Completeness

A proof system is complete if it derives all formulas we want it to derive.
Completeness for classical propositional logic means that the system derives all propositional tautologies.
Is our example system complete with respect to formulas built from $\rightarrow$ and 0?

$$F \rightarrow (G \rightarrow F) \qquad ((F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H)) \qquad \frac{F \rightarrow G, F}{G}$$

In particular, if we are given an arbitrary valid propositional formula (propositional tautology), is there always a way to derive it using the axioms and the MP rule?

Completeness

A proof system is complete if it derives all formulas we want it to derive.
Completeness for classical propositional logic means that the system derives all propositional tautologies.
Is our example system complete with respect to formulas built from $\rightarrow$ and 0?

$$F \rightarrow (G \rightarrow F) \qquad ((F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H)) \qquad \frac{F \rightarrow G, F}{G}$$

In particular, if we are given an arbitrary valid propositional formula (propositional tautology), is there always a way to derive it using the axioms and the MP rule?

Does there exist a tautology that is not reachable from axioms using MP rule?

Completeness

A proof system is complete if it derives all formulas we want it to derive.
Completeness for classical propositional logic means that the system derives all propositional tautologies.
Is our example system complete with respect to formulas built from $\rightarrow$ and 0?

$$F \rightarrow (G \rightarrow F) \qquad ((F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H)) \qquad \frac{F \rightarrow G, F}{G}$$

In particular, if we are given an arbitrary valid propositional formula (propositional tautology), is there always a way to derive it using the axioms and the MP rule?

Does there exist a tautology that is not reachable from axioms using MP rule?

Is valid formula $0 \rightarrow a$ provable?

Do axioms or MP say anything about the 0 constant?

$$F \to (G \to F) \qquad ((F \to (G \to H)) \to ((F \to G) \to (F \to H)) \qquad \frac{F \to G, F}{G}$$

Do axioms or MP say anything about the 0 constant?

$$F \to (G \to F) \qquad ((F \to (G \to H)) \to ((F \to G) \to (F \to H)) \qquad \frac{F \to G, F}{G}$$

No! It is as if it was any other subformula.
Take proof

$$\begin{array}{l} \dots \\ a \to a \\ (a \to a) \to (0 \to (a \to a)) \quad (Axiom) \\ 0 \to (a \to a) \quad (MP) \end{array}$$

Replace 0 with e.g. $b \to c$ in the proof above. I will get a valid proof:

$$\begin{array}{l} \dots \\ a \to a \\ (a \to a) \to ((b \to c) \to (a \to a)) \quad (Axiom) \\ (b \to c) \to (a \to a) \quad (MP) \end{array}$$

Replacing 0 with some fixed formula gives us valid proofs.

Our proof system cannot prove $0 \rightarrow a$

Suppose it can. Then it can also prove a formula resulting by 0 with $a \rightarrow a$, so there would exist a proof of $(a \rightarrow a) \rightarrow a$.
We have shown before that we can derive $a \rightarrow a$, so we would also derive a using MP.
But a is not a tautology, which is a contradiction with the fact that the system is sound.

Our proof system cannot prove $0 \rightarrow a$

Suppose it can. Then it can also prove a formula resulting by 0 with $a \rightarrow a$, so there would exist a proof of $(a \rightarrow a) \rightarrow a$.
We have shown before that we can derive $a \rightarrow a$, so we would also derive a using MP.
But a is not a tautology, which is a contradiction with the fact that the system is sound.

We could add a few more axioms and get complete systems that were introduced by e.g. Frege and Hilbert. Instead, we will look at resolution as a proof system.

Towards Propositional Resolution

A Proof System with Decision and Simplification

Consider propositional formulas with $\wedge, \vee, \neg$.

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

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

Proof of soundness. To show $\{F,G\} \models (F[x:=0] \vee G[x:=1])$, consider an environment e and assume $[\![F]\!]_e = 1$ and $[\![G]\!]_e = 1$.

