Sat Solvers: Introduction to SAT-solving

Barnaby Martin
barnaby.d.martin@durham.ac.uk

Some Motivation

Some Motivation

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

Some Motivation

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

We have requirements on how the system should function, for example safety, reliability, security, availability, absence of deadlocks etc.

Some Motivation

Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

We have requirements on how the system should function, for example safety, reliability, security, availability, absence of deadlocks etc.

How can one ensure that the system satisfies these requirements?

Some Motivation

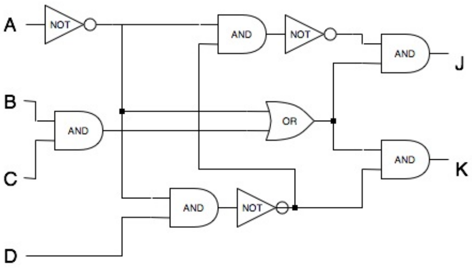
Suppose we design a (complex) system, which may contain various components, for example, sensors, networks, computers. All of these components are using software.

We have requirements on how the system should function, for example safety, reliability, security, availability, absence of deadlocks etc.

How can one ensure that the system satisfies these requirements?

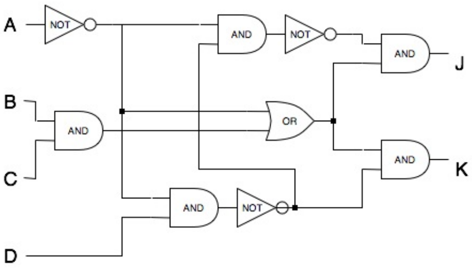
Modern computer systems are unreliable.

Example



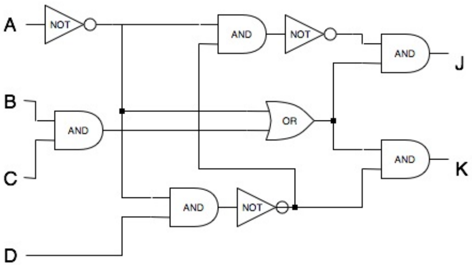
- We used a circuit C1 in a processor and consider replacing it by another circuit C2. For example, we may believe that the use of C2 results in a lower energy consumption.

Example



- We used a circuit C1 in a processor and consider replacing it by another circuit C2. For example, we may believe that the use of C2 results in a lower energy consumption.
- We want to be sure that C2 is correct, that is, it will behave according to some specification.

Example



- We used a circuit C1 in a processor and consider replacing it by another circuit C2. For example, we may believe that the use of C2 results in a lower energy consumption.
- We want to be sure that C2 is correct, that is, it will behave according to some specification.
- If we know that C1 is correct, it is sufficient to prove that C2 is functionally equivalent to C1.

How to establish correctness

How to establish correctness

- Consider the system as a mathematical object. To do this, we will have to build a formal model of the system;

How to establish correctness

- Consider the system as a mathematical object. To do this, we will have to build a formal model of the system;
- Find a formal language for expressing intended properties;
- The language must have a semantics that explains which models satisfy properties expressible in the language.

How to establish correctness

- Consider the system as a mathematical object. To do this, we will have to build a formal model of the system;
- Find a formal language for expressing intended properties;
- The language must have a semantics that explains which models satisfy properties expressible in the language.
- Write a specification, that is, intended properties of the system in this language.

How to establish correctness

- Consider the system as a mathematical object. To do this, we will have to build a formal model of the system;
- Find a formal language for expressing intended properties;
- The language must have a semantics that explains which models satisfy properties expressible in the language.
- Write a specification, that is, intended properties of the system in this language.
- Prove formally that the system model is also a model of the specification.

How to establish correctness

- Consider the system as a mathematical object. To do this, we will have to build a formal model of the system;
- Find a formal language for expressing intended properties;
- The language must have a semantics that explains which models satisfy properties expressible in the language.
- Write a specification, that is, intended properties of the system in this language.
- Prove formally that the system model is also a model of the specification.

- A natural tool for this whole process is logic.

Automated Reasoning

- Formal proofs proceed via reasoning
  - *Reasoning = deriving conclusions from facts.*
- This course is about automation of (i.e. algorithms for) reasoning.

