Satisfiability Checking for Propositional Logic

Viktor Kuncak, EPFL

https://lara.epfl.ch/w/fv

Propositional (Boolean) Logic

Propositional logic is a language for representing Boolean functions $f : \{0,1\}^n \to \{0,1\}$.

- sometimes we write $\bot$ for 0 and $\top$ for 1

Grammar of formulas:

$$P ::= x \mid 0 \mid 1 \mid P \land P \mid \neg P \mid P \lor P \mid P \oplus P \mid P \to P \mid P \leftrightarrow P$$

where x denotes variables (identifiers). Corresponding Scala trees:

```
sealed abstract class Expr
case class Var(id: Identifier) extends Expr
case class BooleanLiteral(b: Boolean) extends Expr
case class And(e1: Expr, e2: Expr) extends Expr
case class Or(e1: Expr, e2: Expr) extends Expr
case class Not(e: Expr) extends Expr
...
```

Environment and Truth of a Formula

An environment e is a partial map from propositional variables to $\{0,1\}$
For vector of n boolean variables $\bar{p} = (p_1, \ldots, p_n)$ and $\bar{v} = (v_1, \ldots, v_n) \in \{0,1\}^n$, we denote $[\bar{p} \mapsto \bar{v}]$ the environment e given by $e(p_i) = v_i$ for $1 \leq i \leq n$.
We write $e \models F$, and define $[\![F]\!]_e = 1$, to denote that F is true in environment e, otherwise define $[\![F]\!]_e = 0$
Let $e = \{(a,1),(b,1),(c,0)\}$ and F be $a \wedge (\neg b \vee c)$. Then:

$$[\![a \wedge (\neg b \vee c)]\!]_e \;=\; e(a) \wedge (\neg e(b) \vee e(c)) \;=\; 1 \wedge (\neg 1 \vee 0) \;=\; 0$$

The general definition is recursive:

$$\begin{array}{rcl} [\![x]\!]_e & = & e(x) \\ [\![0]\!]_e & = & 0 \\ [\![1]\!]_e & = & 1 \\ [\![F_1 \wedge F_2]\!]_e & = & [\![F_1]\!]_e \wedge [\![F_2]\!]_e \\ [\![\neg F_1]\!]_e & = & \neg [\![F_1]\!]_e \end{array}$$

Note: $\wedge$ and $\neg$ on left and right are different things

Truth of a Formula in Scala

The `interpret` method in `Expr.scala` of Labs 02:

```
def interpret(env: Map[Identifier, Boolean]): Boolean = this match {
  case Var(id) => env(id)
  case BooleanLiteral(b) => b
  case Equal(e1, e2) => e1.interpret(env) == e2.interpret(env)
  case Implies(e1, e2) => !e1.interpret(env) || e2.interpret(env)
  case And(e1, e2) => e1.interpret(env) && e2.interpret(env)
  case Or(e1, e2) => e1.interpret(env) || e2.interpret(env)
  case Xor(e1, e2) => e1.interpret(env) ^ e2.interpret(env)
  case Not(e) => !e.interpret(env)
}
```

Satisfiability Problem

Formula F is *satisfiable*, iff there **exists** e such that $[\![F]\!]_e = 1$.
Otherwise we call F *unsatisfiable*: when there does not exist e such that $[\![F]\!]_e = 1$, that is, for all e, $[\![F]\!]_e = 0$.
Example: let F be $a \wedge (\neg b \vee c)$. Then F is satisfiable, with e.g. $e = \{(a,1),(b,0),(c,0)\}$
Its negation of $\neg F$, is also satisfiable, with e.g. $e = \{(a,0),(b,0),(c,0)\}$

SAT is a problem: given a propositional formula, determine whether it is satisfiable.

The problem is decidable because given F we can compute its variables $FV(F)$ and it suffices to look at the 2^n environments for $n = FV(F)$. The problem is NP-complete, but useful heristics exist.

A SAT solver is a program that, given boolean formula F, either:

- returns **sat**, and, optionally, returns one environment e such that $[\![F]\!]_e = 1$, or
- returns **unsat** and, optionally, returns a **proof** that no satisfying assignment exists

Formal Proof System

We will consider a some set of logical formulas $\mathscr{F}$ (e.g. propositional logic)

Definition

An proof system is $(\mathscr{F}, \text{Infer})$ where $\text{Infer} \subseteq \mathscr{F}^* \times \mathscr{F}$ a decidable set of *inference steps*.

- a set is *decidable* iff there is a program to check if an element belongs to it
- given a set S, notation S^* denotes all finite sequences with elements from S

We schematically write an inference step $((P_1, \ldots, P_n), C) \in \text{Infer}$ by

$$\frac{P_1 \ldots P_n}{C}$$

and we say that from $P_1, \ldots, P_n$ (**premises**) we derive C (**conclusion**).
An inference step is called an *axiom instance* when $n = 0$ (it has no premises).
Given a proof system $(\mathscr{F}, \text{Infer})$, a proof is a finite sequence of inference steps such that, for every inference step, each premise is a conclusion of a previous step.

Proof in a Proof System

Definition

Given $(\mathscr{F}, \text{Infer})$ where $\text{Infer} \subseteq \mathscr{F}^* \times \mathscr{F}$ a **proof** in $(\mathscr{F}, \text{Infer})$ is a finite sequence of inference steps $S_0, \ldots, S_m \in \text{Infer}$ such that, for each S_i where $0 \leq i \leq m$, for each premise P_j of S_i there exists $0 \leq k < i$ such that P_j is the conclusion of S_k.

$$
\begin{array}{lll}
S_0 : & ((), & C_0) \\
 & \cdots & \\
S_k : & ((\ldots\ldots\ldots), & \mathbf{P_j}) \\
 & \cdots & \\
S_i : & ((\ldots, \mathbf{P_j}, \ldots), & C_i)
\end{array}
$$

Given the definition of the proof, we can replace each premise P_j with the index k where P_j was the conclusion of S_k ($P_j \equiv \text{Conc}(S_k)$)
A proof is then a sequence of elements from $\{0, 1, \ldots\}^* \times \mathscr{F}$ where each S_i in the sequence is of the form $(k_1, \ldots, k_n, C)$ for $0 \leq k_1, \ldots, k_n < i$ and where $(\text{Conc}(S_{k_1}), \ldots, \text{Conc}(S_{k_n}), C) \in \text{Infer}$.

