An Introduction to SAT Solving

Applied Logic for Computer Science

UWO – December 3, 2017

Plan

1. The Boolean satisfiability problem
2. Converting formulas to conjunctive normal form
3. Tseytin's transformation
4. How SAT solvers work

Plan

Propositional formula

Definition

Let V be a finite set of Boolean valued variables. A *propositional formula* on V is defined inductively as follows:

- each of the constants false, true is a propositional formula on V,
- any element of V is a propositional formula on V,
- if ϕ and ϕ' are propositional formulas on V then $\neg\phi$, (ϕ), $\phi \wedge \phi'$, $\phi \vee \phi'$, $\phi \rightarrow \phi'$, $\phi \leftrightarrow \phi'$ are propositional formulas on V as well.

Examples and counter-examples

- $p \vee \neg q$ is a propositional formula on $V = \{p, q\}$,
- $p + \neg q$ is **not** a propositional formula on $V = \{p, q\}$,
- $p \vee \neg q$ is **not** a propositional formula on $V = \{p\}$,
- $(\,)$ is **not** a propositional formula on $V = \{p, q\}$.

Assignment

Definition

Let again V be a finite set of Boolean valued variables.

- An *assignment on* V is any map from V to $\{\text{false}, \text{true}\}$.
- Any assignment $\mathbf{v}$ on V induces an assignment on the propositional formulas on V by applying the following rules: if ϕ and ϕ' are propositional formulas on V then we have:
  1. $\mathbf{v}(\neg\phi) = \neg\mathbf{v}(\phi)$,
  2. $\mathbf{v}(\phi \wedge \phi') = \mathbf{v}(\phi) \wedge \mathbf{v}(\phi)'$,
  3. $\mathbf{v}(\phi \vee \phi') = \mathbf{v}(\phi) \vee \mathbf{v}(\phi')$,
  4. $\mathbf{v}(\phi \rightarrow \phi') = \mathbf{v}(\phi) \rightarrow \mathbf{v}(\phi')$,
  5. $\mathbf{v}(\phi \leftrightarrow \phi') = \mathbf{v}(\phi) \leftrightarrow \mathbf{v}(\phi')$.

Example

For the set of Boolean variables $V = \{p, q\}$. define $\mathbf{v}(p)$ = false and $\mathbf{v}(q)$ = true. Then, we have:

- $\mathbf{v}(p \rightarrow q)$ = true.
- $\mathbf{v}(p \leftrightarrow q)$ = false.

Satisfiability (1/4)

Definition

Let again V be a finite set of Boolean valued variables and let ϕ propositional formula on V.

- An assignment $\mathbf{v}$ on V is a *satisfying assignment* for ϕ if we have $\mathbf{v}(\phi)$ = true.
- The propositional formula ϕ is said *satisfiable* if there exists a satisfying assignment for ϕ.

Deciding whether or not a propositional formula is satisfiable is called the *Boolean satisfiability problem*, denoted by SAT.

Example

$p \wedge (q \vee \neg p) \wedge (\neg q \vee \neg r)$ is satisfiable for $\mathbf{v}(p)$ = true, $\mathbf{v}(q)$ = true, $\mathbf{v}(r)$ = false.

Satisfiability (2/4)

Example

The formula is $p \wedge (q \vee \neg p) \wedge (\neg q \vee \neg p)$ is unsatisfiable. Indeed:

- the first clause, namely p, implies that p must be true,
- then, the second clause, namely $(q \vee \neg p)$, implies that q must be true,
- then, the third clause, namely $(\neg q \vee \neg p)$ is false,
- thus the whole formula cannot be satisfied.

Remarks

- Simple method for checking satisfiability: the *truth table* method, that is, check all 2^n possibilities for $\mathbf{v}$, where n is the number of variables in V.
- This always works, but has a running time growing exponentially with n.
- Practical algorithms and software solving SAT problems also run in time $O(2^n)$ but plays many tricks to terminate computations as early as possible.

Satisfiability (3/4)

Eight queens puzzle: general statement

- Go to `https://en.wikipedia.org/wiki/Eight_queens_puzzle` and read.
- For $n = 4$, there are two solutions.
- **Exercise**: how to phrase the search for those solutions into a SAT problem?
- **Hints**:
  - What should the Boolean variables represent?
  - What should the propositional formula represent?
- Remember the rules:
  - at most one (and at least one) queen in every row,
  - at most one (and at least one) queen in every column,
  - at most one queen in every diagonal.