Applications of Automated Reasoning

- SAT solvers
- Software and hardware verification
  - 2007 Turing Award to Clarke, Emerson and Sifakis for developing AR-based model checking technique, now widely adopted in industry

Applications of Automated Reasoning

- SAT solvers
- Software and hardware verification
  - 2007 Turing Award to Clarke, Emerson and Sifakis for developing AR-based model checking technique, now widely adopted in industry
- Artificial Intelligence, including
  - planning
  - constraint satisfaction
- Circuit design

Applications of Automated Reasoning

- SAT solvers
- Software and hardware verification
  - 2007 Turing Award to Clarke, Emerson and Sifakis for developing AR-based model checking technique, now widely adopted in industry
- Artificial Intelligence, including
  - planning
  - constraint satisfaction
- Circuit design
- Cryptography
- Databases
- Theorem proving in mathematics

Different Logics for Different Applications

- classical logic (textbook style logic, $a \vee \overline{a}$ always holds)
- temporal logic (truth changes over time)
- modal logic (handles ‘possibly’, ‘necessarily’ and ‘always’ true statements)

Different Logics for Different Applications

- classical logic (textbook style logic, $a \vee \overline{a}$ always holds)
- temporal logic (truth changes over time)
- modal logic (handles ‘possibly’, ‘necessarily’ and ‘always’ true statements)
- many-valued logics (ideas like ‘true’, ‘false’, and ‘don’t know’, or intermediate values between true/false)
- fuzzy logic (the standard set of truth values is [0,1], can speak about the degree of membership of x in a set A)
- non-monotonic logic (more facts may yield fewer conclusions)

Different Logics for Different Applications

- classical logic (textbook style logic, $a \vee \overline{a}$ always holds)
- temporal logic (truth changes over time)
- modal logic (handles 'possibly', 'necessarily' and 'always' true statements)
- many-valued logics (ideas like 'true', 'false', and 'don't know', or intermediate values between true/false)
- fuzzy logic (the standard set of truth values is [0,1], can speak about the degree of membership of x in a set A)
- non-monotonic logic (more facts may yield fewer conclusions)

In the course we will mainly consider *propositional logic*.

Reminder: Propositional logic

- Propositional variables: $x, y, z, \ldots$ take values $0/1$ (or `false`/`true`)
- Connectives: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\oplus$

Reminder: Propositional logic

- Propositional variables: $x, y, z, \ldots$ take values $0/1$ (or `false`/`true`)
- Connectives: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\oplus$
- Formulas are built recursively:
  - All variables are formulas
  - If Φ and Ψ are formulas then $\Phi \vee \Psi$ is a formula
  - etc

Reminder: Propositional logic

- Propositional variables: $x, y, z, \ldots$ take values $0/1$ (or `false`/`true`)
- Connectives: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\oplus$
- Formulas are built recursively:
  - All variables are formulas
  - If Φ and Ψ are formulas then $\Phi \vee \Psi$ is a formula
  - etc
- Truth assignment (TA) = assignment of values to variables in a formula
- Obvious way to evaluate a formula on a given TA

Reminder: Propositional logic

- Propositional variables: $x, y, z, \ldots$ take values $0/1$ (or `false`/`true`)
- Connectives: $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$, $\oplus$
- Formulas are built recursively:
  - All variables are formulas
  - If Φ and Ψ are formulas then $\Phi \vee \Psi$ is a formula
  - etc
- Truth assignment (TA) = assignment of values to variables in a formula
- Obvious way to evaluate a formula on a given TA
- TA t satisfies a formula Φ if Φ evaluates to 1 on t.
- Φ is satisfying if some TA satisfies it.

Satisfaction vs. Consequence

- Let Ψ and Φ be two propositional formulas. Then Φ is a logical consequence of Ψ, or Ψ entails Φ (denoted $\Psi \models \Phi$) if every TA that satisfies Ψ also satisfies Φ.
  - Trivial case: any formula is a consequence of an unsatisfiable formula.

Satisfaction vs. Consequence

- Let Ψ and Φ be two propositional formulas. Then Φ is a logical consequence of Ψ, or Ψ entails Φ (denoted $\Psi \models \Phi$) if every TA that satisfies Ψ also satisfies Φ.
  - Trivial case: any formula is a consequence of an unsatisfiable formula.
- Fact: For any two propositional formulas Ψ and Φ, we have $\Psi \models \Phi$ if and only if the formula $\Psi \wedge \neg\Phi$ is unsatisfiable.

