M \parallel F, C \vee l \implies M\ l \parallel F, C \vee l \quad \text{if} \begin{cases} M \vDash \neg C \\ l \text{ is undefined in } M \end{cases}$$

The most expensive part of a SAT solver is the part that checks for and applies instances of the UnitProp rule.

A key insight that can be used to speed this up is that as long as a clause has at least two unassigned literals, it cannot participate in an application of UnitProp.

For every clause, we assign two of its unassigned literals as the *watched* literals.

Every time a literal is assigned, only those clauses in which it is watched need to be checked for a possible triggering of the UnitProp rule.

For those clauses that are inspected, if UnitProp is not triggered, a new unassigned literal is chosen to be watched.

Other considerations

Modern SAT solvers have many other tricks to speed things up:

- "Backjump" & "Learn" rules
- Highly tuned code
- Optimization for cache performance
- Preprocessing and clever CNF encodings
- Automatic tuning of program parameters

There are competitions every year, connected to the *International Conference on Theory and Applications of Satisfiability Testing*

APPLICATIONS OF SAT-SOLVERS

Program verification

- See next lecture

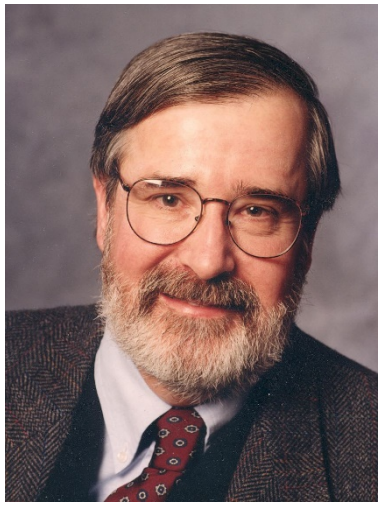
Hardware (circuit) verification

- See: across Olden Street

Symbolic model checking

Research since the 1980s pioneered by Ed Clarke. In the 20th century, used Binary Decision Diagrams (BDDs), but in the 21st century . . .

E. Clarke, A. Biere, R. Raimi, and Y. Zhu.
Bounded Model Checking Using Satisfiability Solving.
Formal Methods in System Design, 19(1):7–34, 2001



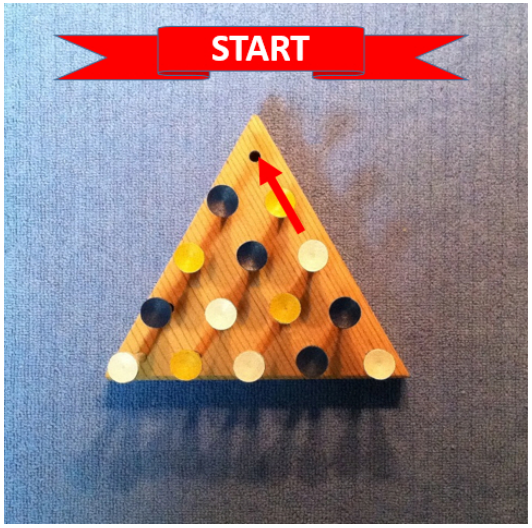
Edmund Clarke, 1945-2020
PhD 1976, Cornell
Professor, CMU, 1982-2015

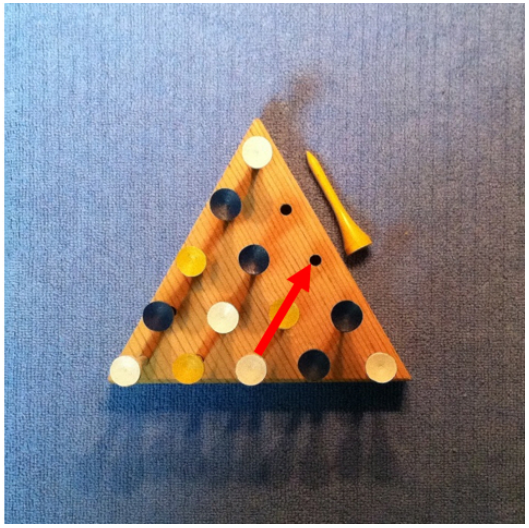
Adviser:
Robert Constable, whose adviser was
Stephen Kleene, whose adviser was
Alonzo Church

Turing Award, 2007

"for . . . developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries."