Satisfiability (4/4)

Eight queens puzzle: case $n = 2$

- Associate a Boolean variable with each of the four corners, say a, b, c and d in clock-wise order.
- Exactly one queen on the top row writes: $(a \vee b) \wedge \neg(a \wedge b)$.
- Exactly one queen on the bottom row writes: $(c \vee d) \wedge \neg(c \wedge d)$.
- Exactly one queen on the left column writes: $(a \vee d) \wedge \neg(a \wedge d)$.
- Exactly one queen on the left column writes: $(b \vee c) \wedge \neg(b \wedge c)$.
- No two queens on the same diagonal writes: $\neg(a \wedge c) \wedge \neg(b \wedge d)$.
- We need some help to determine of the conjunction of these 5 formulas is satisfiable!

SAT solvers

- Modern SAT solvers are based on *resolution*, and only apply to input formulas in *conjunctive normal form* (CNF).
- A *conjunctive normal* form (CNF) is a conjunction of clauses
- A *clause* is a disjunction of literals, say $a_1 \vee \cdots \vee a_n$, where $a_1, \ldots, a_n$ are literals.
- A *literal* is a Boolean variable or the negation of a Boolean variable
- Hence a CNF is of the form

$$\bigwedge_{1 \le i \le s} \left(\bigvee_{1 \le j \le t_j} \ell_{i,j} \right)$$

  where the $\ell_{i,j}$'s are literals.

SAT solvers have many applications. Problems involving

- binary arithmetic
- program correctness
- termination of rewriting
- puzzles like Sudoku

can be encoded as SAT problems

The yices SAT solver (1/3)

Calling sequence

- The solver `yices` is simply used by calling

```
yices -e -smt test.smt
```

  where `test.smt` contains the formula to test.
- The input file uses a Lisp-like syntax where `and` and `or` can have any number of arguments.

Availability

- The SAT solver `yices` is publicly available at `http://yices.csl.sri.com/old/download-yices1-full.html` for Linux, MacOS and Windows.
- **Important**: `yices` requires the GMP library `https://gmplib.org/`.
- Installation under Linux and MacOS is straightforward; some work is needed under Windows, if GMP is not yet installed; see the details in OWL.
- **Important**: Here, we use the version **1** of `yices`.

The yices 1 SAT solver (2/3)

Example

```
(benchmark test.smt
:extrapreds ((A) (B) (C) (D))
:formula (and
(iff A (and D B))
(implies C B)
(not (or A B (not D)))
(or (and (not A) C) D)
))
```

produces

```
sat
(= A false)
(= B false)
(= D true)
(= C false)
```

The `yices 1` SAT solver (3/3)

Returning to the n queens puzzle for $n = 2$.

Example

```
(benchmark test.smt :extrapreds ((A) (B) (C) (D))
              :formula (and (and (or A B) (not (and A B)))
                            (and (or C D) (not (and C D)))
                            (and (or A D) (not (and A D)))
                            (and (or C B) (not (and C B)))
                            (and (not (and A C)) (not (and B D)))
))
```

produces

```
unsat
```

Plan

Conjunctive normal form

Recall

- A *literal* is either a propositional variable or the negation of a propositional variable.
- A *clause* is a disjunction of literals.
- A formula is in *conjunctive normal form* (CNF), if it is a conjunction of clauses; for instance:

  $$(p \vee \neg q \vee r) \wedge (q \vee r) \wedge (\neg p \vee \neg q) \wedge r$$

  is a CNF on $V = \{p, q, r\}$.

definition

Let V a finite set of Boolean valued variables.

- Two clauses C, C' on V are *equivalent* if they have the same truth table; in this case we write $C \leftrightarrow C'$.
- Similarly, two formulas ϕ, ψ on V are *equivalent* if they have the same truth table; in this case we write $\phi \leftrightarrow \psi$.

Properties of clauses

Recall

Let V a finite set of Boolean valued variables and ϕ, ψ be propositional formulas on V. The following properties hold:

commutativity of $\wedge$: $\phi \wedge \psi \quad \leftrightarrow \quad \psi \wedge \phi$,

commutativity of $\vee$: $\phi \vee \psi \quad \leftrightarrow \psi \vee \phi$,

absorption of $\wedge$: $\phi \wedge \phi \quad \leftrightarrow \quad \phi$,

absorption of $\vee$: $\phi \vee \phi \quad \leftrightarrow \quad \phi$.