Proofs as Dags

We can view proofs as directed acyclic graphs.

Given a proof as a sequence of steps, for each $(k_1, \ldots, k_n, C)$ in the sequence we introduce a node labelled by C, and directed labelled edges $(\mathrm{Conc}(S_{k_j}), j, C)$ for all premises $k_1, \ldots, k_n$.

To check such proof, for each node, follow all of its incoming edges backwards in the order of their indices to find the premises, then check that the inference step is in Infer.

A Minimal Propositional Logic Proof System

Formulas $\mathscr{F}$ defined by $F ::= x \mid 0 \mid F \to F$

Shorthand:
$\neg F \equiv F \to 0$

Inference rules: Infer $= P_2 \cup P_3 \cup \text{MP}$ where: (W: Hilbert system)

$$
\begin{array}{rcll}
P_2 & = & \{((), & F \to (G \to F) \quad) \mid F, G \in \mathscr{F}\} \\
P_3 & = & \{((), & ((F \to (G \to H)) \to ((F \to G) \to (F \to H)) \quad) \mid F, G, H \in \mathscr{F}\} \\
\text{MP} & = & \{((F \to G, F), & G \quad) \mid F, G \in \mathscr{F}\}
\end{array}
$$

Elements of P_1, P_2, P_3 are all axioms. These are infinite sets, but are given a schematic way and there is an algorithm to check if a given formula satisfies each of the schemas.

Exercise: draw a DAG representing proof of $a \to a$ where a is a propositional variable.

An Example Proof

Hint: use P_3 for $F \equiv a$, $G \equiv a \rightarrow a$, $H \equiv a$

An Example Proof

Hint: use P_3 for $F \equiv a$, $G \equiv a \rightarrow a$, $H \equiv a$
Apply MP to the above instance of P_3 and an instance of P_2, then to another instance of P_2.

Derivation is a Proof from Assumptions

Definition

Given $(\mathscr{F}, \text{Infer})$, $\text{Infer} \subseteq \mathscr{F}^* \times \mathscr{F}$ **and a set of assumptions** $A \subseteq \mathscr{F}$, a **derivation from** A in $(\mathscr{F}, \text{Infer})$ is a proof in $(\mathscr{F}, \text{Infer}')$ where:

$$\text{Infer}' = \text{Infer} \cup \{((), F) \mid F \in A\}$$

Thus, assumptions from A are treated just as axioms.

Definition

We say that $F \in \mathscr{F}$ is provable from assumptions A, denoted $A \vdash_{\text{Infer}} F$ iff there exists a derivation from A in Infer that contains an inference step whose conclusion is F.

We write $\vdash_{\text{Infer}} F$ to denote that there exists a proof in Infer containing F as a conslusion (same as $\emptyset \vdash_{\text{Infer}} F$).

Consequence and Soundness in Propositional Logic

Given a set $A \subseteq \mathscr{F}$ where $\mathscr{F}$ are in propositional logic, and $C \in \mathscr{F}$, we say that C is a **semantic consequence** of A, denoted $A \models C$ iff for every environment e that defines all variables in $FV(C) \cup \bigcup_{P \in A} FV(P)$, if $[\![P]\!]_e = 1$ for all $P \in A$, then then $[\![C]\!]_e = 1$.

Definition

Given $(\mathscr{F}, \text{Infer})$ where $\mathscr{F}$ are propositional, step $((P_1 \dots P_n), C) \in \text{Infer}$ is **sound** iff $\{P_1, \dots, P_n\} \models C$. Proof system Infer is sound if every inference step is sound.

For axioms, this definition reduces to saying that C is true for all interpretations, i.e., that C is a valid formula (tautology).

Theorem

Let $(\mathscr{F}, Infer)$ where $\mathscr{F}$ are propositional logic formulas. If every inference rule in Infer is sound, then $A \vdash_{Infer} F$ implies $A \models F$.

Proof is immediate by induction on the length of the formal proof.
Consequence: $\vdash_{\text{Infer}} F$ implies F is a tautology.

A Proof System with Decision and Simplification

Propositional formulas F and G are semantically equivalent if $F \models G$ and $G \models F$.

Case analysis proof rule $((F,G), F[x:=0] \vee G[x:=1]) \mid F, G \in \mathscr{F}, x-$variable$\}$:

$$\frac{F \qquad\qquad G}{F[x:=0] \vee G[x:=1]}$$

Proof of soundness: consider an environment e (that defines x as well as $FV(F) \cup FV(G)$), and assume $[\![F]\!]_e = 1$ and $[\![G]\!]_e = 1$.

- If $e(x) = 0$, then $[\![F[x:=0]]\!]_e = [\![F]\!]_e = 1$.
- If $e(x) = 1$, then $[\![G[x:=1]]\!]_e = [\![G]\!]_e = 1$.

Simplification rules that preserve equivalence can be applied: $0 \wedge F \rightsquigarrow 0$, $1 \wedge F \rightsquigarrow F$, $0 \vee F \rightsquigarrow F$, $1 \vee F \rightsquigarrow 1$, $\neg 0 \rightsquigarrow 1$, $\neg 1 \rightsquigarrow 0$.
Introduce inferences $\{((F), F') \mid F'$ is simplified $F\}$. These rules are also sound. Call this Infer_D.

Example Derivation

Derivation from $A = \{a \wedge b,\ \neg b \vee \neg a\}$. Draw the arrows to get a proof DAG

$a \wedge b$

$\neg b \vee \neg a$

$(0 \wedge b) \vee (1 \wedge b)$

$(a \wedge 0) \vee (a \wedge 1)$

b

a

$0 \vee (\neg 1 \vee \neg a)$

$\neg a$

$0 \vee (\neg 1)$

0

Example Derivation

Derivation from $A = \{a \wedge b,\ \neg b \vee \neg a\}$. Draw the arrows to get a proof DAG

$a \wedge b$

$\neg b \vee \neg a$

$(0 \wedge b) \vee (1 \wedge b)$

$(a \wedge 0) \vee (a \wedge 1)$

b

a

$0 \vee (\neg 1 \vee \neg a)$

$\neg a$

$0 \vee (\neg 1)$

0

This derivation shows that: $A \vdash 0$

Proving Unsatisfiability

A *set* A of formulas is *satisfiable* if there exists e such that, for every $F \in A$, $[\![F]\!]_e = 1$.