Satisfaction vs. Consequence

- Let Ψ and Φ be two propositional formulas. Then Φ is a logical consequence of Ψ, or Ψ entails Φ (denoted $\Psi \models \Phi$) if every TA that satisfies Ψ also satisfies Φ.
  - Trivial case: any formula is a consequence of an unsatisfiable formula.
- Fact: For any two propositional formulas Ψ and Φ, we have $\Psi \models \Phi$ if and only if the formula $\Psi \wedge \neg\Phi$ is unsatisfiable.
  - Hint: if a TA satisfies Ψ then it also satisfies Φ and so can't satisfy $\neg\Phi$.

Satisfaction vs. Consequence

- Let Ψ and Φ be two propositional formulas. Then Φ is a logical consequence of Ψ, or Ψ entails Φ (denoted $\Psi \models \Phi$) if every TA that satisfies Ψ also satisfies Φ.
  - Trivial case: any formula is a consequence of an unsatisfiable formula.
- Fact: For any two propositional formulas Ψ and Φ, we have $\Psi \models \Phi$ if and only if the formula $\Psi \wedge \neg\Phi$ is unsatisfiable.
  - Hint: if a TA satisfies Ψ then it also satisfies Φ and so can't satisfy $\neg\Phi$.
- Similarly, $\Psi_1, \ldots, \Psi_n \models \Phi$ if and only if $\Psi_1 \wedge \ldots \wedge \Psi_n \wedge \neg\Phi$ is unsatisfiable

Satisfaction vs. Consequence

- Let Ψ and Φ be two propositional formulas. Then Φ is a logical consequence of Ψ, or Ψ entails Φ (denoted $\Psi \models \Phi$) if every TA that satisfies Ψ also satisfies Φ.
  - Trivial case: any formula is a consequence of an unsatisfiable formula.
- Fact: For any two propositional formulas Ψ and Φ, we have $\Psi \models \Phi$ if and only if the formula $\Psi \wedge \neg\Phi$ is unsatisfiable.
  - Hint: if a TA satisfies Ψ then it also satisfies Φ and so can't satisfy $\neg\Phi$.
- Similarly, $\Psi_1, \ldots, \Psi_n \models \Phi$ if and only if $\Psi_1 \wedge \ldots \wedge \Psi_n \wedge \neg\Phi$ is unsatisfiable
- Thus, knowing how to check for entailment is the same as knowing how to check for (un)satisfiability.

Clauses

- Literals: variables $x, y, z, \ldots$ and their negations $\overline{x}, \overline{y}, \overline{z}, \ldots$
- Clauses: formulas of the form $\ell_1 \vee \ldots \vee \ell_n$ where each ℓ_i is a literal
  - — can be thought of as a set $\{\ell_1, \ldots, \ell_n\}$.

Clauses

- Literals: variables $x, y, z, \ldots$ and their negations $\overline{x}, \overline{y}, \overline{z}, \ldots$
- Clauses: formulas of the form $\ell_1 \vee \ldots \vee \ell_n$ where each ℓ_i is a literal
  - can be thought of as a set $\{\ell_1, \ldots, \ell_n\}$.
- Conjunctive normal form: $C_1 \wedge \ldots \wedge C_m$ where each C_i is a clause
- Fact: each formula is equivalent to one in CNF.
- A clause-set is a set of clauses
  - Each CNF can be thought of as a clause-set $\{C_1, \ldots, C_m\}$

Clauses

- Literals: variables $x, y, z, \ldots$ and their negations $\overline{x}, \overline{y}, \overline{z}, \ldots$
- Clauses: formulas of the form $\ell_1 \vee \ldots \vee \ell_n$ where each ℓ_i is a literal
  - can be thought of as a set $\{\ell_1, \ldots, \ell_n\}$.
- Conjunctive normal form: $C_1 \wedge \ldots \wedge C_m$ where each C_i is a clause
- Fact: each formula is equivalent to one in CNF.
- A clause-set is a set of clauses
  - Each CNF can be thought of as a clause-set $\{C_1, \ldots, C_m\}$
- Enough to know how to check satisfiability of clause-sets

Example

- $F = \{\{u, \overline{v}, \overline{y}\}, \{\overline{u}, z\}, \{\overline{v}, \overline{w}\}, \{w, \overline{x}\}, \{x, y, \overline{z}\}\}$
- $var(F) = \{u, v, w, x, y, z\}$
- Since F is a *set*, we have $F = \{\{\overline{u}, z\}, \{\overline{v}, \overline{w}\}, \{u, \overline{v}, \overline{y}\}, \{w, \overline{x}\}, \{\overline{u}, z\}, \{x, y, \overline{z}\}\}$.