Consequences

- If a clause C' is obtained by reordering the literals of a clause C then the two clauses are equivalent.
- If a clause contains more than one occurrence of the same literal then it is equivalent to the close obtained by deleting all but one of these occurrences.
- From these properties, we can represent a clause as a set of literals, by making disjunction implicit and by ignoring replication and order of literals; for instance $(p \vee q \vee r \vee \neg r)$ is represented by the set $\{p, q, r, \neg r\}$.

Properties of clauses

Properties

- The order of clauses in a CNF formula does not matter; for instance we have:

$$(a \vee b) \wedge (c \vee \neg b) \wedge (\neg b) \quad \leftrightarrow \quad (c \vee \neg b) \wedge (\neg b) \wedge (a \vee b).$$

- If a CNF formula contains more than one occurrence of the same clause then it is equivalent to the formula obtained by deleting all but one of the duplicated occurrences:

$$(a \vee b) \wedge (c \vee \neg b) \wedge (a \vee b) \quad \leftrightarrow \quad (a \vee b) \wedge (c \vee \neg b).$$

- From the properties of clauses and of CNF formulas, we can represent a CNF formula as a set of sets of literals; for instance: $(a \vee b) \wedge (c \vee \neg b) \wedge (\neg b)$ is represented by the set $\{\{a, b\}, \{c, \neg b\}, \{\neg b\}\}$.

Conversion to CNF

Proposition

Let V a finite set of Boolean valued variables. Let ϕ be a formula on V. Then there exists a CNF formula ψ such that we $\phi \leftrightarrow \psi$.

Idea of the proof

- The proof by defining a function CNF which turns any formula ϕ on V into a CNF formula a ψ.
- The function CNF is defined inductively.
- We will prove that this definition is well-formed in the sense that it does not lead to infinitely many recursive calls.
- Then, we will prove by structural induction that this function computes correctly a CNF formula equivalent to the its input argument.
- The function CNF essentially follows the rules of Boolean algebra. See previous lectures or the *monotone laws* in `https://en.wikipedia.org/wiki/Boolean_algebra`.

The function CNF (1/4)

Let V a finite set of Boolean valued variables. For any two formulas ϕ, ψ on V and any Boolean variable $p \in V$, we define:

1. $\mathsf{CNF}(p) = p$
2. $\mathsf{CNF}(\neg p) = \neg p$
3. $\mathsf{CNF}(\phi \to \psi) = \mathsf{CNF}(\neg\phi) \otimes \mathsf{CNF}(\psi)$
4. $\mathsf{CNF}(\phi \wedge \psi) = \mathsf{CNF}(\phi) \wedge \mathsf{CNF}(\psi)$
5. $\mathsf{CNF}(\phi \vee \psi) = \mathsf{CNF}(\phi) \otimes \mathsf{CNF}(\psi)$
6. $\mathsf{CNF}(\phi \leftrightarrow \psi) = \mathsf{CNF}(\phi \to \psi) \wedge \mathsf{CNF}(\psi \to \phi)$
7. $\mathsf{CNF}(\neg\neg\phi) = \mathsf{CNF}(\phi)$
8. $\mathsf{CNF}(\neg(\phi \to \psi)) = \mathsf{CNF}(\phi) \wedge \mathsf{CNF}(\neg\psi)$
9. $\mathsf{CNF}(\neg(\phi \wedge \psi)) = \mathsf{CNF}(\neg\phi) \otimes \mathsf{CNF}(\neg\psi)$
10. $\mathsf{CNF}(\neg(\phi \vee \psi)) = \mathsf{CNF}(\neg\phi) \wedge \mathsf{CNF}(\neg\psi)$
11. $\mathsf{CNF}(\neg(\phi \leftrightarrow \psi) = \mathsf{CNF}(\phi \wedge \neg\psi) \otimes \mathsf{CNF}(\psi \wedge \neg\phi)$

where $(C_1 \wedge \ldots \wedge C_n) \otimes (D_1 \wedge \cdots \wedge D_m)$ is defined as $(C_1 \vee D_1) \wedge \cdots \wedge (C_1 \vee D_m) \wedge \cdots \wedge (C_n \vee D_1) \wedge \cdots \wedge (C_n \vee D_m)$ where $C_1, \ldots, C_n, D_1, \ldots, D_m$ are clauses.