- when $A = \{F_1, \ldots, F_n\}$ the notion is the same as the satisfiability of $F_1 \wedge \ldots \wedge F_n$

Otherwise, we call the set A *unsatisfiable*.

Theorem (Soundness Consequence)

If $A \vdash_{Infer_D} 0$ then A is unsatisfiable.

If there exists e is such that $e(F) = 1$ for all $F \in A$ then by soundness of Infer_D, $e(0) = 1$, a contradiction. So there is no such e.

Theorem (Refutation Completeness)

If a finite set A is unsatisfiable, then $A \vdash_{Infer_D} 0$

Proof hint: take conjunction of formulas in A and existentially quantify it to get A'. What is the relationship of the truth of A' and the satisfiability of A? For a conjunction of formulas F, can you express $\exists x.F$ using Infer_D?

Illustration of Completeness

Let $A = \{F_1, F_2\}$ and let $FV(F_1) \cup FV(F_2) = \{x_1, \ldots, x_n\}$ and let x be some x_i
We have the following equivalences:

$$
\begin{array}{cl}
\exists x.(F_1 \land F_2) & \\
(F_1 \land F_2)[x := 0] \lor (F_1 \land F_2)[x := 1] & \text{try both values} \\
(F_1[x := 0] \land F_2[x := 0]) \lor (F_1[x := 1] \land F_2[x := 1]) & \text{meaning of substitution} \\
(F_1[x := 0] \lor F_1[x := 1]) \land (F_1[x := 0] \lor F_2[x := 1]) \land & \\
(F_2[x := 0] \lor F_1[x := 1]) \land (F_2[x := 0] \lor F_2[x := 1]) &
\end{array}
$$

Existentially quantifying over a variable gives us result of applying decision rule to all pairs of formulas F_1, F_2.
Systematically applying rules will derive formula Z equivalent to $\exists x_1. \ldots \exists x_n.(F_1 \land F_2)$.
When A is unsatisfiable, Z is equivalent to 0, and has no free variables. By simplification rules, we can derive 0.

Conjunctive Form, Literals, and Clauses

A propositional *literal* is either a variable (e.g., x) or its negation ($\neg x$).
A *clause* is a disjunction of literals.
For convenience, we can represent clause as a finite *set of literals*
(because of associativity, commutativity, and idempotence of $\vee$).

Example: $a \vee \neg b \vee c$ represented as $\{a, \neg b, c\}$

If C is a clause then $[\![C]\!]_e = 1$ iff there exists a literal $l \in C$ such that $[\![l]\!]_e = 1$. We represent 0 using the empty clause $\emptyset$.
As for any formulas, a finite set of clauses A can be interpreted as a conjunction.
Thus, a set of clauses can be viewed as a formula in conjunctive normal form:

$$A = \{\{a\},\ \{b\},\ \{\neg a, \neg b\}\}$$

represents the formula

$$a \wedge b \wedge (\neg a \vee \neg b)$$

Resolution on Clauses as a Proof System

$a \vee b \vee c$ $\quad$ $d \vee \neg c$

$(a \vee b \vee 0) \vee (d \vee \neg 1))$

$a \vee b \vee d$

$\{a, b, c\}$ $\quad$ $\{d, \neg c\}$

$\{a, b, d\}$

Clausal resolution rule (decision rule for clauses):

$$\frac{C_1 \cup \{x\} \quad C_2 \cup \{\neg x\}}{C_1 \cup C_2}$$

resolve two clauses with respect to x

Theorem (Soundness)

Clausal resolution is sound for all clauses C_1, C_2 and propositional variable x, $\{C_1 \cup \{x\}, C_2 \cup \{\neg x\}\} \models C_1 \cup C_2$.

Theorem (Refutational Completeness)

A finite set of clauses A is satisfiable if and only if there exists a derivation of the empty clause from A using clausal resolution.

Resolution as Transitivity of Implication

For three formulas F_1, F_2, F_3 if $F_1 \rightarrow F_2$ and $F_2 \rightarrow F_3$ are true, so is $F_1 \rightarrow F_3$.
Thus, $\rightarrow$ denotes a transitive relation on $\{0, 1\}$.

We can view resolution as a consequence of transitivity.
We use the fact that $P \rightarrow Q$ is equivalent to $\neg P \vee Q$:

$$\frac{C_1 \vee x \qquad C_2 \vee \neg x}{C_1 \vee C_2} \qquad\qquad \frac{(\neg C_1) \rightarrow x \qquad x \rightarrow C_2}{(\neg C_1) \rightarrow C_2}$$

Exercise

Use resolution to prove that the following formula is valid:

$$\neg(a \wedge b \wedge (\neg a \vee \neg b))$$

Exercise

Use resolution to prove that the following formula is valid:

$$\neg(a \wedge b \wedge (\neg a \vee \neg b))$$

Prove that its negation is unsatisfiable set of clauses:

$\{a\}$ $\{b\}$ $\{\neg a, \neg b\}$

Exercise

Use resolution to prove that the following formula is valid:

$$\neg(a \wedge b \wedge (\neg a \vee \neg b))$$

Prove that its negation is unsatisfiable set of clauses:

$\{a\}$ $\{b\}$ $\{\neg a, \neg b\}$

$\{\neg b\}$

Exercise

Use resolution to prove that the following formula is valid:

$$\neg(a \wedge b \wedge (\neg a \vee \neg b))$$

Prove that its negation is unsatisfiable set of clauses:

$\{a\}$ $\{b\}$ $\{\neg a, \neg b\}$

$\{\neg b\}$

$\emptyset$

Unit Resolution

A *unit clause* is a clause that has precisely one literal; it's of the form $\{L\}$
Given a literal L, its complement $\overline{L}$ is defined by $\overline{x} = \neg x$, $\overline{\neg x} = x$.

Unit resolution is a special case of resolution where at least one of the clauses is a unit clause:

$$\frac{C \qquad \{L\}}{C \setminus \{\overline{L}\}}$$

Soundness: if L is true, then $\overline{L}$ is false, so it can be deleted from a disjunction C.

Subsumption: when applying resolution, if we obtain a clause $C' \subseteq C$ that is subset of a previosly derived one, we can delete C so we do not consider it any more. Any use of C can be replaced by use of C' with progress towards $\emptyset$ at least as good.