The DIMACS format

DIMACS stands for *Center for Discrete Mathematics and Theoretical Computer Science, Rutgers, New Jersey*.

The DIMACS format

DIMACS stands for *Center for Discrete Mathematics and Theoretical Computer Science, Rutgers, New Jersey*. A simple

text format for representing clause-sets was introduced for a SAT Competition 1993.
Standard format for most of today's SAT solvers.

The DIMACS format

DIMACS stands for *Center for Discrete Mathematics and Theoretical Computer Science, Rutgers, New Jersey*. A simple

text format for representing clause-sets was introduced for a SAT Competition 1993.
Standard format for most of today's SAT solvers.

- Variables are indicated as positive integers, negative literals by negative integers.
- Any line starting with a `c` is a comment.
- At the top of the file is a line of form
  `p cnf N M`
  where N is the number of the largest variable, and M is the total number of clauses in the clause-set.
- Clauses are separated by a zero ("0").

Example

Clause-set $F = \{\{u, \overline{v}, \overline{y}\}, \{\overline{u}, z\}, \{\overline{v}, \overline{w}\}, \{w, \overline{x}\}, \{x, y, \overline{z}\}\}$ written in DIMACS format looks like this:

```
c This is an example
c NOTE: Satisfiable
c
p cnf 6 5
1 -2 -5 0
-1 6 0
-2 -3 0
3 -4 0
-4 5 -6 0
```

Example

Clause-set $F = \{\{u, \overline{v}, \overline{y}\}, \{\overline{u}, z\}, \{\overline{v}, \overline{w}\}, \{w, \overline{x}\}, \{x, y, \overline{z}\}\}$ written in DIMACS format looks like this:

```
c This is an example
c NOTE: Satisfiable
c
p cnf 6 5
1 -2 -5 0
-1 6 0
-2 -3 0
3 -4 0
-4 5 -6 0
```

Try solving F by this online SAT solver

Satisfiability of formulas vs. clause-sets

- Want to concentrate on algorithms for clause-sets
- What about general propositional formulas?

Satisfiability of formulas vs. clause-sets

- Want to concentrate on algorithms for clause-sets
- What about general propositional formulas?
- Can always convert them to CNF.

Satisfiability of formulas vs. clause-sets

- Want to concentrate on algorithms for clause-sets
- What about general propositional formulas?
- Can always convert them to CNF. But at what cost?
- Standard procedure for converting a formula Φ to CNF (clause-set) $F_{\mathrm{CNF}(\Phi)}$:
  1. Express all connectives using $\vee$, $\wedge$, and $\neg$
  2. Push $\neg$ inside
  3. Use De Morgan's Laws to convert to CNF
     See slides at the end for more details and examples

Satisfiability of formulas vs. clause-sets

- Want to concentrate on algorithms for clause-sets
- What about general propositional formulas?
- Can always convert them to CNF. But at what cost?
- Standard procedure for converting a formula Φ to CNF (clause-set) $F_{\text{CNF}(\Phi)}$:
  1. Express all connectives using $\vee$, $\wedge$, and $\neg$
  2. Push $\neg$ inside
  3. Use De Morgan's Laws to convert to CNF
     See slides at the end for more details and examples
- For $n > 0$ let Ψ_n be the formula

$$(x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n).$$

- It turns out that $F_{\text{CNF}(\Psi_n)}$ is rather huge!

Converting Ψ_n

Converting Ψ_n

- CNF(Ψ_n) contains all 2^n possible terms $(z_1 \vee \cdots \vee z_n)$ for $z_i \in \{x_i, y_i\}$.
- Thus $F_{\mathrm{CNF}(\Psi_n)}$ contains 2^n clauses.

Converting Ψ_n

- CNF(Ψ_n) contains all 2^n possible terms $(z_1 \vee \cdots \vee z_n)$ for $z_i \in \{x_i, y_i\}$.
- Thus $F_{\mathrm{CNF}(\Psi_n)}$ contains 2^n clauses.
- It is not difficult to show that one cannot do it with fewer clauses.

Fact

For every $n > 1$ there exists a propositional formula of length $2n$ for which any logical equivalent clause-set contains at least 2^n clauses.