The function CNF (2/4)

Remarks

- The first two rules can be understood as a *base case*.
- The other nine rules can be understood as the *inductive step*.

Example

Consider $V = \{a, b, c, d\}$. we have:

$$\begin{array}{ll}
\mathsf{CNF}((a \wedge b) \vee (c \wedge d)) & \leftrightarrow \\
\mathsf{CNF}(a \wedge b) \otimes \mathsf{CNF}(c \wedge d) & \leftrightarrow \\
(\mathsf{CNF}(a) \wedge \mathsf{CNF}(b)) \otimes (\mathsf{CNF}(c) \wedge \mathsf{CNF}(d)) & \leftrightarrow \\
(a \wedge b) \otimes (c \wedge d) & \leftrightarrow \\
(a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \wedge d) &
\end{array}$$

Lemma

With $C_1, \ldots, C_n, D_1, \ldots, D_m$ as above, we have

$$(C_1 \wedge \ldots \wedge C_n) \otimes (D_1 \wedge \cdots \wedge D_m) \quad \leftrightarrow \quad (C_1 \wedge \ldots \wedge C_n) \vee (D_1 \wedge \cdots \wedge D_m).$$

The function CNF (3/4)

Proof of the lemma (1/2)

We observe that proving the lemma means that the following equivalence holds:

$$\begin{array}{c} (C_1 \vee D_1) \wedge \cdots \wedge (C_1 1 \vee D_m) \wedge \cdots \wedge (C_n \vee D_1) \wedge \cdots \wedge (C_n \vee D_m) \\ \leftrightarrow \\ (C_1 \wedge \ldots \wedge C_n) \vee (D_1 \wedge \cdots \wedge D_m). \end{array} \tag{1}$$

The proof of Relation (1) can be done by induction on $n + m \geq 2$, for all positive integers n, m.

- **Base case.** That is, $n + m = 2$, thus $n = m = 1$. In this case, we have

$$\begin{array}{lr} (C_1 \wedge \ldots \wedge C_n) \vee (D_1 \wedge \cdots \wedge D_m) & \leftrightarrow \\ (C_1) \vee (D_1) & \leftrightarrow \\ (C_1 \vee D_1) \wedge \cdots \wedge (C_1 1 \vee D_m) \wedge \cdots \wedge (C_n \vee D_1) \wedge \cdots \wedge (C_n \vee D_m) & \end{array}$$

- **Step case.** Assume tha the property holds for $n + m \geq 2$, with positive integers n, m. We shall prove that it holds for $n + (m + 1)$ and $(n + 1) + m$ as well. Both cases being similar, we just do one, say $n + (m + 1)$.

The function CNF (4/4)

Proof of the lemma (1/2)

By associativity of $\wedge$, we have:

$$(C_1 \wedge \ldots \wedge C_n) \vee (D_1 \wedge \cdots \wedge D_{m+1})$$
$$\leftrightarrow$$
$$(C_1 \wedge \ldots \wedge C_n) \vee ((D_1 \wedge \cdots \wedge D_m) \wedge D_{m+1})$$

By distributivity of $\vee$ over $\wedge$, we obtain:

$$(C_1 \wedge \ldots \wedge C_n) \vee (D_1 \wedge \cdots \wedge D_{m+1})$$
$$\leftrightarrow$$
$$(C_1 \wedge \ldots \wedge C_n) \vee (D_1 \wedge \cdots \wedge D_m) \wedge (C_1 \wedge \ldots \wedge C_n) \vee D_{m+1}$$

By induction hypothesis, we deduce:

$$\begin{array}{ll} (C_1 \wedge \ldots \wedge C_n) \vee (D_1 \wedge \cdots \wedge D_{m+1}) & \\ & \leftrightarrow \\ ((C_1 \vee D_1) \wedge \cdots \wedge (C_1 \vee D_m) \wedge \cdots \wedge (C_n \vee D_1) \wedge \cdots \wedge (C_n \vee D_m)) & \\ \wedge ((C_1 1 \vee D_{m+1}) \wedge \cdots \wedge (C_n \vee D_{m+1})) . & \end{array}$$

Finally, we conclude that the desired property holds by associativity and commutativity of $\wedge$.

Any call to the function CNF terminates

Proposition

The function CNF *terminates for every input formula* ϕ *on* V.

