

Propositional Logic and SAT Solvers

Prof. Vijay Ganesh



WHAT IS LOGIC?

References

Introduction to Logic by Enderton

Decision Procedures by Strichman and Kroening

What is Formal Logic

Formal Logic consists of

- Precise language
 - syntax – what is a legal sentence in the logic
 - semantics – what is the meaning of a sentence in the logic
- Axioms, foundations, and proofs – formal (syntactic) procedure to construct valid/true sentences
- Computation: Godel, Turing, Hilbert,...
 - George Cantor – Father of Set Theory – Can natural number be put in one-to-one correspondence with reals?
 - Cantor proved that natural numbers are countably infinite while the reals are uncountably infinite, and uncountable is much larger than the countable

Formal logic provides

- a language to precisely express knowledge, requirements, facts
- a formal way to reason about consequences of given facts rigorously

Where is Formal Logic used in SE?

Programming languages

- meaning (semantics) of programs

Requirements and specification of software systems

- rigorous definition of what is to be constructed (logic specifications)

Specification of computer hardware

- computers are built out of simple logical gates
- most hardware can be specified and understood in propositional logic

Testing and Verification

- rigorously validate that software satisfies its specifications

Algorithms and Optimization

- many complex problems can be reduced to logic and solved effectively using automated decision procedures (e.g., SAT solvers)

AI = ML and Logic

- Robot planning, combinations of ML and logic

Propositional Logic (or Boolean Logic)

Explores simple grammatical connections such as *and*, *or*, and *not* between simplest “atomic sentences”

A = “Paris is the capital of France”

B = “mice chase elephants”

The subject of propositional logic is to declare formally the truth of complex structures from the truth of individual atomic components

A and B

A or B

if A then B

MOTIVATION TO STUDY SAT SOLVERS

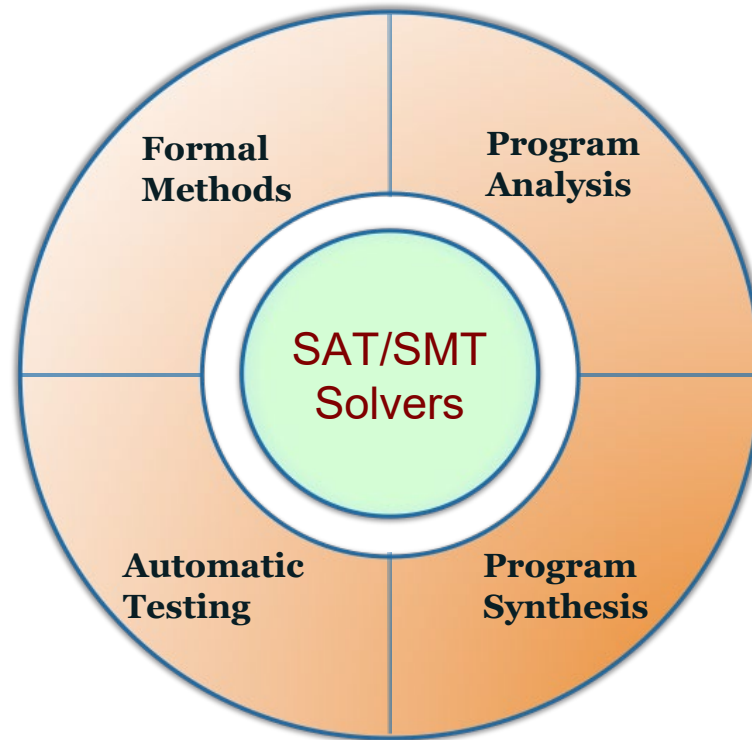
PART I

MOTIVATION

WHY SHOULD YOU CARE ABOUT SAT SOLVERS?

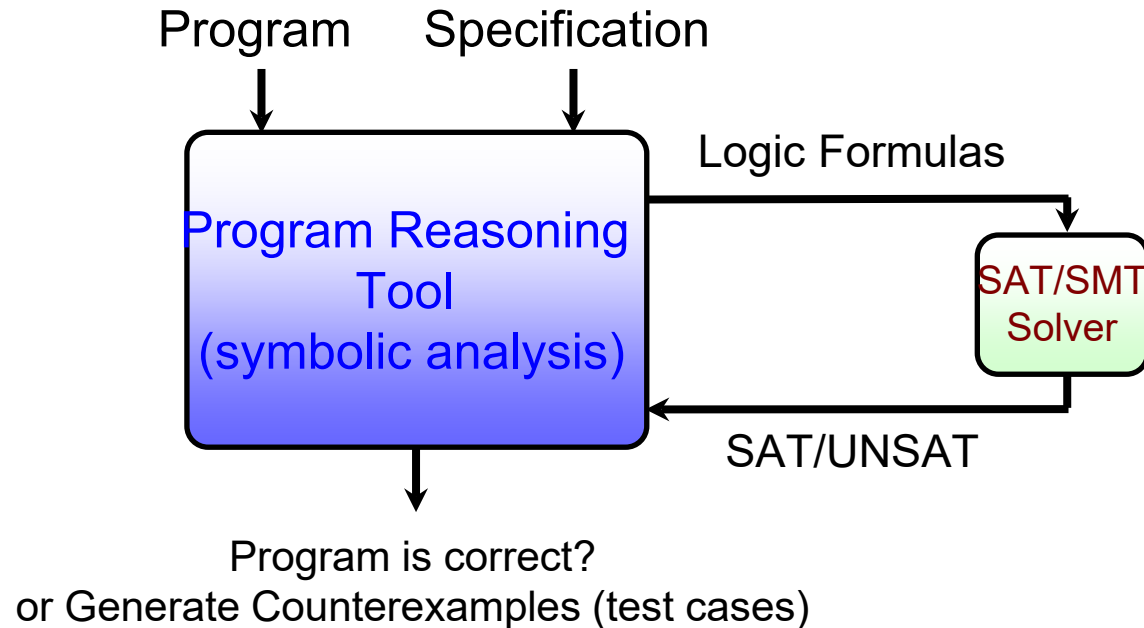
Software Engineering and SAT/SMT Solvers