Unit resolution with $\{L\}$ can remove all occurences of L and $\overline{L}$ from our set.

Constructing a Conjunctive Normal Form

How would we transform this formula into a set of clauses:

$$\neg(((c \land a) \lor (\neg c \land b)) \leftrightarrow ((c \rightarrow b) \land (\neg c \rightarrow b)))$$

Which equivalences are guaranteed to produce a conjunctive normal form?

$$\begin{aligned}
\neg(F_1 \land F_2) &\quad\longleftrightarrow\quad (\neg F_1) \lor (\neg F_2) \\
F_1 \land (F_2 \lor F_3) &\quad\longleftrightarrow\quad (F_1 \land F_2) \lor (F_1 \lor F_3) \\
F_1 \lor (F_2 \land F_3) &\quad\longleftrightarrow\quad (F_1 \lor F_2) \land (F_1 \lor F_3)
\end{aligned}$$

Constructing a Conjunctive Normal Form

How would we transform this formula into a set of clauses:

$$\neg(((c \wedge a) \vee (\neg c \wedge b)) \longleftrightarrow ((c \rightarrow b) \wedge (\neg c \rightarrow b)))$$

Which equivalences are guaranteed to produce a conjunctive normal form?

$$\begin{array}{rcl} \neg(F_1 \wedge F_2) & \longleftrightarrow & (\neg F_1) \vee (\neg F_2) \\ F_1 \wedge (F_2 \vee F_3) & \longleftrightarrow & (F_1 \wedge F_2) \vee (F_1 \vee F_3) \\ F_1 \vee (F_2 \wedge F_3) & \longleftrightarrow & (F_1 \vee F_2) \wedge (F_1 \vee F_3) \end{array}$$

What is the complexity of such transformation in the general case?

Constructing a Conjunctive Normal Form

How would we transform this formula into a set of clauses:

$$\neg(((c \wedge a) \vee (\neg c \wedge b)) \longleftrightarrow ((c \rightarrow b) \wedge (\neg c \rightarrow b)))$$

Which equivalences are guaranteed to produce a conjunctive normal form?

$$\begin{array}{rcl} \neg(F_1 \wedge F_2) & \longleftrightarrow & (\neg F_1) \vee (\neg F_2) \\ F_1 \wedge (F_2 \vee F_3) & \longleftrightarrow & (F_1 \wedge F_2) \vee (F_1 \vee F_3) \\ F_1 \vee (F_2 \wedge F_3) & \longleftrightarrow & (F_1 \vee F_2) \wedge (F_1 \vee F_3) \end{array}$$

What is the complexity of such transformation in the general case?
Are there efficient algorithms for checking satisfiability of formulas in *disjunctive* normal form (disjunctions of conjunctions of literals)?
When checking satisfiability, is conversion into *conjunctive* normal form any better than disjunctive normal form?

Discussion of Normal Form Transformation

Transformation is exponential in general, applying from left to right equivalence

$$F_1 \vee (F_2 \wedge F_3) \longleftrightarrow (F_1 \vee F_2) \wedge (F_1 \vee F_3)$$

duplicates sub-formulas F_1, which may result in an exponentially larger formula.

If we were willing to do transformation using those rules, we might just as well transform formula into *disjunctive* normal form, because checking satisfiability of formula in disjunctive normal form is trivial, such formulas is a disjunction of conjunctions D_i and we have these equivalences:

$$\exists e. [\![D_1 \vee \ldots \vee D_n]\!]_e = 1$$
$$\exists e. ([\![D_1]\!]_e = 1 \vee \ldots \vee [\![D_n]\!]_e = 1)$$
$$(\exists e. [\![D_1]\!]_e = 1) \vee \ldots \vee (\exists e. [\![D_n]\!]_e = 1)$$

and the last condition is trivial to check, because we check satisfiability of conjunction D_i separately.

Equivalence and Equisatisfiability

Formulas F_1 and F_2 are **equivalent** iff: $F_1 \models F_2$ and $F_2 \models F_1$ $(\forall e. [\![F_1]\!]_e = [\![F_2]\!]_e)$

Formulas F_1 and F_2 are **equisatisfiable** iff: F_1 is satisfiable whenever F_2 is satisfiable.

Equivalent formulas are always equisatisfiable, but the converse is not the case in general. For example, formulas a and b are equisatisfiable, because they are both satisfiable.

Consider these two formulas:

- F_1: $(a \wedge b) \vee c$
- F_2: $(x \longleftrightarrow (a \wedge b)) \wedge (x \vee c)$

They are equisatisfiable but not equivalent. For example, given $e = \{(a,1),(b,1),(c,0),(x,0)\}$, $[\![F_1]\!]_e = 1$ whereas $[\![F_2]\!]_e = 0$. Interestingly, every choice of a, b, c that makes F_1 true can be extended to make F_2 true appropriately, if we choose x as $[\![a \wedge b]\!]_e$.

Flatenning as Satisfiability Preserving Transformation

Observation: Let F be a formula, G another formula, and $x \notin FV(F)$ a propositional variable. Let $F[G := x]$ denote the result of replacing an occurence of formula G inside F with x. Then F is equisatisfiable with

$$(x = G) \wedge F[G := x]$$

(Here, $=$ denotes $\longleftrightarrow$.)
Proof of equisatisfiability: a satisfying assignment for new formula is also a satisfying assignment for the old one. Conversely, since x does not occur in F, if $[\![F]\!]_e = 1$, we can change $e(x)$ to be defined as $[\![G]\!]_e$, which will make the new formula true.

(A transformation that produces an equivalent formula: *equivalence preserving*.)
A transformation that produces an equisatisfiable formula: *satisfiability preserving*.
Flattening is this satisfiability preserving transformation in any formalism that supports equality (here: equivalence): pick a subformula and given it a name by a fresh variable, applying the above observation.
Strategy: apply transformation from smallest non-variable subformulas.