Proof

- We define the *complexity* of the formula ϕ, denoted by Complexity(ϕ), as the maximal number of nested logical operators $\neg, \wedge, \vee$ it contains explicitly or implicitly (when rewriting $\leftrightarrow$ and $\rightarrow$).
- Hence for all $p \in V$, we have Complexity(p) $= 0$ and Complexity($\neg p$) $= 1$.
- For all formulas ϕ, ψ on V, we have:

$$\begin{array}{lcc} \text{Complexity}(\phi \vee \psi) & = & \text{Complexity}(\phi) + \text{Complexity}(\psi) + 1, \\ \text{Complexity}(\phi \rightarrow \psi) & = & \text{Complexity}(\neg\phi) + \text{Complexity}(\psi) + 1, \\ \text{Complexity}(\phi \leftrightarrow \psi) & = & \text{Complexity}(\phi \rightarrow \psi) + \text{Complexity}(\psi \rightarrow \phi) + 1. \end{array}$$

- The termination of CNF is guaranteed since the complexity of the formula given in input to all the recursive applications of CNF is always decreasing.
- Since the complexity of every formula is finite, then after a finite number of recursive calls of CNF, the base case is reached.

The function CNF is correct

Proposition

For an input formula ϕ on V, the function CNF *computes a CNF formula ψ on V, such that we have $\phi \leftrightarrow \psi$.*

Proof

By induction on the definition of the function CNF:

Base case: clearly CNF(ϕ) is in CNF for the first two rules and we have $\phi \leftrightarrow \mathrm{CNF}(\phi)$.

Step case: Consider the third rule.

- By the induction hypothesis, CNF($\neg\phi$) and CNF(ψ) are in CNF, and respectively equivalent to $\neg\phi$ and ψ
- Thanks to the above lemma, we deduce:

$$\mathrm{CNF}(\neg\phi) \otimes \mathrm{CNF}(\psi) \leftrightarrow \mathrm{CNF}(\neg\phi) \vee \mathrm{CNF}(\psi) \leftrightarrow (\neg\phi) \vee \psi.$$

- Therefore, we have proved $\mathrm{CNF}(\phi \to \psi) \leftrightarrow (\neg\phi) \vee \psi$.
- By induction hypothesis CNF($\neg\phi$) and CNF(ψ) are in CNF.
- Therefore, CNF($\phi \to \psi$) is in CNF as well.

Conversion to CNF: bad news

Remark

Proceeding as in the proof of the above proposition, the size of the output formula *blows up* with respect to the size of the input formula. Consider:

$$(p_1 \wedge \cdots \wedge p_n) \vee (q_1 \wedge \cdots \wedge q_m)$$

will produce nm clauses. In fact, one can prove the following result, given here without proof.

Proposition

Every CNF equivalent to the formula below has at least $2^n - 1$ *clauses;*

$$(\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3)\cdots \leftrightarrow p_n)$$

Plan

Dealing with exponential explosion

Example

In the formula

$$p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow (p_5 \leftrightarrow p_6))))$$

replace, the sub-formula $(p_5 \leftrightarrow p_6)$ with a new variable n_1, we obtain

$$p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (p_4 \leftrightarrow n_1))).$$

Then replacing successively $p_4 \leftrightarrow n_1$ with n_2, $p_3 \leftrightarrow n_2$ with n_3, $p_2 \leftrightarrow n_3$ with n_4, $p_1 \leftrightarrow n_4$ with n_5, we obtain a CNF formula

$$\begin{aligned} & n_5 \wedge (n_5 \leftrightarrow (p_1 \leftrightarrow n_4)) \wedge (n_4 \leftrightarrow (p_2 \leftrightarrow n_3)) \wedge (n_3 \leftrightarrow (p_3 \leftrightarrow n_2)) \\ & \wedge (n_2 \leftrightarrow (p_4 \leftrightarrow n_1)) \wedge (n_2 \leftrightarrow (p_5 \leftrightarrow p_6)). \end{aligned}$$

We shall see this strategy can produce a CNF formula whose size grows linearly with that of the input formula.

Equisatisfiability

Definition

Two propositional formulas α and β are *equisatisfiable* if:

one is satisfiable if and only if the other is satisfiable.

Example

- if p and q are two Boolean variables, then p and q are equisatisfiable.
- If a, b, n are three Boolean variables, then $a \wedge b$ and $(n \leftrightarrow (a \wedge b)) \wedge n$ are equisatisfiable; however, they are not equivalent.