Fast Transformation

Fast Transformation

A faster transformation was proposed by Tseitin in 1968.

- **Idea**: Transformation needs not preserve logical equivalence, equivalence w.r.t. satisfiability is enough.

Definition (equisatisfiability)

Two propositional formulas or clause-sets are equisatisfiable if either both are satisfiable or both are unsatisfiable.

- We may introduce new variables (“extension variables”)

Tseitin's Algorithm

Tseitin's Algorithm

- **Input**: a propositional formula Ψ; assume that double negations are eliminated. Initially, put $F = \emptyset$.
- While Ψ has a subformula $\ell \circ m$ where $\circ$ is any binary logical connective and ℓ, m are literals
  - replace in Ψ the subformula $\ell \circ m$ with a new variable $x_{\ell \circ m}$,
  - extend the clause set: $F \cup F_{\ell \circ m}$ and use this as a new F
    $F_{\ell \circ m}$ is a clause-set logically equivalent to the propositional formula $x_{\ell \circ m} \leftrightarrow (\ell \circ m)$.
- Now Ψ is a single literal ℓ. Add the unit clause $\{\ell\}$ to F and halt.
- **Output**: clause-set F.
- Note we can stop as soon as Ψ is in CNF and add the clauses of F_Ψ to F.

Defining clause-set $F_{\ell \wedge m}$

Defining clause-set $F_{\ell \wedge m}$

- The clause-set $F_{\ell \wedge m}$ needs to be logically equivalent to the formula $x \equiv \ell \wedge m$.
- We derive $F_{\ell \wedge m}$ using the "slow" method:
  - $x \equiv \ell \wedge m$
  - $[x \to (\ell \wedge m)] \wedge [(\ell \wedge m) \to x]$
  - $[\neg x \vee (\ell \wedge m)] \wedge [\neg(\ell \wedge m) \vee x]$
  - $[(\neg x \vee \ell) \wedge (\neg x \vee m)] \wedge [(\neg \ell \vee \neg m) \vee x]$
  - $(\neg x \vee \ell) \wedge (\neg x \vee m) \wedge (\neg \ell \vee \neg m \vee x)$
- Hence $F_{\ell \wedge m} = \{\{\overline{x}, \ell\}, \{\overline{x}, m\}, \{\overline{\ell}, \overline{m}, x\}\}$.

List of defining clause-sets

List of defining clause-sets

$$
\begin{aligned}
F_{\ell\wedge m} &= \{\{\overline{\ell}, \overline{m}, x_{\ell\wedge m}\}, \{\ell, \overline{x_{\ell\wedge m}}\}, \{m, \overline{x_{\ell\wedge m}}\}\} \\
F_{\ell\vee m} &= \{\{\overline{\ell}, x_{\ell\vee m}\}, \{\overline{m}, x_{\ell\vee m}\}, \{\ell, m, \overline{x_{\ell\vee m}}\}\} \\
F_{\ell\to m} &= \{\{\ell, x_{\ell\to m}\}, \{\overline{m}, x_{\ell\to m}\}, \{\overline{\ell}, m, \overline{x_{\ell\to m}}\}\} \\
F_{\ell\leftrightarrow m} &= \{\{\ell, m, x_{\ell\leftrightarrow m}\}, \{\overline{\ell}, \overline{m}, x_{\ell\leftrightarrow m}\}, \\
&\quad \{\overline{\ell}, m, \overline{x_{\ell\leftrightarrow m}}\}, \{\ell, \overline{m}, \overline{x_{\ell\leftrightarrow m}}\}\} \\
F_{\ell\oplus m} &= \{\{\overline{\ell}, m, x_{\ell\oplus m}\}, \{\ell, \overline{m}, x_{\ell\oplus m}\}, \\
&\quad \{\ell, m, \overline{x_{\ell\oplus m}}\}, \{\overline{\ell}, \overline{m}, \overline{x_{\ell\oplus m}}\}\}
\end{aligned}
$$

Example 1

Example 1

- Input: $\Psi = \neg(x \rightarrow (y \rightarrow x))$.