An Indispensable Tactic for Any Strategy



Software Engineering using Solvers

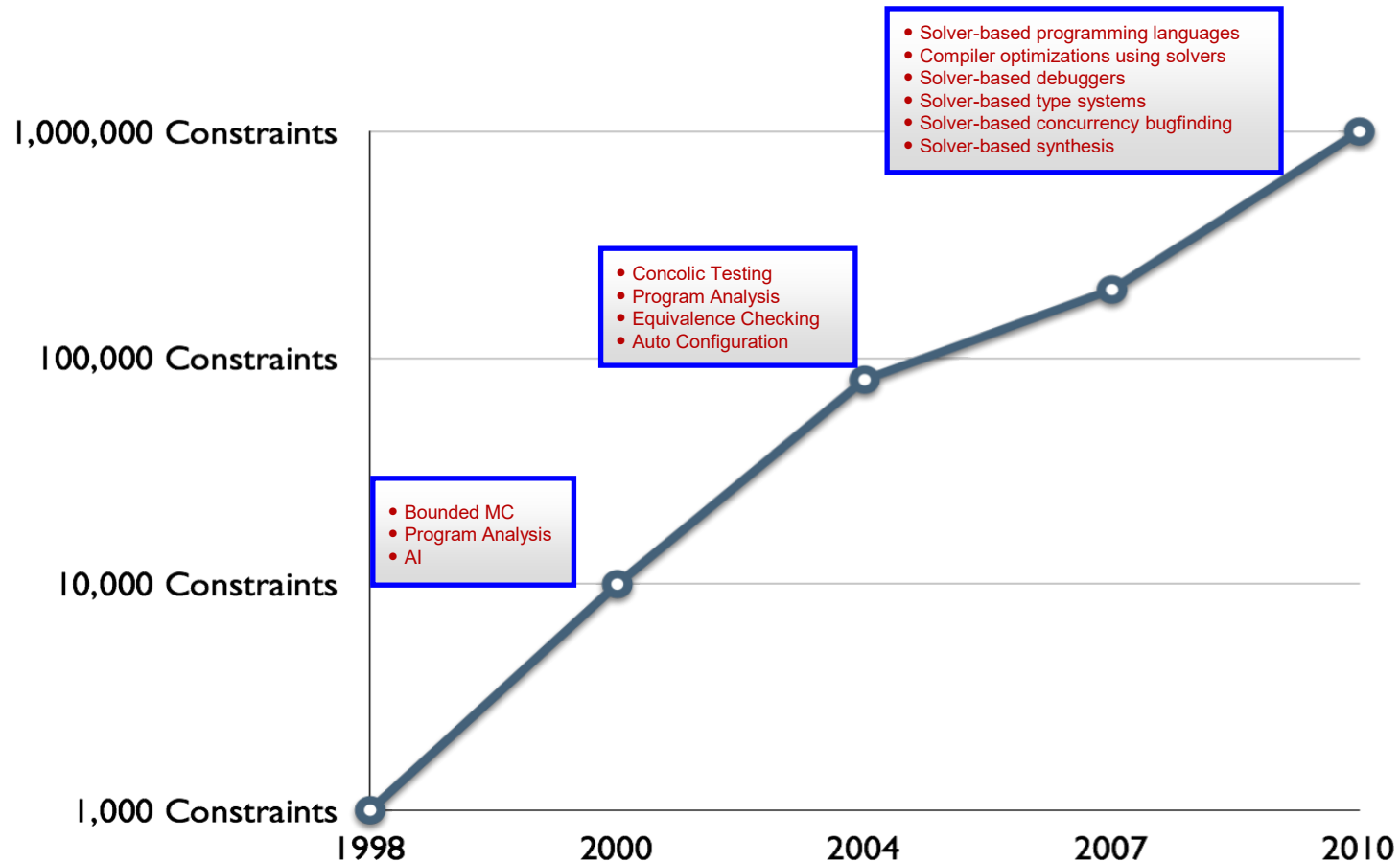
Engineering, Usability, Novelty





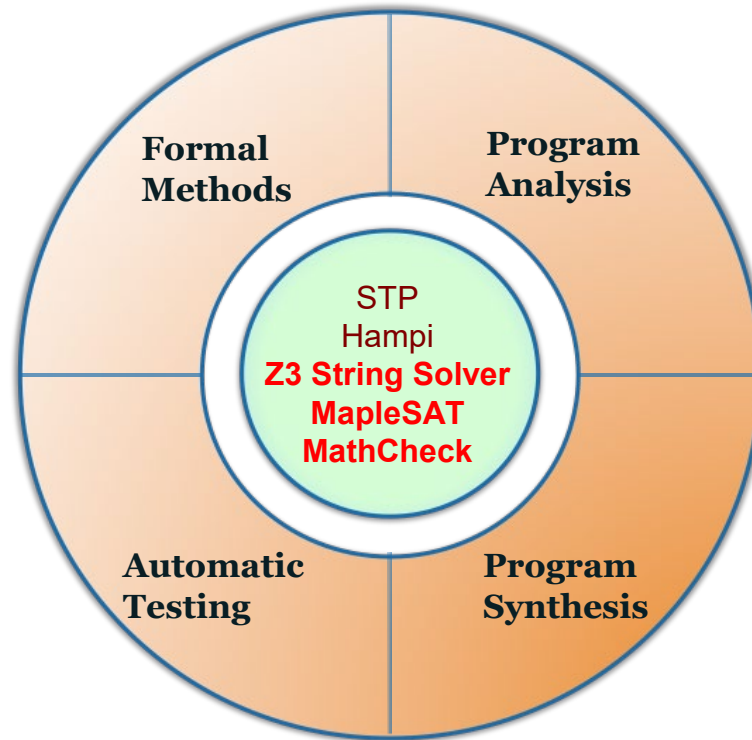
SAT/SMT Solver Research Story

A 1000x+ Improvement



IMPORTANT CONTRIBUTIONS

An Indispensable Tactic for Any Strategy




DEEPER DIVE INTO PROPOSITIONAL (BOOLEAN) LOGIC


Syntax and Semantics

Syntax

- MW: the way in which linguistic elements (such as words) are put together to form constituents (such as phrases or clauses)
- Determines and restricts how things are written

[[SEMANTICS]]
of a structure

[[

[[

Semantics

- MW: the study of meanings
- Determines how syntax is interpreted to give meaning

Syntax of Propositional Logic

Constants: True, False

Propositional variables: A_1, A_2, \dots ranges over $\{T, F\}$

An *atomic formula* has a form A_i , where $i = 1, 2, 3 \dots$

Formulas are defined **inductively** as follows:

- All atomic formulas are formulas
- For every formula F , $\neg F$ (called not F) is a formula
- For all formulas F, G , then $F \wedge G$ (called and) and $F \vee G$ (called or) are formulas

Abbreviations

- use A, B, C, \dots instead of A_1, A_2, \dots
- use $F_1 \rightarrow F_2$ instead of $\neg F_1 \vee F_2$ (implication)
- use $F_1 \leftrightarrow F_2$ instead of $(F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)$ (iff)

CFG or Formal Syntax of Propositional Logic

constant ::= true | false

variable ::= p | q | r | A | B | C | A_0 | A_1 | ...

atom ::= constant | variable