Tseytin's Transformation (see also Calculus of Computation, Section 1.7.3)

Consider formula with $\neg, \wedge, \vee, \rightarrow, =, \oplus$

- Replace $F_1 \rightarrow F_2$ with $\neg F_1 \vee F_2$ and push negation into the propositional variables using De Morgan's laws and switching between $\oplus$ and $=$.
- Repeat: flatten an occurrence of a binary connective whose arguments are literals
- In the resulting conjunction, express each equivalence as a conjunction of clauses:

conjunct	corresponding clauses
$x = (a \wedge b)$	$\{\overline{x}, a\}, \{\overline{x}, b\}, \{\overline{a}, \overline{b}, x\}$
$x = (a \vee b)$	$\{\overline{x}, a, b\}, \{\overline{a}, x\}, \{\overline{b}, x\}$
$x = (a = b)$	
$x = (a \oplus b)$	

Exercise: Complete the missing entries. Are the rules in the last step equivalence preserving or only equisatisfiability preserving? Why is the resulting algorithm polynomial?

Example: Find an Equisatisfiable Set of Formulas in CNF

$$\{\boxed{c \wedge a} \vee (\neg c \wedge b)\}$$

Example: Find an Equisatisfiable Set of Formulas in CNF

$$\{\boxed{c \wedge a} \vee (\neg c \wedge b)\}$$

$$\{x_1 \vee \boxed{\neg c \wedge b},\ x_1 \longleftrightarrow (c \wedge a)\}$$

Example: Find an Equisatisfiable Set of Formulas in CNF

$$\{\boxed{c \land a} \lor (\neg c \land b)\}$$

$$\{x_1 \lor \boxed{\neg c \land b},\ x_1 \longleftrightarrow (c \land a)\}$$

$$\begin{aligned}\{x_1 \lor x_2,\ & x_2 \longleftrightarrow (\neg c \land b), \\ & x_1 \longleftrightarrow (c \land a)\}\end{aligned}$$

Example: Find an Equisatisfiable Set of Formulas in CNF

$$\{\boxed{c \wedge a} \vee (\neg c \wedge b)\}$$

$$\{x_1 \vee \boxed{\neg c \wedge b},\ x_1 \longleftrightarrow (c \wedge a)\}$$

$$\begin{aligned}\{x_1 \vee x_2,\ & x_2 \longleftrightarrow (\neg c \wedge b),\\ & x_1 \longleftrightarrow (c \wedge a)\}\end{aligned}$$

$$\begin{aligned}\{x_1 \vee x_2,\ & x_2 \rightarrow (\neg c \wedge b),\ (\neg c \wedge b) \rightarrow x_2,\\ & x_1 \rightarrow (c \wedge a),\ (c \wedge a) \rightarrow x_1\}\end{aligned}$$

$$\begin{aligned}\{x_1 \vee x_2,\ & \neg x_2 \vee \neg c,\ \neg x_2 \vee b,\ c \vee \neg b \vee x_2,\\ & \neg x_1 \vee c,\ \neg x_1 \vee a,\ \neg c \vee \neg a \vee x_1\}\end{aligned}$$

When representing clauses as sets:

$$\begin{aligned}\{\{x_1, x_2\},\ & \{\neg x_2, \neg c\},\ \{\neg x_2, b\},\ \{c, \neg b, x_2\},\\ & \{\neg x_1, c\},\ \{\neg x_1, a\},\ \{\neg c, \neg a, x_1\}\}\end{aligned}$$

SAT Solvers

A SAT solver takes as input a set of clauses.

To check satisfiability, convert to equisatisfiable set of clauses in polynomial time using Tseytin's transformation.

To check validity of a formula, take negation, check satisfiability, then negate the answer.

How should we check satisfiability of a set of clauses?

- resolution on clauses, favoring unit resolution and applying subsumption (complete)
  Davis and Putnam, 1960
- truth table method: check one value of a variable, then other (space efficient)

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm Sketch

```
def DPLL(S: Set[Clause]) : Bool =
  val S' = subsumption(UnitProp(S))
  if ∅ ∈ S' then false // unsat
  else if S' has only unit clauses then true // unit clauses give e
  else
    val L = a literal from a clause of S' where {L} ∉ S'
    DPLL(S' ∪ {{L}}) || DPLL(S' ∪ {{complement(L)}})

def UnitProp(S: Set[Clause]): Set[Clause] = // Unit Propagation (BCP)
  if C ∈ S, unit U ∈S, resolve(U,C) ∉ S
  then UnitProp((S - {C}) ∪ {resolve(U,C)}) else S

def subsumption(S: Set[Clause]): Set[Clause] =
  if C1,C2 ∈ S such that C1 ⊆ C2
  then subsumption(S - {C2}) else S
```

SAT Solvers: A Condensed History

- Deductive
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by "resolution"
- Backtrack Search
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment
- Conflict Driven Clause Learning [CDCL]
  - GRASP: Integrate a constraint learning procedure, 1996
- Locality Based Search
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation
- "Pre-processing"
  - Peephole optimization, e.g. miniSAT, 2005

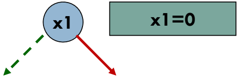
Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

J. P. Marques-Silva and Karem A. Sakallah, “GRASP: A Search Algorithm for Propositional Satisfiability”, *IEEE Trans. Computers*, C-48, 5:506-521, 1999.

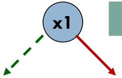
Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’



x1=0

Conflict Driven Learning and Non-chronological Backtracking



x1=0, x4=1

x1 + x4

x1 + x3' + x8'

x1 + x8 + x12

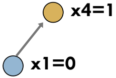
x2 + x11

x7' + x3' + x9

x7' + x8 + x9'

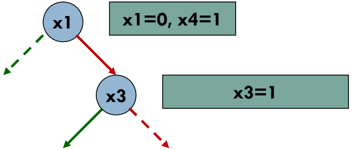
x7 + x8 + x10'

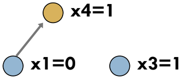
x7 + x10 + x12'



Conflict Driven Learning and Non-chronological Backtracking

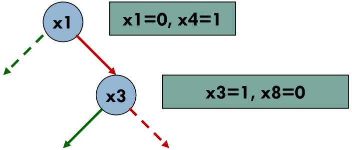
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

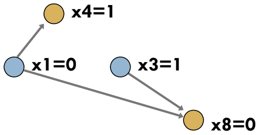




Conflict Driven Learning and Non-chronological Backtracking

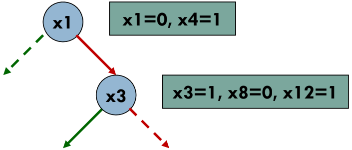
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