Example 1

- Input: $\Psi = \neg(x \to (y \to x))$.
- $\Psi = \neg(x \to (y \to x))$, $F = \emptyset$.
- $\Psi = \neg(x \to u)$, $F = \{\{y, u\}, \{\overline{x}, u\}, \{\overline{y}, x, \overline{u}\}\}$.
- $\Psi = \neg v$,
  $F = \{\{y, u\}, \{\overline{x}, u\}, \{\overline{y}, x, \overline{u}\}, \{x, v\}, \{\overline{u}, v\}, \{\overline{x}, u, \overline{v}\}\}$.
- Output: $F = \{\{y, u\}, \{\overline{x}, u\}, \{\overline{y}, x, \overline{u}\}, \{x, v\}, \{\overline{u}, v\}, \{\overline{x}, u, \overline{v}\}, \{\overline{v}\}\}$.

Example 2

Example 2

- $\Psi_n = (x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n)$, $F = \emptyset$

Example 2

- $\Psi_n = (x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n)$, $F = \emptyset$
- $x_{x_1 \wedge y_1} \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$, $F = F_{x_1 \wedge y_1}$.
- $x_{x_1 \wedge y_1} \vee x_{x_2 \wedge y_2} \vee (x_3 \wedge y_3) \vee \cdots \vee (x_n \wedge y_n)$,
  $F = F_{x_1 \wedge y_1} \cup F_{x_2 \wedge y_2}$.
- etc.
- $x_{x_1 \wedge y_1} \vee x_{x_2 \wedge y_2} \vee \cdots \vee x_{x_n \wedge y_n}$, $F = F_{x_1 \wedge y_1} \cup \cdots \cup F_{x_n \wedge y_n}$.
- $F = \{\{x_{x_1 \wedge y_1}, \ldots, x_{x_n \wedge y_n}\}\} \cup F_{x_1 \wedge y_1} \cup \cdots \cup F_{x_n \wedge y_n}$.

Results

Results

Let Ψ be a propositional formula and $T(\Psi)$ the clause-set obtained using the fast transformation.

- Ψ and $T(\Psi)$ are equisatisfiable.
- Ψ is a tautology if and only if $T(\neg\Psi)$ is unsatisfiable.

Results

Let Ψ be a propositional formula and $T(\Psi)$ the clause-set obtained using the fast transformation.

- Ψ and $T(\Psi)$ are equisatisfiable.
- Ψ is a tautology if and only if $T(\neg\Psi)$ is unsatisfiable.
- Let n be the number of occurrences of binary connectives in Ψ. Then $T(\Psi)$ contains at most $4n + 1$ clauses, each clause contains at most 3 literals. Thus $|T(\Psi)| = O(n)$, i.e., the number of clauses in $T(\Psi)$ is linear in the number n of occurrences of binary connectives in Ψ.
- Note, however, that Ψ and $T(\Psi)$ are equisatisfiable but not logically equivalent. (What does this mean?)

A search algorithm

A search algorithm

- Algorithm A(F)
- outputs *sat* or *unsat*
- Pseudocode:

if $\mathrm{var}(F) = \emptyset$ then

if $F = \emptyset$ then exit(sat).

else exit(unsat). /* $F = \{\Box\}$ where $\Box$ is empty clause */

else choose $x \in \mathrm{var}(F)$,

/* x is the branching variable */

if A($F[x = 0]$) = sat then exit(sat).

else if A($F[x = 1]$) = sat then exit(sat).

else exit(unsat).

Pure Literals

- A literal x of a clause-set F is a pure literal of F if some clauses of F contain x but no clause of F contains $\overline{x}$.
- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.

Pure Literals

- A literal x of a clause-set F is a pure literal of F if some clauses of F contain x but no clause of F contains $\overline{x}$.
- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- Let F' be the clause-set obtained from F by removing all clauses that contain pure literals.

  Then F and F' are equisatisfiable (why?).

  We say that F' is obtained from F by pure literal elimination.

Applying Pure Literal Elimination

- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- In the above example we obtain $F' = \{\{\overline{y}, \overline{z}\}\}$. Now $\overline{y}$ and $\overline{z}$ are pure literals of F', and we can apply pure literal elimination again, obtaining the empty clause-set.

Applying Pure Literal Elimination

- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- In the above example we obtain $F' = \{\{\overline{y}, \overline{z}\}\}$. Now $\overline{y}$ and $\overline{z}$ are pure literals of F', and we can apply pure literal elimination again, obtaining the empty clause-set.
- For a clause-set F we denote by $PL(F)$ the smallest clause-set that can be obtained from F by (possibly repeated) applications of pure literal elimination.
- In the above example we have $PL(F) = \emptyset$.