literal ::= atom | \neg atom

formula ::= literal |
 \neg formula |
 formula \wedge formula |
 formula \vee formula |
 (formula)

Example

$$F = \neg((A_5 \wedge A_6) \vee \neg A_3)$$

Sub-formulas are

$$\begin{aligned} &F, ((A_5 \wedge A_6) \vee \neg A_3), \\ &A_5 \wedge A_6, \neg A_3, \\ &A_5, A_6, A_3 \end{aligned}$$

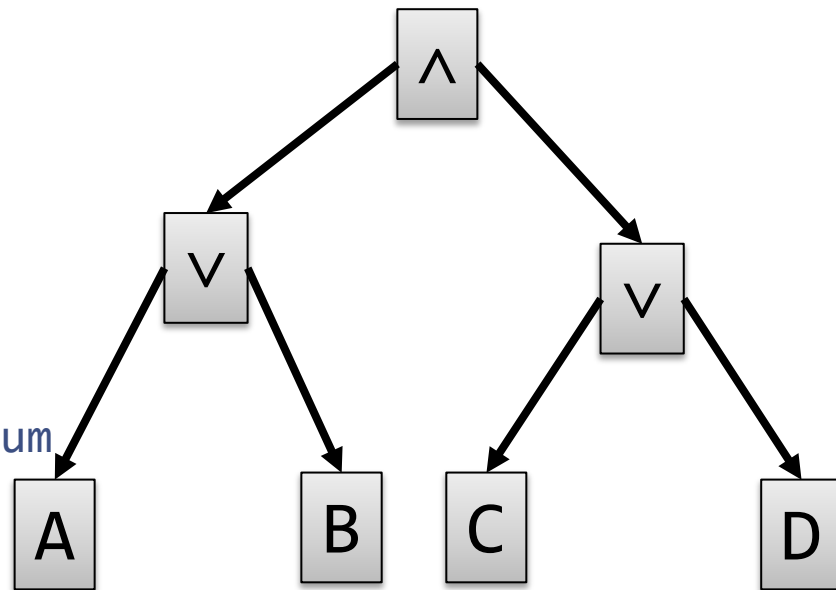
Abstract Syntax Tree (AST)

AST is an abstract tree representation of formulas

- each node represents a syntactic construct occurring in the formula
 - e.g., variable, operator, etc.
- called “**abstract**” because some details of concrete syntax are omitted
 - AST normalizes (provides common representation) of irrelevant differences in syntax (e.g., white space, order of operations)
- example AST: $(A \vee B) \wedge (C \vee D)$

```
post-order(T) {  
    if(T == NULL) return;  
    post-order (T->left);  
    post-order(T->right);  
    return process();  
}
```

Complexity is linear in num
Of variables



Semantics of propositional logic

1/2

Start with two truth values: $\{0, 1\}$

- 0 stands for false, and 1 stands for true

Let \mathbf{D} be any subset of the *atomic* formulas

An *assignment* \mathbf{A} is a map $\mathbf{D} \rightarrow \{0, 1\}$

- \mathbf{A} assigns true/false to every atomic in \mathbf{D}

Let $\mathbf{E} \supseteq \mathbf{D}$ be a set of formulas built from \mathbf{D} using propositional connectives

Extended assignment \mathbf{A}' : $\mathbf{E} \rightarrow \{0, 1\}$ extends \mathbf{A} from atomic formulas to all formulas