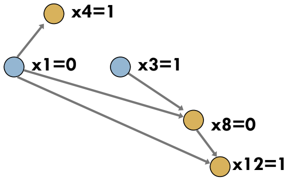




Conflict Driven Learning and Non-chronological Backtracking

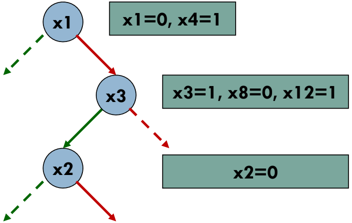
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

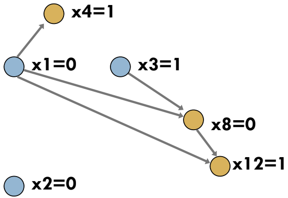




Conflict Driven Learning and Non-chronological Backtracking

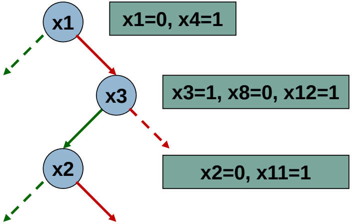
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

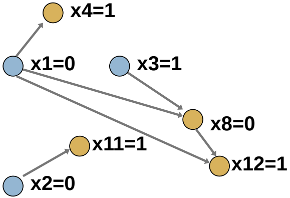




Conflict Driven Learning and Non-chronological Backtracking

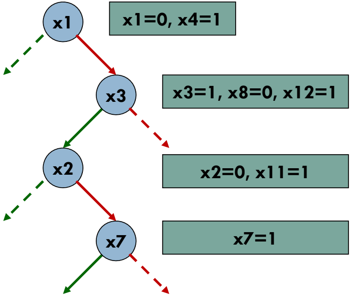
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

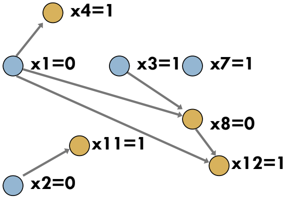




Conflict Driven Learning and Non-chronological Backtracking

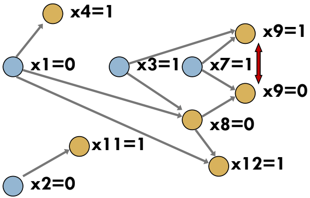
x1 + x4
x1 + x3’ + x8’
x1 + x8 + x12
x2 + x11
x7’ + x3’ + x9
x7’ + x8 + x9’
x7 + x8 + x10’
x7 + x10 + x12’

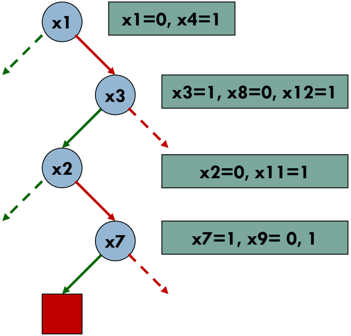




Conflict Driven Learning and Non-chronological Backtracking

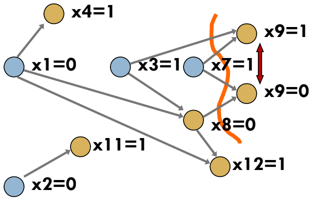
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

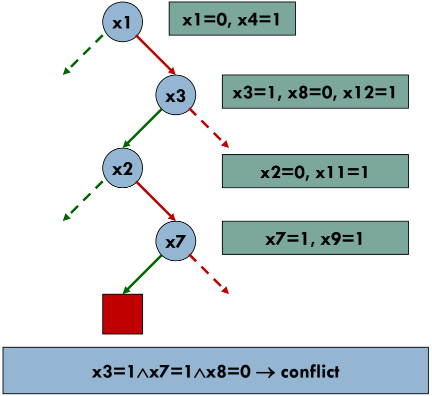




Conflict Driven Learning and Non-chronological Backtracking

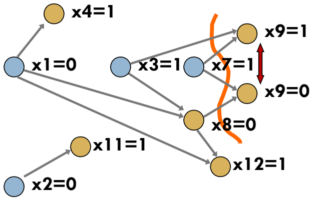
x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

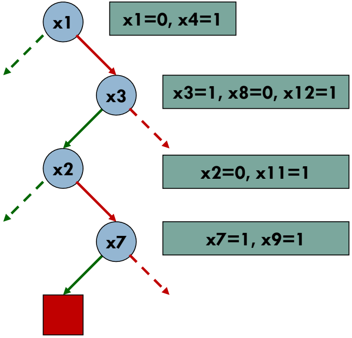




Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'



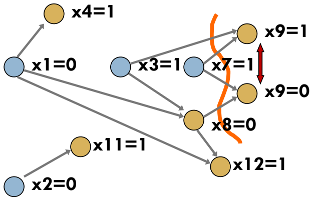


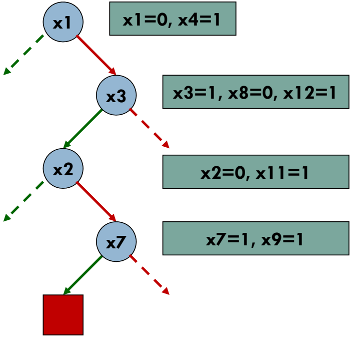
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict
Add conflict clause: x3'+x7'+x8

Conflict Driven Learning and Non-chronological Backtracking

- x1 + x4
- x1 + x3' + x8'
- x1 + x8 + x12
- x2 + x11
- x7' + x3' + x9
- x7' + x8 + x9'
- x7 + x8 + x10'
- x7 + x10 + x12'

(x7' + x3' + x9) and (x7' + x8 + x9') → x3'+x7'+x8



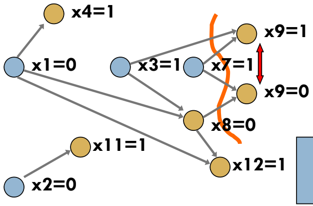


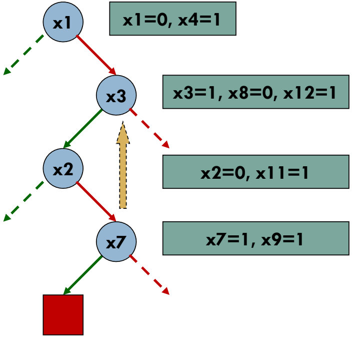
$x3=1 \wedge x7=1 \wedge x8=0 \rightarrow$ conflict
Add conflict clause: x3'+x7'+x8