Applying Pure Literal Elimination

- Example: $F = \{\{x, y\}, \{\overline{y}, \overline{z}\}, \{z, x\}, \{u, \overline{z}\}\}$.
  x and u are the pure literals of F.
- In the above example we obtain $F' = \{\{\overline{y}, \overline{z}\}\}$. Now $\overline{y}$ and $\overline{z}$ are pure literals of F', and we can apply pure literal elimination again, obtaining the empty clause-set.
- For a clause-set F we denote by $PL(F)$ the smallest clause-set that can be obtained from F by (possibly repeated) applications of pure literal elimination.
- In the above example we have $PL(F) = \emptyset$.
- F and $PL(F)$ are always equisatisfiable.

Unit Propagation

- When a clause-set F contains a unit clause $\{\ell\}$, we can obtain by unit propagation the clause-set $F[\ell = 1]$ from F.
- In that case F and $F[\ell = 1]$ are equisatisfiable (why?)
- We write $UP(F)$ for the clause-set obtained from F by applying unit propagation as often as possible.

Unit Propagation

- When a clause-set F contains a unit clause $\{\ell\}$, we can obtain by unit propagation the clause-set $F[\ell = 1]$ from F.
- In that case F and $F[\ell = 1]$ are equisatisfiable (why?)
- We write $UP(F)$ for the clause-set obtained from F by applying unit propagation as often as possible.
- Example: Let $F = \{\{x, y\}, \{\overline{y}\}, \{z, \overline{x}, v\}\}$.
  - From F we obtain by unit propagation the clause set $F' = \{\{x\}, \{z, \overline{x}, v\}\}$.
  - With a second step of unit propagation we obtain from F' the clause-set $F'' = \{\{z, v\}\}$.

  Consequently $UP(F) = \{\{z, v\}\}$.
- F and $UP(F)$ are always equisatisfiable.

First UP and then PL, or vice versa?

Fact

For any clause-set F, we have $UP(PL(UP(F))) = PL(UP(F))$.

Proof: exercise (Question 3a.i in 2009 Advanced AI exam)

Fact

There is a clause-set F such that $PL(UP(PL(F))) \neq UP(PL(F))$.

Proof: exercise (Question 3a.ii in 2009 Advanced AI exam)

So, first UP and then PL, or the other way around?

The DPLL algorithm

- `Algorithm DPLL`(F)
- `outputs` *sat* `or` *unsat*

$F := UP(F)$.

$F := PL(F)$.

`if` $\mathrm{var}(F) = \emptyset$ `then`

    `if` $F = \emptyset$ `then exit(sat).`

    `else exit(unsat).   /*` $F = \{\square\}$ `*/`

`else choose` $x \in \mathrm{var}(F)$,

`/*` x `is the branching variable */`

`if A`$(F[x = 0]) =$ `sat then exit(sat).`

`else if A`$(F[x = 1]) =$ `sat then exit(sat).`

`else exit(unsat).`

Some Well-Known DPLL-based SAT Solvers

- Important SAT solvers:
  - Grasp (Marques-Silva & Sakallah 1996)
  - Relsat (Bayardo Jr. & Schrag 1997)
  - chaff (Moskewicz et al 2001), zChaff (Zhang 2001)
  - Minisat (Een & Sörensson 2003)
  - RSat (Pipatsrisawat & Darwiche 2007)
  - Glucose (Audemard & Simon 2009)
  - CryptoMiniSat (Soos 2010)
  - Lingeling, PicoSAT (Biere 2010)
- Many of them are available for free and continually improved.

Some Well-Known DPLL-based SAT Solvers

- Important SAT solvers:
  - Grasp (Marques-Silva & Sakallah 1996)
  - Relsat (Bayardo Jr. & Schrag 1997)
  - chaff (Moskewicz et al 2001), zChaff (Zhang 2001)
  - Minisat (Een & Sörensson 2003)
  - RSat (Pipatsrisawat & Darwiche 2007)
  - Glucose (Audemard & Simon 2009)
  - CryptoMiniSat (Soos 2010)
  - Lingeling, PicoSAT (Biere 2010)
- Many of them are available for free and continually improved.
- Key improvements to DPLL that boosted the performance:
  - combination of clever branching heuristics with
  - clause learning,
  - non-chronological backtracking,
  - restart strategies,
  - implementation of propagation.
- Check *www.satlive.org* for more info on SAT solvers!