Conversion to CNF: good news (1/2)

Proposition

For any propositional logic ϕ on a finite set of Boolean variables V, one can compute a finite set of Boolean variables V' and a propositional logic ψ such that:

- *$V \subseteq V'$ holds,*
- *ϕ and ψ are equisatisfiable,*
- *the size of ψ (counting connectives and literals) is proportional to the size ϕ.*

Proof and algorithm (1/2)

Step 1: Introduce a new variable p_ψ for every sub-formula ψ of ϕ (unless ψ is a literal)

- For instance, if $\psi = \psi_1 \wedge \psi_2$, introduce two new variables p_{ψ_1} and p_{ψ_2} representing ψ_1 and ψ_2 respectively.

Step 2: Consider each sub-formula $\psi = \psi_1 \circ \psi_2$, where $\circ$ is an arbitrary Boolean connective.

- Stipulate that the representative of ψ is equivalent to the representative of $\psi_1 \circ \psi_2$, that is, $p_\psi \leftrightarrow p_{\psi_1} \circ p_{\psi_2}$.

Conversion to CNF: good news (1/2)

Proof and algorithm (1/2)

Step 3: Convert $p_\psi \leftrightarrow p_{\psi_1} \circ p_{\psi_2}$ to CNF using the function CNF defined in the previous section.

- Since $p_\psi \leftrightarrow p_{\psi_1} \circ p_{\psi_2}$ contains at most three propositional variables and exactly two connectives, the size of this formula in CNF is bounded over by a constant.

- For the original formula ϕ, let p_ϕ be its representative and let subf(ϕ) be the set of all sub-formulas of ϕ, including ϕ itself.
- Then the output formula is:

$$p_\phi \ \wedge \ \left(\bigvee_{\psi_1 \circ \psi_2 \in \mathsf{subf}(\phi)} \ \mathsf{CNF}(p_\psi \leftrightarrow p_{\psi_1} \circ p_{\psi_2}) \right)$$

- This formula is equisatisfiable to ϕ. Moreover, it is in CNF.
- Let n be the size of ϕ.
- The number of elements in subf(ϕ) is bounded over by n
- Since CNF($p_\psi \leftrightarrow p_{\psi_1} \circ p_{\psi_2}$) has a size bounded over by a constant, say C, we deduce that the size of the output formula is in big-oh of $C\,n$.

Tseytins transformation: example (1/2)

Consider the following formula $\phi := ((p \vee q) \wedge r) \rightarrow (\neg s)$
Consider all sub-formulas (except the variables themselves):

$$\neg s \tag{2}$$

$$p \vee q \tag{3}$$

$$(p \vee q) \wedge r \tag{4}$$

$$((p \vee q) \wedge r) \rightarrow (\neg s) \tag{5}$$

Introduce a new variable for each sub-formula:

$$x_1 \leftrightarrow \neg s \tag{6}$$

$$x_2 \leftrightarrow p \vee q \tag{7}$$

$$x_3 \leftrightarrow x_2 \wedge r \tag{8}$$

$$x_4 \leftrightarrow x_3 \rightarrow x_1 \tag{9}$$

Conjunct all substitutions and the substitution for ϕ:

$$T(\phi) := x_4 \wedge (x_4 \leftrightarrow x_3 \rightarrow x_1) \wedge (x_3 \leftrightarrow x_2 \wedge r) \wedge (x_2 \leftrightarrow p \vee q) \wedge (x_1 \leftrightarrow \neg s)$$

Tseytins transformation: example (1/2)

All substitutions can be transformed into CNF, e.g.

$$\begin{aligned} x_2 \leftrightarrow p \vee q &\equiv x_2 \rightarrow (p \vee q) \wedge ((p \vee q) \rightarrow x_2) && (10)\\ &\equiv (\neg x_2 \vee p \vee q) \wedge (\neg(p \vee q) \vee x_2) && (11)\\ &\equiv (\neg x_2 \vee p \vee q) \wedge ((\neg p \wedge \neg q) \vee x_2) && (12)\\ &\equiv (\neg x_2 \vee p \vee q) \wedge (\neg p \vee x_2) \wedge (\neg q \vee x_2) && (13) \end{aligned}$$

See `https://en.wikipedia.org/wiki/Tseytin_transformation` for more details on Tseytins transformation.

Plan

Satisfiability of a CNF formula

Notations

- Let $C_1, \ldots, C_m$ be clauses over a finite set V of Boolean variables $p_1, \ldots, p_n$.
- Define $\phi := C_1 \wedge \cdots \wedge C_m$.
- Let $\mathbf{v}$ be an assignment on V.