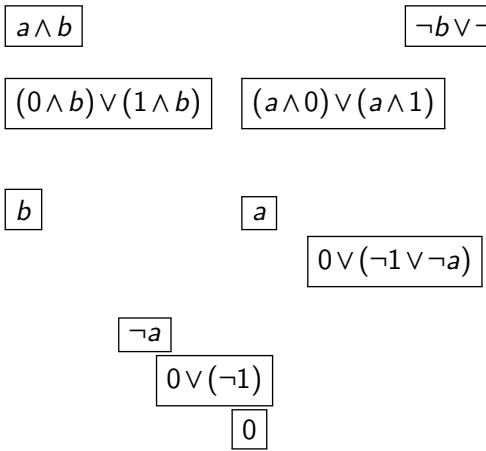
- If $e(x) = 0$, then $[\![F[x:=0]]\!]_e = [\![F]\!]_e = 1$, so the first disjunct is 1
- If $e(x) = 1$, then $[\![G[x:=1]]\!]_e = [\![G]\!]_e = 1$, so the second disjunct is 1.

In both cases, the disjunction evaluates to 1 in e.

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

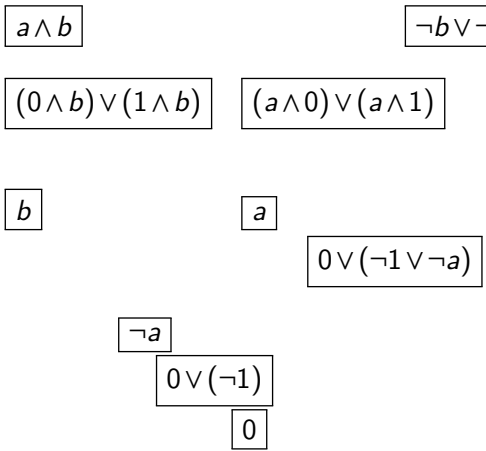
Example Derivation

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



Example Derivation

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



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:

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

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 $\exists x_1 \ldots \exists x_n.(F_1 \wedge F_2)$.
When A is unsatisfiable, Z is equivalent to 0, and has no free variables. By simplification rules, we can derive 0.

Resolution on Clauses

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

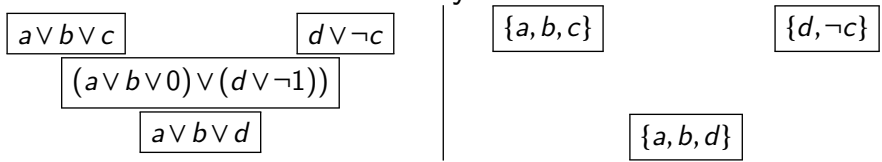
If C is a clause then $[\![C]\!]_e = 1$ iff there exists a literal $L \in C$ such that $[\![L]\!]_e = 1$.
We represent 0 using the empty clause $\emptyset$.
As for any formulas, a finite set of clauses A can be interpreted as a conjunction.
Thus, a set of clauses can be viewed as a formula in conjunctive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



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 unsatisfiable 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 $F_1 \to F_2$ and $F_2 \to F_3$ are true, so is $F_1 \to F_3$.
Thus, $\to$ denotes a transitive relation on $\{0, 1\}$.

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

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

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 $\overline{x} = \neg x, \quad \overline{\neg x} = x$.

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.

If we derive $\{L\}$ we can remove all other occurrences of L and $\overline{L}$: if $L \in C$ then C is subsumed by $\{L\}$ and if $\overline{L} \in C$ then C is subsumed by $C \setminus \{\overline{L}\}$.

From Formulas to Clauses

Constructing a Conjunctive Normal Form

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

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

Which equivalences are guaranteed to produce a conjunctive normal form?

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

Constructing a Conjunctive Normal Form

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

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

Which equivalences are guaranteed to produce a conjunctive normal form?

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

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:

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

Which equivalences are guaranteed to produce a conjunctive normal form?