Semantics of propositional logic

2/2

For an atomic formula A_i in \mathbf{D} : $\mathbf{A}'(A_i) = \mathbf{A}(A_i)$

$$\begin{aligned} \mathbf{A}'(F \wedge G) &= 1 && \text{if } \mathbf{A}'(F) = 1 \text{ and } \mathbf{A}'(G) = 1 \\ &= 0 && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathbf{A}'(F \vee G) &= 1 && \text{if } \mathbf{A}'(F) = 1 \text{ or } \mathbf{A}'(G) = 1 \\ &= 0 && \text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathbf{A}'(\neg F) &= 1 && \text{if } \mathbf{A}'(F) = 0 \\ &= 0 && \text{otherwise} \end{aligned}$$

Exercise: Define Extended Assignment

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

$$\mathcal{A}(A) = 1$$

$$\mathcal{A}(B) = 1$$

$$\mathcal{A}(C) = 0$$

Is F true or false under \mathcal{A} '?

Truth Tables for Basic Operators

$\mathcal{A}(F)$	$\mathcal{A}(G)$	$\mathcal{A}'(F \wedge G)$
0	0	0
0	1	0
1	0	0
1	1	1

$\mathcal{A}(F)$	$\mathcal{A}(G)$	$\mathcal{A}'(F \vee G)$
0	0	0
0	1	1
1	0	1
1	1	1

An extended assignment \mathcal{A}' extends the truth table from atomic propositions to propositional formulas

$\mathcal{A}(F)$	$\mathcal{A}'(\neg F)$
0	1
1	0

Formula

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

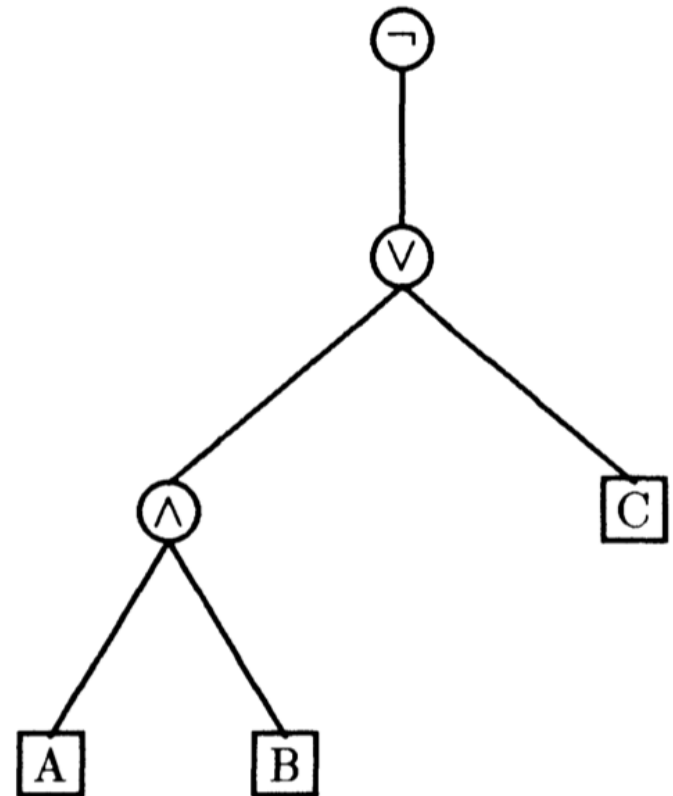
Assignment

$$\mathcal{A}(A) = 1$$

$$\mathcal{A}(B) = 1$$

$$\mathcal{A}(C) = 0$$

Abstract Syntax Tree (AST)



Propositional Logic: Semantics

An assignment A is *suitable/complete* for a formula F if A assigns a truth value to every atomic proposition of F

An assignment A is a *model* for F , written $A \models F$, iff

- A is suitable for F
- $A'(F) = 1$, i.e., F *evaluates to true (or holds)* under A

A formula F is *satisfiable* iff there exists at least one model for F . Otherwise, we say F is *unsatisfiable* (or contradictory)

A formula F is *valid* (or a tautology), written $\models F$, iff every suitable assignment for F is a model for F

Determining Satisfiability via a Truth Table

A formula F with n atomic sub-formulas has 2^n suitable assignments

Build a truth table enumerating all assignments

F is satisfiable iff there is at least one entry with 1 in the output

	A_1	A_2	\dots	A_{n-1}	A_n	F
$\mathcal{A}_1:$	0	0		0	0	$\mathcal{A}_1(F)$
$\mathcal{A}_2:$	0	0		0	1	$\mathcal{A}_2(F)$
\vdots			\ddots			\vdots
$\mathcal{A}_{2^n}:$	1	1		1	1	$\mathcal{A}_{2^n}(F)$

Problem: Is formula F SAT?

$$F = (\neg A \rightarrow (A \rightarrow B))$$

A	B	$\neg A$	$(A \rightarrow B)$	F
0	0	1	1	1
0	1	1	1	1
1	0	0	0	1
1	1	0	1	1

Validity and Unsatisfiability

Theorem:

- A formula F is valid if and only if $\neg F$ is unsatisfiable
- A formula F is not valid (invalid) if and only if $\neg F$ is satisfiable

Proof:

F is valid \Leftrightarrow every suitable assignment for F is a model for F
 \Leftrightarrow every suitable assignment for $\neg F$ is not a model for $\neg F$
 $\Leftrightarrow \neg F$ does not have a model
 $\Leftrightarrow \neg F$ is unsatisfiable

How to check the correctness of a combinational circuit C w.r.t a logical specification ϕ (Boolean logic)?