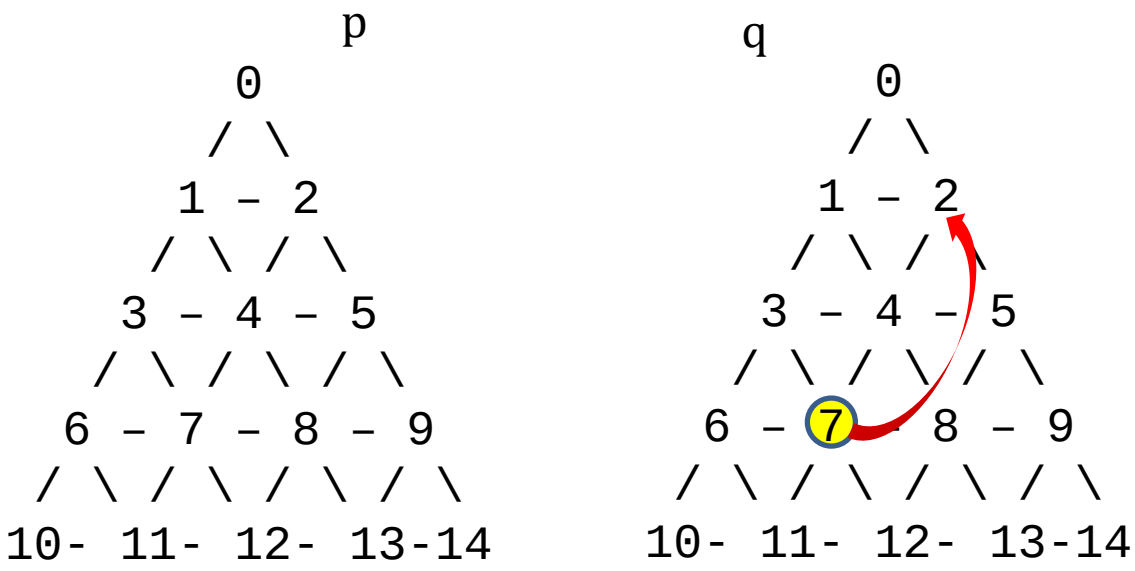
Using SAT to solve peg solitaire





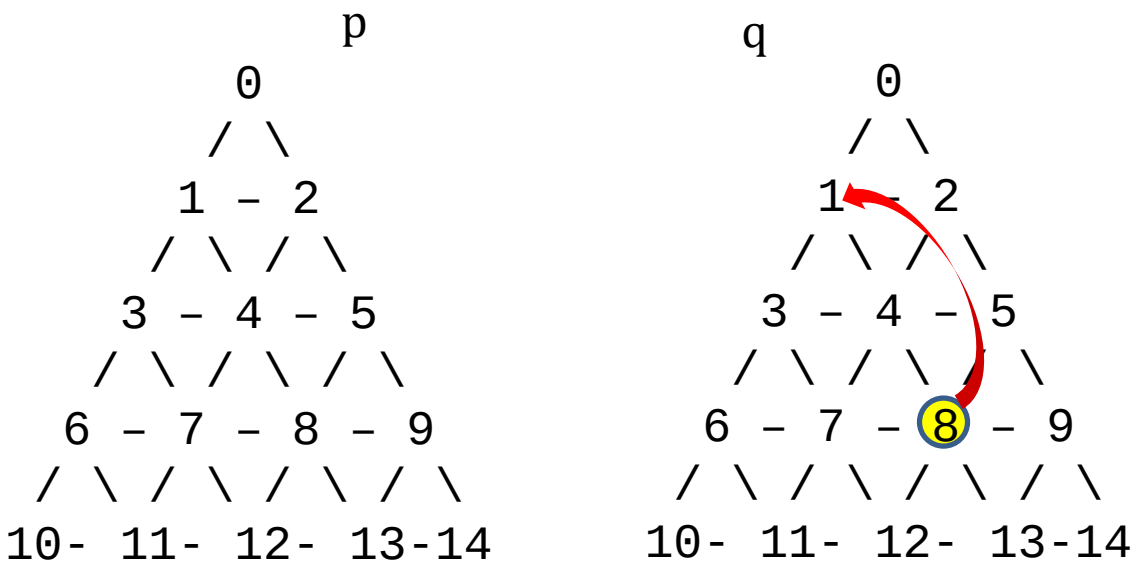
https://www.youtube.com/watch?v=ILKXEnX_YGM

Modeling one move



(p7)(p4)(¬p2)(¬q7)(¬q4)(q2)
(p0↔q0) (p1↔q1) (p3↔q3)(p5↔q5) (p6↔q6) (p8↔q8)
(p9↔q9) (p10↔q10) (p11↔q11)(p12↔q12) (p13↔q13) (p14↔q14)