Conflict Driven Learning and Non-chronological Backtracking

x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'

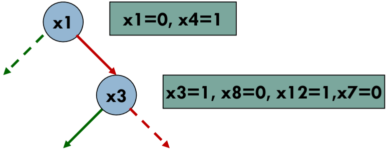
x3'+x7'+x8



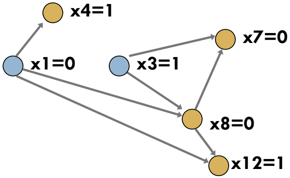


Backtrack to the decision level of x3=1

Conflict Driven Learning and Non-chronological Backtracking

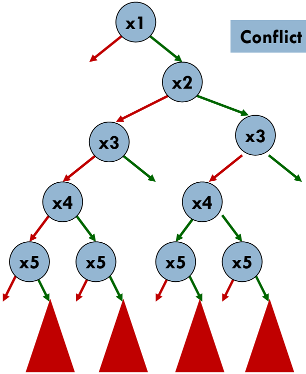


x1 + x4
x1 + x3' + x8'
x1 + x8 + x12
x2 + x11
x7' + x3' + x9
x7' + x8 + x9'
x7 + x8 + x10'
x7 + x10 + x12'
x3' + x7' + x8 ←new clause



Backtrack to the decision level of x3=1
Assign x7 = 0

What's the big deal?



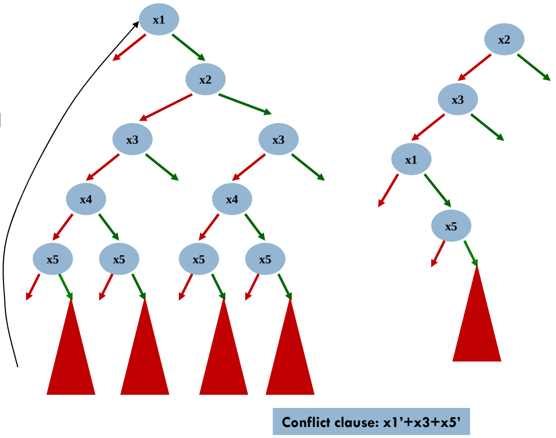
Conflict clause: x1'+x3+x5'

Significantly prune the search space – learned clause is useful forever!

Useful in generating future conflict clauses.

Restart

- Abandon the current search tree and reconstruct a new one
- The clauses learned prior to the restart are *still there* after the restart and can help pruning the search space
- Adds to robustness in the solver



SAT Solvers: A Condensed History

- Deductive
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”
- Backtrack Search
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment
- Conflict Driven Clause Learning [CDCL]
  - GRASP: Integrate a constraint learning procedure, 1996
- Locality Based Search
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation
- “Pre-processing”
  - Peephole optimization, e.g. miniSAT, 2005

Success with Chaff

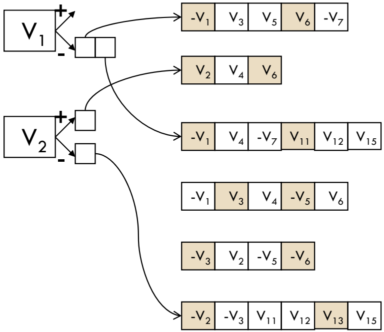
- First major instance: Tough (Industrial Processor Verification)
  - Bounded Model Checking, 14 cycle behavior
- Statistics
  - 1 million variables
  - 10 million literals initially
    - 200 million literals including added clauses
    - 30 million literals finally
  - 4 million clauses (initially)
    - 200K clauses added
  - 1.5 million decisions
  - 3 hour run time

M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc., 38th Design Automation Conference (DAC2001), June* 2001.

Chaff Contribution 1: Lazy Data Structures
2 Literal Watching for Unit-Propagation

- Avoid expensive book-keeping for unit-propagation
- N-literal clause can be unit or conflicting only after N-1 of the literals have been assigned to F
  - (v1 + v2 + v3): implied cases: (0 + 0 + v3) or (0 + v2 + 0) or (v1 + 0 + 0)
- Can completely ignore the first N-2 assignments to this clause
- Pick two literals in each clause to “watch” and thus can ignore any assignments to the other literals in the clause.
  - Example: (v1 + v2 + v3 + v4 + v5)
  - (**v1=X + v2=X** + v3=? {i.e. X or 0 or 1} + v4=? + v5=?)
- *Maintain the invariant*: If a clause can become newly implied via any sequence of assignments, then this sequence will include an assignment of one of the watched literals to F

2 Literal Watching



For every clause, two literals are watched

- When a variable is assigned true, only need to visit clauses where its watched literal is false (only one polarity)
  - Pointers from each literal to all clauses it is watched in
- In a *n* clause formula with *v* variables and *m* literals
  - Total number of pointers is *2n*
  - On average, visit *n*/*v* clauses per assignment
- *No updates to watched literals on backtrack*

Decision Heuristics – Conventional Wisdom

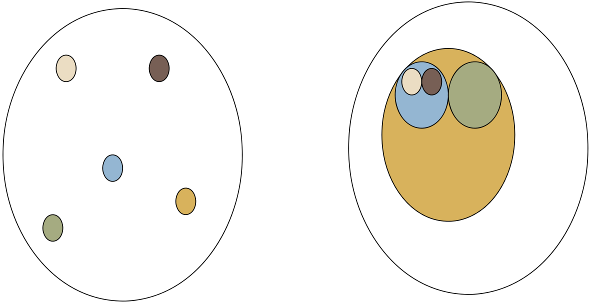
- "Assign most tightly constrained variable" : e.g. DLIS (Dynamic Largest Individual Sum)
  - Simple and intuitive: At each decision simply choose the assignment that satisfies the most unsatisfied clauses.
  - Expensive book-keeping operations required
    - Must touch *every* clause that contains a literal that has been set to true. Often restricted to initial (not learned) clauses.
    - Need to reverse the process for un-assignment.
- Look ahead algorithms even more compute intensive

  C. Li, Anbulagan, "Look-ahead versus look-back for satisfiability problems" *Proc. of CP*, 1997.
- Take a more "global" view of the problem