- Convert C into an equivalent Boolean formula F (n -input, 1-output)
- $(y \Leftrightarrow F(x_1, \dots, x_n)) \text{ AND } \phi(x_1, \dots, x_n, y)$ is given as input to a SAT solver
- SAT solver returns satisfiable (bug-finder)
- SAT solver returns UNSAT (F is correct w.r.t ϕ)

NORMAL FORMS



SINCE 1828

JOIN MWU

GAMES

BROWSE THESAURUS

WORD OF THE DAY

WORDS AT PLAY

normal form

DICTIONARY

THESAURUS

normal form noun

Definition of *normal form*

logic

: a canonical or standard fundamental form of a statement to which others can be reduced

especially : a compound statement in the propositional calculus consisting of nothing but a conjunction of disjunctions whose disjuncts are either elementary statements or negations thereof

<https://www.merriam-webster.com/dictionary/normal%20form>

Negation Normal Form (NNF)

A formula F is in **Negation Normal Form** (NNF) if every occurrence of negation (\neg) in F is applied to an atomic sub-formula of F .

For example,

- $\neg a \vee \neg b \vee c$ is in NNF
- $\neg (a \wedge b \wedge \neg c)$ is NOT in NNF (why?)

Theorem (NNF): For every formula F there is a semantically equivalent form G in NNF. In symbols

- For every F , exists G in NNF, such that $F \equiv G$

Proof of NNF Theorem

By **structural induction** on the structure of a formula F

(base case): atomic formulas A and $\neg A$ are in NNF

(IH): Assume that the theorem is true for every sub-formula of F : For every sub-formula H of F there exists J in NNF such that $J \equiv H$. Show that there exists G in NNF such that $G \equiv F$

(\neg case): Assume $F = \neg H$. Then, $F \equiv \neg(\neg J) \equiv J$

(\vee case): Assume $F = H_1 \vee H_2$. Then, $F \equiv J_1 \vee J_2$

(\wedge case): Assume $F = H_1 \wedge H_2$. Then, $F \equiv J_1 \wedge J_2$

Normal Form: DNF

A *literal* is either an atomic proposition v or its negation $\neg v$

A *cube* is a conjunction of literals

- e.g., $(v1 \wedge \neg v2 \wedge v3)$

A formula F is in *Disjunctive Normal Form* (DNF) if F is a disjunction of conjunctions of literals

$$\bigvee_{i=1}^n \left(\bigwedge_{j=1}^{m_i} L_{i,j} \right)$$

(Fun) Fact: determining whether a DNF formula F is satisfiable is easy

- easy == linear in the size of the formula

Normal Form: CNF

A *literal* is either an atomic proposition v or its negation $\neg v$

A *clause* is a disjunction of literals

- e.g., $(v1 \vee \neg v2 \vee v3)$

A formula F is in *Conjunctive Normal Form* (CNF) if F is a conjunction of disjunctions of literals

$$\bigwedge_{i=1}^n \left(\bigvee_{j=1}^{m_i} L_{i,j} \right)$$

(Fun) Fact: determining whether a CNF formula F is satisfiable is hard

- hard == NP-complete

Normal Form Theorem

Theorem: For every formula F , there is an equivalent formula F_1 in CNF, and an equivalent formula F_2 in DNF.

That is, CNF and DNF are normal forms:

- Every propositional formula can be converted to CNF and to DNF without affecting its meaning (i.e., semantics)!

Proof: (by induction on the structure of the formula F)

Details are left as an exercise!

Converting a formula to CNF

Given a formula F

1. Substitute in F every occurrence of a sub-formula of the form

$\neg\neg G$ by G

$\neg(G \wedge H)$ by $(\neg G \vee \neg H)$

$\neg(G \vee H)$ by $(\neg G \wedge \neg H)$

The result is a formula in Negation Normal Form (NNF)

2. Substitute in F each occurrence of a sub-formula of the form

$(F \vee (G \wedge H))$ by $((F \vee G) \wedge (F \vee H))$

$((F \wedge G) \vee H)$ by $((F \vee H) \wedge (G \vee H))$

The resulting formula F is in CNF

- the result in CNF might be exponentially bigger than original formula F

Example: From Truth Table to CNF and DNF

DNF

$$\begin{aligned} &(\neg A \wedge \neg B \wedge \neg C) \vee \\ &(A \wedge \neg B \wedge \neg C) \vee \\ &(A \wedge \neg B \wedge C) \end{aligned}$$

CNF

$$\begin{aligned} &(A \vee B \vee \neg C) \wedge \\ &(A \vee \neg B \vee C) \wedge \\ &(A \vee \neg B \vee \neg C) \wedge \\ &(\neg A \vee \neg B \vee C) \wedge \\ &(\neg A \vee \neg B \vee \neg C) \end{aligned}$$

Truth table

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

From Truth Table to DNF / CNF

From a truth table T to DNF formula F

- For every row i of T with value 1, construct a conjunction F_i that characterizes the row. That is, F_i is a formula that is true **exactly** for the assignment of row i
- Let $F = \vee F_i$, then F is equivalent to T and is in DNF
- Fact: F has as many disjuncts as rows with value 1 in T
- Question: How big can F be? How small can F be?

From a truth table T to CNF formula G

- For every row i of T with value 0, construct a conjunction F_i that characterizes the row
- Let G_i be NNF of $\neg F_i$
- Let $G = \wedge G_i$, then G is equivalent to T and is in CNF
- Fact: F has as many disjuncts as rows with value 0 in T
- Question: How big can F be? How small can F be?