Modeling one move



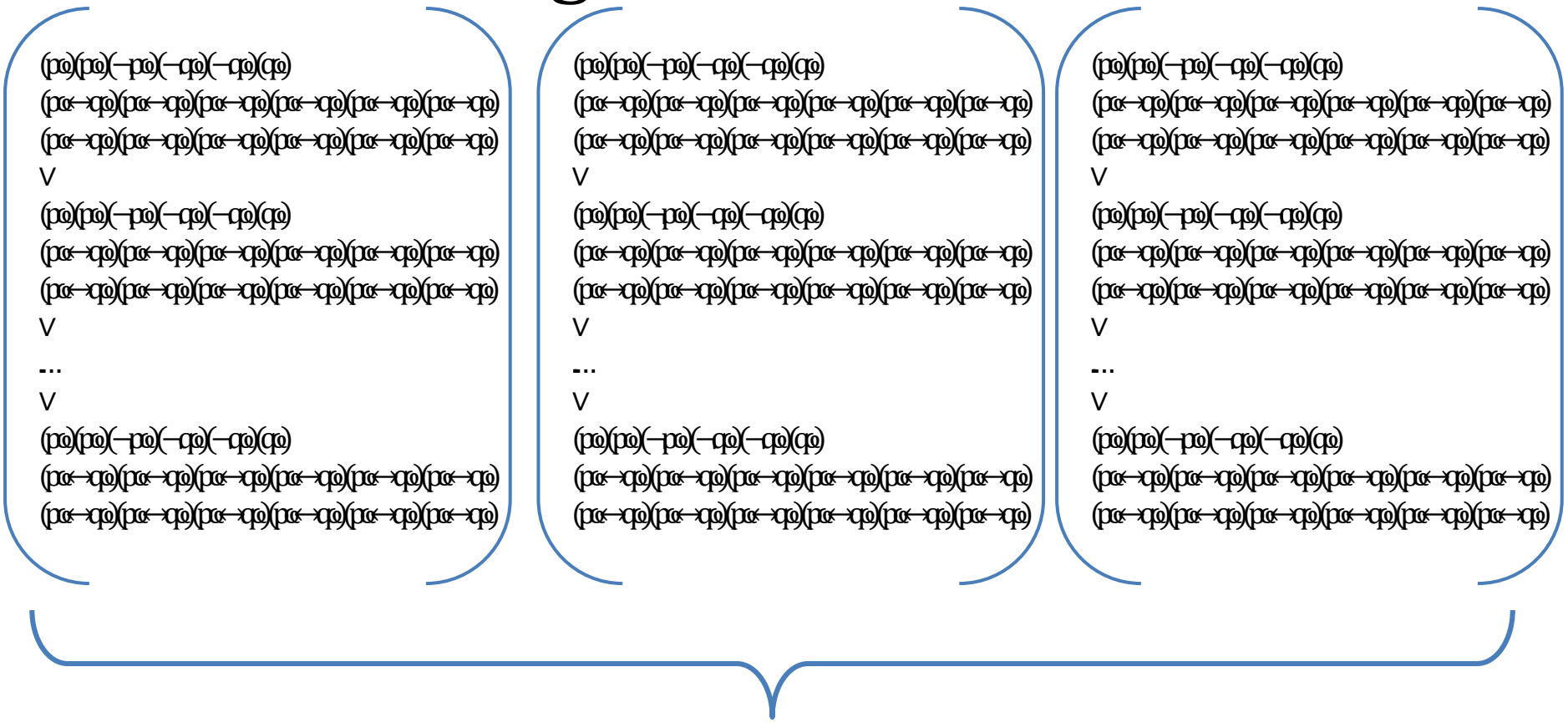
(p7)(p4)(¬p2)(¬q7)(¬q4)(q2)
(p0↔q0) (p1↔q1) (p3↔q3)(p5↔q5) (p6↔q6) (p8↔q8)
(p9↔q9) (p10↔q10) (p11↔q11)(p12↔q12) (p13↔q13) (p14↔q14)
∨
(p8)(p4)(¬p1)(¬q8)(¬q4)(q1)
(p0↔q0) (p2↔q2) (p3↔q3)(p5↔q5) (p6↔q6) (p7↔q7)
(p9↔q9) (p10↔q10) (p11↔q11)(p12↔q12) (p13↔q13) (p14↔q14)

Modeling one move

$(p\omega)(p\omega)(\neg p\omega)(\neg q\omega)(\neg q\omega)(q\omega)$
$(p\ \omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)$
$(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)$
$\vee$
$(p\omega)(p\omega)(\neg p\omega)(\neg q\omega)(\neg q\omega)(q\omega)$
$(p\ \omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)$
$(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)$
$\vee$
$\ldots$
$\vee$
$(p\omega)(p\omega)(\neg p\omega)(\neg q\omega)(\neg q\omega)(q\omega)$
$(p\ \omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)$
$(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)(p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)\ (p\omega \leftrightarrow q\omega)$

36 of these

Modeling consecutive moves



This is 210 variables and thousands of clauses. No problem!
Solves in about 1.5 seconds. Other encodings solve in 0.06 seconds.

Why? Abstract DPLL is very efficient at unit propagation, et cetera.