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

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

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

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 formula 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 $\leftrightarrow$.)
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

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

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

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

Example: Find an Equisatisfiable Set of Formulas in CNF

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

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

$$\{x_1 \vee x_2,\ x_2 \leftrightarrow (\neg c \wedge b),\ x_1 \leftrightarrow (c \wedge a)\}$$

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

$$\{x_1 \vee x_2,\ \neg x_2 \vee \neg c,\ \neg x_2 \vee b,\ c \vee \neg b \vee x_2,\ \neg x_1 \vee c,\ \neg x_1 \vee a,\ \neg c \vee \neg a \vee x_1\}$$

When representing clauses as sets:

$$\{\{x_1, x_2\},\ \{\neg x_2, \neg c\},\ \{\neg x_2, b\},\ \{c, \neg b, x_2\},\ \{\neg x_1, c\},\ \{\neg x_1, a\},\ \{\neg c, \neg a, x_1\}\}$$

Finding Proofs

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 S1 = subsumption(UnitProp(S))
  if $\emptyset \in$ S1 **then false** // unsat
  else if S1 has only unit clauses **then true** // S1 gives satisfying assignment
  else
    // instead of doing general resolution, make more unit literals, recurse
    val L = a literal from a clause of S1 where $\{L\} \notin$ S1
    DPLL(S1 $\cup$ {{L}}) || DPLL(S1 $\cup$ {{$\bar{L}$}})

def UnitProp(S: Set[Clause]): Set[Clause] = // Unit Propagation (BCP)
  if C $\in$ S, unit U $\in S$, $\bar{U} \in C$, C $- \{\bar{U}\} \notin$ S
  then UnitProp((S $- \{C\}) \cup \{C - \{\bar{U}\}\}$) **else** S

def subsumption(S: Set[Clause]): Set[Clause] =
  if C1,C2 $\in$ S such that C1 $\subseteq$ C2
  then subsumption(S $- \{C2\}$) **else** S

Data Structures in a SAT Solver

Previous algorithm

- generates new clauses in UnitProp
- deletes clauses in UnitProp and subsumption

This is very inefficient. SAT solvers use more efficient data structures:

- all unit clauses are represented as a current assignment: a *partial map*, environment e from some of the variables to truth values (starts as empty map)
  - unit clause $\{\neg a\}$ becomes $e(a) = 0$, unit clause $\{a\}$ becomes $e(a) = 1$
- whenever a new literal L becomes true, we check if e assigns its value in the contradictory way and, if so, we detect a *conflict*, corresponding to $\emptyset$
- instead of resolving $\{L_1, L_2, \ldots, L_n\}$ with a unit literal $\{\overline{L_1}\}$: interpret each clause in the context of current e: once $[\![L_1]\!]_e = 0$, we interpret clause as $\{L_2, \ldots, L_n\}$
- when all literals in a clause are 0 except for one literal, say, a variable p, then p must be true, so update assignment $e' = e \cup \{(a, 1)\}$, but report contradiction if already $e(a) = 0$. *Two-watched literal* data structure makes detecting this fast.
- instead of subsumption: mark and ignore clauses that are true in current e

Clause Learning. Generating Proofs from SAT Solver Runs

CDCL (conflict-driven clause learning): the solver maintains the progress in exploring the space using learned clauses. Each learned clause is derived by resolution from existing ones. For example, if decisions that set literals L_1, L_2, L_3 to true lead to a conflict, then the clause $\neg(L_1 \wedge L_2 \wedge L_3)$ is a resolution consequence of the input formula, so we can add it to a clause set.

Given the length of proofs and subtlety of SAT solvers, there exist very compact formats (e.g. DRAT) that can be checked independenty using simple and efficient proof checkers, which operate in polynomial time.

There exists combinatorial statements (e.g. Pigenhole principle) that generate an infinite family of unsat formulas $F_1, F_2, \ldots$ such that the shortest *resolution* proof $F_i \vdash \emptyset$ is exponential in the size of F_i (Urquhart 1987). No such hard cases are known for *extended* resolution, which can introduce fresh propositional variables that name given clauses (Audemard, Katsirelos, Simon, AAAI-10). An efficiently checkable proof system with short proofs would imply NP = coNP.