2-CNF Fragment

A formula F is in 2-CNF iff

- F is in CNF
- every clause of F has at most 2 literals

Theorem: There is a polynomial algorithm for deciding whether a 2-CNF formula F is satisfiable

Horn Fragment

A formula F is in Horn fragment iff

- F is in CNF
- in every clause, at most one literal is positive

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

- Note that each clause can be written as an implication
 - e.g. $C \wedge A \Rightarrow D$, $A \wedge B \Rightarrow \text{False}$, $\text{True} \Rightarrow D$

$$(B \rightarrow A) \wedge (A \wedge C \rightarrow D) \wedge (A \wedge B \rightarrow 0) \wedge (1 \rightarrow D) \wedge (E \rightarrow 0)$$

Theorem: There is a polynomial time algorithm for deciding satisfiability of a Horn formula F

Horn Satisfiability

Input: a Horn formula F

Output: UNSAT or SAT + satisfying assignment for F

Step 1: Mark every occurrence of an atomic formula A in F if there is an occurrence of sub-formula of the form A in F

Step 2: pick a formula G in F of the form $A_1 \wedge \dots \wedge A_n \rightarrow B$ such that all of A_1, \dots, A_n are already marked

- if $B = 0$, return UNSAT
- otherwise, mark B and go back to Step 2

Step 3: Construct a suitable assignment S such that $S(A_i) = 1$ iff A_i is marked. Return SAT with a satisfying assignment S .

Exercise 21

Apply Horn satisfiability algorithm on a formula

$$(\neg A \vee \neg B \vee \neg D)$$

$$\neg E$$

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

$$C$$

$$B$$

$$(\neg G \vee D)$$

$$G$$

COMPLEXITY OF SAT

3-CNF Fragment

A formula F is in 3-CNF iff

- F is in CNF
- every clause of F has at most 3 literals

Theorem: Deciding whether a 3-CNF formula F is satisfiable is at least as hard as deciding satisfiability of an arbitrary CNF formula G

Proof: by effective *reduction* from CNF to 3-CNF

Let G be an arbitrary CNF formula. Replaced every clause of the form

$$(\ell_0 \vee \cdots \vee \ell_n)$$

with 3-literal clauses

$$(\ell_0 \vee b_0) \wedge (\neg b_0 \vee \ell_1 \vee b_1) \wedge \cdots \wedge (\neg b_{n-1} \vee \ell_n)$$

where $\{b_i\}$ are fresh atomic propositions not appearing in F

Complexity of 3-CNF Satisfiability

Theorem (Cook-Levin): The Boolean Satisfiability Problem is NP-complete

Consequences

- If a formula F is satisfiable, then there exists a certificate for satisfiability that can be checked in P (polynomial) time.
 - That is, checking solutions is easy
- Any other problem that has polynomial certificates is polynomial reducible to Boolean Satisfiability
 - That is, such problems can be solved by writing a loop-free program, compiling it to a Boolean circuit, and checking whether the circuit ever accepts some input
- **MANY MANY MANY OPTIMIZATION PROBLEMS ARE LIKE THAT**
- Boolean Satisfiability is easy iff $P = NP$
 - i.e., Boolean satisfiability is believed to be a very hard problem in general!

Background Reading: SAT

←

→

http://cacm.acm.org/magazines/2009/8/34498-boolean-satisfiability-from-theoretical-h

Boolean Satisfiability: From ...

Find: currency

Previous

Next

Options

TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS

ACM.org | Join ACM | About Communications | ACM Resources | Alerts & Feeds

SIGN IN

COMMUNICATIONS
OF THE
ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRACTICE | CAREERS | MAGAZINE ARCHIVE

Search





Home / Magazine Archive / August 2009 (Vol. 52, No. 8) / Boolean Satisfiability: From Theoretical Hardness... / Full Text

REVIEW ARTICLES








Boolean Satisfiability: From Theoretical Hardness to Practical Success


By Sharad Malik, Lintao Zhang
Communications of the ACM, Vol. 52 No. 8, Pages 76-82
10.1145/1536616.1536637
[Comments](#)

VIEW AS:



SHARE:





There are many practical situations where we need to satisfy several potentially conflicting constraints. Simple examples of this abound in daily life, for example, determining a schedule for a series of games that resolves the availability of players and venues, or finding a seating assignment at dinner consistent with various rules the host would like to impose. This also applies to applications in computing, for example, ensuring that a hardware/software system functions correctly with its overall behavior constrained by the behavior of its components and their composition, or finding a plan for a robot to reach a goal that is

SIGN IN for Full Access

User Name

Password

[» Forgot Password?](#)

[» Create an ACM Web Account](#)

SIGN IN

ARTICLE CONTENTS:

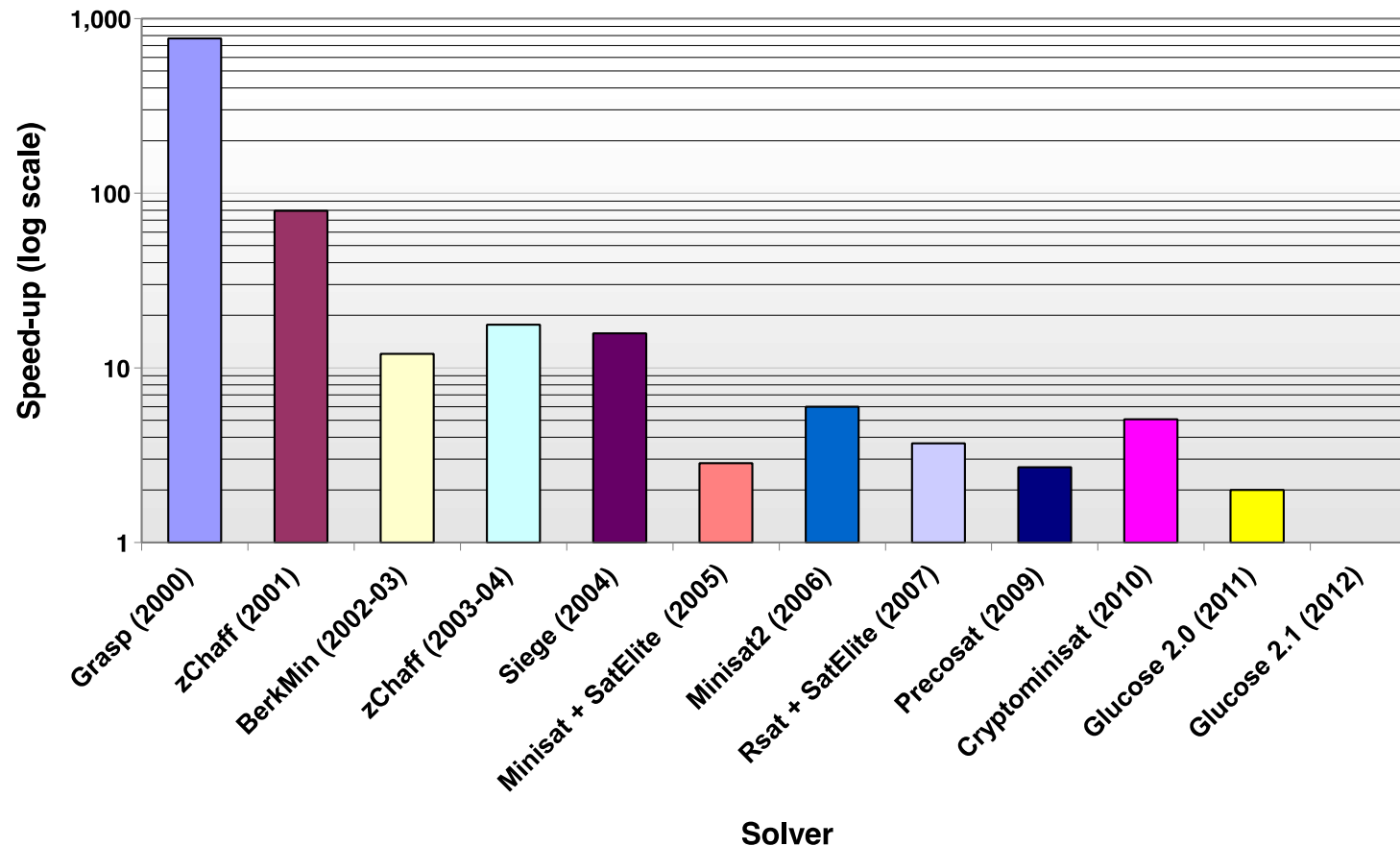
[Introduction](#)

[Boolean Satisfiability](#)

[Theoretical hardness: SAT and NP Completeness](#)

Some Experience with SAT Solving

Speed-up of 2012 solver over other solvers



from M. Vardi, <https://www.cs.rice.edu/~vardi/papers/highlights15.pdf>

SAT - Milestones

Problems impossible 10 years ago are trivial today

year	Milestone
1960	Davis-Putnam procedure
1962	Davis-Logeman-Loveland
1984	Binary Decision Diagrams
1992	DIMACS SAT challenge
1994	SATO: clause indexing
1997	GRASP: conflict clause learning
1998	Search Restarts
2001	zChaff: 2-watch literal, VSIDS
2005	Preprocessing techniques
2007	Phase caching
2008	Cache optimized indexing
2009	In-processing, clause management
2010	Blocked clause elimination

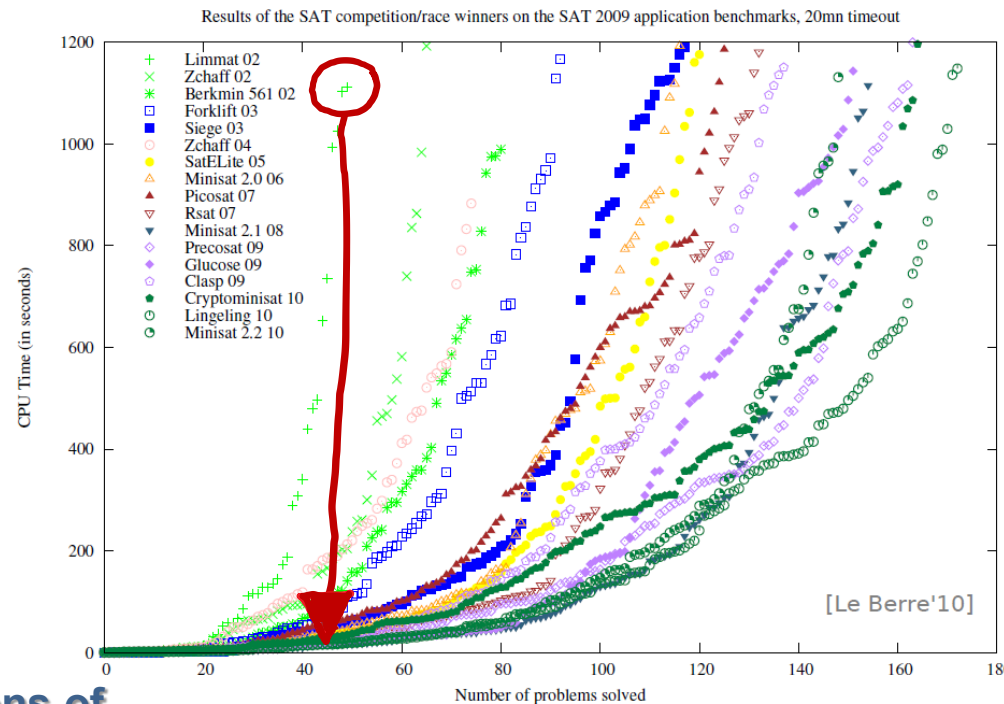
Concept



**Millions of
variables from
HW designs**

2002

2010



Courtesy Daniel le Berre

ENCODING PROBLEMS TO SAT

Graph k-Coloring

Given a graph $G = (V, E)$, and a natural number $k > 0$ is it possible to assign colors to vertices of G such that no two adjacent vertices have the same color.