Remarks

Observations:

1. The CNF formula ϕ is satisfiable by $\mathbf{v}$ if, and only if, the clause C_i, for all $i \in \{1, \ldots, m\}$, is satisfiable by $\mathbf{v}$.
2. A clause C_i, for some $i \in \{1, \ldots, m\}$, is satisfiable by $\mathbf{v}$ if, and only if, at least one of its literals evaluates to true by $\mathbf{v}$.

Consequences:

- To check whether or not $\mathbf{v}$ satisfies ϕ, we may not need to know the truth values of all literals in all clauses.
- For instance, if $\mathbf{v}(p) =$ true and $\mathbf{v}(q) =$ false, we can see that the formula $\phi = (p \vee q \vee \neg r) \wedge (\neg q \vee s \vee q)$ is satisfied without considering $\mathbf{v}(r)$ and $\mathbf{v}(s)$.

Partial assignment

Definition

- A *partial assignment* on V is a partial function that associates to some Boolean variables of V a truth value (either true or false) and can be undefined for the others.
- Under a partial assignment on V, a clause C on V can be
  - *true* if one of its literals is true,
  - *false* (or *conflicting*) if all its literals are false,
  - *undefined* (or unresolved) if it is neither *true* not *false*.

Remarks

- Partial assignments allow us to construct assignments for a set of clauses incrementally, that is, one clause after another.
- The SAT solving algorithms to be presented next use partial assignments.
- They all start with an empty assignment (that is, the truth values of all Boolean variables are not defined) and try to extend this assignment, assigning one Boolean variable after another.

Simplification of a formula by an evaluated literal (1/2)

Definition

For the CNF formula ϕ and a Boolean variables p, we denote by $\phi|_p$ the formula obtained from ϕ by replacing all occurrences of p by true and simplifying the result by removing:

- all true clauses,
- all false literals from undefined clauses.

The operation that maps (ϕ, p) to $\phi|_p$ is called the *simplification* of ϕ at p. Similarly, we define $\phi|_{\neg p}$ as the formula obtained from ϕ by replacing all occurrences of p by false and simplifying the result by removing:

- all true clauses,
- all false literals from undefined clauses.

The operation that maps (ϕ, p) to $\phi|_{\neg p}$ is called the *simplification* of ϕ at $\neg p$.

Simplification of a formula by an evaluated literal (2/2)

Example

For $\phi := (p \vee q \vee \neg r) \wedge (\neg p \vee \neg r)$ we have:

$$\phi|_{\neg p} \quad \leftrightarrow \quad q \vee \neg r.$$

Proposition

For ϕ and p as in the above definition, we have:

- *if $\phi|_p$ is satisfiable, then ϕ is satisfiable too,*
- *if $\phi|_{\neg p}$ is satisfiable, then ϕ is satisfiable too.*
- *if ϕ is satisfiable then either $\phi|_p$ or $\phi|_{\neg p}$ is satisfiable.*

Remarks

- The above proposition is a key argument in the SAT solving algorithms to be presented hereafter.
- The proof of this proposition is easy and left as an exercise

Principles of SAT solving algorithms

Principles

The SAT solving algorithms presented hereafter are based on the following ideas.

1. Each algorithm is stated as a recursive function taking a propositional formula ϕ and a partial assignment $\mathbf{v}$ as arguments.
2. These functions may modify their arguments:
   - the partial assignment can be extended,
   - the formula ϕ can be simplified at the newly assigned literal.
3. Before extending the assignment $\mathbf{v}$ and simplifying the formula ϕ, the function checks whether ϕ can be proved to be true or false, whatever are the values of the remaining unassigned Boolean variables.

Naive solver (1/3)

Input: A propositional formula ϕ on a finite set V and a partial assignment $\mathbf{v}$ on V such that only variables unassigned by $\mathbf{v}$ occur in ϕ.