Chaff Contribution 2: Activity Based Decision Heuristics

- VSIDS: **V**ariable **S**tate **I**ndependent **D**ecaying **S**um
  - Rank variables by literal count in the initial clause database
  - Only increment counts as new (learnt) clauses are added
  - Periodically, divide all counts by a constant
- Quasi-static:
  - Static because it doesn't depend on variable state
  - Not static because it gradually changes as new clauses are added
    - Decay causes bias toward *recent* conflicts.
    - Has a beneficial interaction with 2-literal watching

Activity Based Heuristics and Locality Based Search



- By focusing on a sub-space, the covered spaces tend to coalesce
  - More opportunities for resolution since most of the variables are common.
  - Variable activity based heuristics lead to locality based search

SAT Solvers: A Condensed History

- Deductive
  - Davis-Putnam 1960 [DP]
  - Iterative existential quantification by “resolution”
- Backtrack Search
  - Davis, Logemann and Loveland 1962 [DLL]
  - Exhaustive search for satisfying assignment
- Conflict Driven Clause Learning [CDCL]
  - GRASP: Integrate a constraint learning procedure, 1996
- Locality Based Search
  - Emphasis on exhausting local sub-spaces, e.g. Chaff, Berkmin, miniSAT and others, 2001 onwards
  - Added focus on efficient implementation
- “Pre-processing”
  - Peephole optimization, e.g. miniSAT, 2005

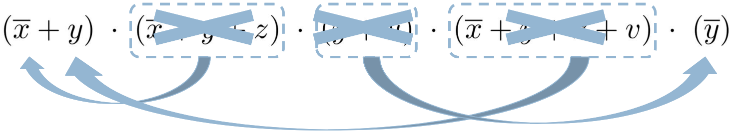
Pre-Processing of CNF Formulas

N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination, In *Proceedings of SAT 2005*

- Use structural information to simplify
  - Subsumption
  - Self-subsumption
  - Substitution

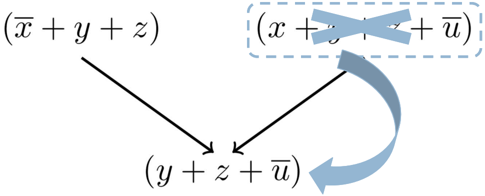
Pre-Processing: Subsumption

- Clause C_1 *subsumes* clause C_2 if C_1 *implies* C_2
- Subsumed clauses can be discarded



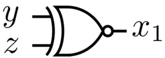
Pre-Processing: Self-Subsumption

- Subsumption *after* resolution step



Pre-Processing: Substitution

- Tseitin transformation introduces *definition* of variable



$$\underbrace{(x_1 \leftrightarrow (y \leftrightarrow z))}_{(\overline{x}_1+\overline{y}+z)\cdot(\overline{x}_1+\overline{z}+y)\cdot(\overline{y}+\overline{z}+x_1)\cdot(y+z+x_1)}$$

- Occurrence of x_1 can be eliminated by substitution
  - Corresponds to resolution with defining clauses

$$(x_1 + u) \quad \cdot \quad (\overline{x}_1 \qquad (y \leftrightarrow z) \quad \cdot \quad (\overline{x}_1 + \overline{z} + y)$$

$$(u + \overline{y} + z) \qquad (u + x_1) \quad \cdot \qquad (u + \overline{z} + y)$$

Concluding Remarks

- SAT: Significant shift from theoretical interest to practical impact.
- Quantum leaps between generations of SAT solvers
- Successful application of diverse CS techniques
  - Logic (Deduction and Solving), Search, Caching, Randomization, Data structures, efficient algorithms
  - Engineering developments through experimental computer science
- Presence of drivers results in maximum progress.
  - Electronic design automation – primary driver and main beneficiary
  - Software verification- the next frontier
- Opens attack on even harder problems
  - SMT, Max-SAT, QBF…

Sharad Malik and Lintao Zhang. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* 52, 8 (August 2009), 76-82.

References

- [GJ79] Michael R. Garey and David S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, W. H. Freeman and Company, San Francisco, 1979
- [T68] G. Tseitin, On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, Part 2 (1968)
- [DP 60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM, 7:201–215, 1960*
- [DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM, 5:394–397, 1962*
- [SS99] J. P. Marques-Silva and Karem A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Trans. Computers*, C-48, 5:506-521, 1999.
- [BS97] R. J. Bayardo Jr. and R. C. Schrag "Using CSP look-back techniques to solve real world SAT instances." *Proc. AAAI*, pp. 203-208, 1997
- [BS00] Luís Baptista and João Marques-Silva, "Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability," In *Principles and Practice of Constraint Programming – CP 2000*, 2000.

References

- [H07] J. Huang, "The effect of restarts on the efficiency of clause learning," Proceedings of the Twentieth International Joint Conference on Automated Reasoning, 2007
- [MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: Engineering and efficient sat solver. In *Proc., 38th Design Automation Conference (DAC2001), June* 2001.
- [ZS96] H. Zhang, M. Stickel, "An efficient algorithm for unit-propagation" In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics,1996
- [ES03] N. Een and N. Sorensson. An extensible SAT solver. In SAT-2003
- [B02] F. Bacchus "Exploring the Computational Tradeoff of more Reasoning and Less Searching", *Proc. 5th Int. Symp. Theory and Applications of Satisfiability Testing*, pp. 7-16, 2002.
- [GN02] E.Goldberg and Y.Novikov. BerkMin: a fast and robust SAT-solver. In *Proc., DATE-2002, pages 142–149, 2002.*

References

- [R04] L. Ryan, Efficient algorithms for clause-learning SAT solvers, M. Sc. Thesis, Simon Fraser University, 2002.
- [EB05] N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination, In *Proceedings of SAT 2005*
- [ZM03] L. Zhang and S. Malik, Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications, In Proceedings of Design Automation and Test in Europe, 2003.
- [LSB07] M. Lewis, T. Schubert, B. Becker, Multithreaded SAT Solving, In Proceedings of the 2007 Conference on Asia South Pacific Design Automation
- [HJS08] Youssef Hamadi, Said Jabbour, and Lakhdar Sais, ManySat: solver description, Microsoft Research-TR-2008-83
- [B86] R. E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers* , vol.C-35, no.8, pp.677-691, Aug. 1986
- [ZM09] Sharad Malik and Lintao Zhang. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* 52, 8 (August 2009), 76-82.