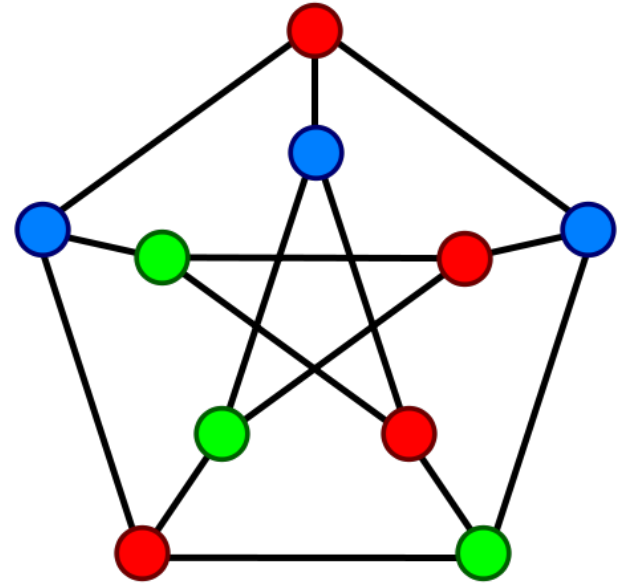
Formally:

- does there exist a function $f : V \rightarrow [0..k)$ such that
- for every edge (u, v) in E , $f(u) \neq f(v)$

Graph coloring for $k > 2$ is NP-complete

Problem: Encode k-coloring of G into CNF

- construct CNF C such that C is SAT iff G is k-colorable





k -coloring as CNF

Let a Boolean variable $f_{v,i}$ denote that vertex v has color i

- if $f_{v,i}$ is true if and only if $f(v) = i$

Every vertex has at least one color

$$\bigvee_{0 \leq i < k} f_{v,i} \quad (v \in V)$$

No vertex is assigned two colors

$$\bigwedge_{0 \leq i < j < k} (\neg f_{v,i} \vee \neg f_{v,j}) \quad (v \in V)$$

No two adjacent vertices have the same color

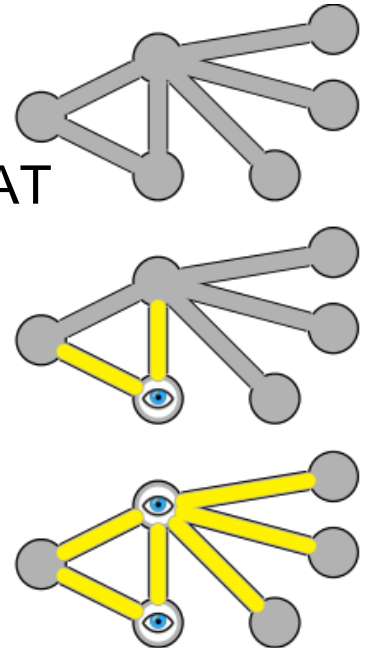
$$\bigwedge_{0 \leq i < k} (\neg f_{v,i} \vee \neg f_{u,i}) \quad ((v, u) \in E)$$

Vertex Cover

Given a graph $G=(V,E)$. A vertex cover of G is a subset C of vertices in V such that every edge in E is incident to at least one vertex in C

see a4_encoding.pdf for details of reduction to CNF-SAT

- will be given together with assignment 4



USING A SAT SOLVER

DIMACS interface to a SAT Solver

Input:

- a CNF in DIMACS format

Output:

- SAT/UNSAT + satisfying assignment

We will use a SAT solver called MiniSAT

- available at <https://github.com/agurfinkel/minisat>
- written in C++
- use as a library in Assignment 4
- use via DIMACS interface today in class
- MiniSat examples:
 - <https://github.com/eceuwaterloo/ece650-minisat>

DIMACS CNF File Format

Textual format to represent CNF-SAT problems

```
c start with comments
c
c
p cnf 5 3
 1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Details

- comments start with **c**
- header line: **p cnf nbvar nbclauses**
 - **nbvar** is # of variables, **nbclauses** is # of clauses
- each clause is a sequence of distinct numbers terminating with **0**
 - **positive** numbers are variables, **negative** numbers are negations

MiniSat

MiniSat is one of the most famous modern SAT-solvers

- written in C++
- designed to be easily understandable and customizable
- many new SAT-solvers use MiniSAT as their base

Web page: <http://minisat.se/>

We will use a slightly updated version from GitHub:

<https://github.com/agurfinkel/minisat>

Good references for understanding SAT solving details

- MiniSat architecture: <http://minisat.se/downloads/MiniSat.pdf>
- Donald Knuth's SAT13 (also based on MiniSat)
 - <http://www-cs-faculty.stanford.edu/~knuth/programs/sat13.w>

<https://git.uwaterloo.ca/ece650-1209/minisat>

MINISAT EXAMPLES