Output: true if ϕ is satisfiable, false otherwise.

```
Naive_SAT($\phi, \mathbf{v}$) {
  1 if every clause of $\phi$ has a true literal, return true;
  2 if any clause of $\phi$ has all false literals, return false;
  3 choose an $p \in V$ that is unassigned in $\mathbf{v}$;
  4 assign $p$ to true, that is, let $\mathbf{v}(p) =$ true;
  5 if Naive_SAT($\phi|_p, \mathbf{v}$) returns true, then return true; # this is a recursive call
  6 assign $p$ to false, that is, let $\mathbf{v}(p) =$ false;
  7 if Naive_SAT($\phi|_{\neg p}, \mathbf{v}$) returns true, then return true; # this is a recursive call
  8 unassign $\mathbf{v}(p)$; # backtracking takes place here
  9 return false; }
```

Naive solver (2/3)

Remarks

The function call Naive_SAT$(\phi, \mathbf{v})$ can terminate early if:

- the formula is satisfied before all truth assignments are tested,
- all clauses are false before all variables have been assigned.

Proposition

The call Naive_SAT$(\phi, \mathbf{v})$ *terminates for all* ϕ *and* $\mathbf{v}$.

Naive solver (2/3)

Remarks

The function call Naive_SAT($\phi, \mathbf{v}$) can terminate early if:

- the formula is satisfied before all truth assignments are tested,
- all clauses are false before all variables have been assigned.

Proposition

The call Naive_SAT($\phi, \mathbf{v}$) *terminates for all* ϕ *and* $\mathbf{v}$.

Proof

We can proceed by induction on the number s of unassigned Boolean variables. Note that $0 \leq s \leq n$ always holds, where n is the number of Boolean variables in V.

- if $s = 0$ then no recursive calls happen and termination is clear.
- if $0 < s \leq n$, we assume (by induction hypothesis) that the recursive calls terminate; then Naive_SAT($\phi, \mathbf{v}$) clearly terminates as well.

Naive solver (3/3)

Proposition

The call Naive_SAT$(\phi, \mathbf{v})$ *correctly decides whether* ϕ *is satisfiable.*

Naive solver (3/3)

Proposition

The call Naive_SAT$(\phi, \mathbf{v})$ *correctly decides whether* ϕ *is satisfiable.*

Proof

We can proceed again by induction on the number s of unassigned Boolean variables. Recall that $0 \le s \le n$ always holds, where n is the number of Boolean variables in V.

- if $s = 0$ then no recursive calls happen and correctness is clear.
- Assume $0 < s \le n$. Assume also, by induction hypothesis, that the recursive calls are correct.
  1. if one of them returns true, say Naive_SAT$(\phi|_p, \mathbf{v})$, then $\phi|_p$ is satisfiable and therefore so is ϕ.
  2. if none of them returns true, then ϕ is not satisfiable.
- In the first case, Naive_SAT$(\phi, \mathbf{v})$ correctly returns true.
- In the second case, Naive_SAT$(\phi, \mathbf{v})$ correctly returns false.
- Therefore, Naive_SAT$(\phi, \mathbf{v})$ correctly decides whether ϕ is satisfiable.

The pure literal rule

Proposition

Given a propositional CNF formula ϕ for which we check whether ϕ is satisfiable or not, if a variable is always positive *(that is, never appears with a $\neg$) or* always negative *(that is, always appears with a $\neg$) in ϕ, one only needs to set it to one value:* true *for positive variables,* false *for negative variables. Note that such variables are called* pure*.*

Proof

- Suppose p occurs only as positive literals in ϕ,
- If ϕ is satisfied by $\mathbf{v}$ and $\mathbf{v}(p) =$ false, then ϕ is also satisfied by $\mathbf{v}'$ which is identical to $\mathbf{v}$ except that $\mathbf{v}'(p) =$ true.
- so we do not need to try $\mathbf{v}(p) =$ false.

Remarks

- Note that literals may become pure as variables are assigned.
- Indeed, as simplification happens, true clauses are removed.
- Hence, an unassigned variable which was not pure may become pure.

Unit propagation

Principle

- Unit propagation (a.k.a Boolean constraint propagation or BCP) is a key component to fast SAT solving.
- Whenever all the literals in a clause are false except one, the remaining literal must be true in any satisfying assignment; such a clause is called a **unit clause**.
- Therefore, the algorithm can assign it to true immediately.
- After assigning a variable, there are often many unit clauses.
- Setting a literal in a unit clause often creates other unit clauses, leading to a cascade.
- Based on those observations, we define a sub-algorithm BCP as follows.
- The input specifications of BCP$(\phi, \mathbf{v})$ are the same as Naive_SAT$(\phi, \mathbf{v})$.
- Moreover, similarly to Naive_SAT$(\phi, \mathbf{v})$, the call BCP$(\phi, \mathbf{v})$may modify its arguments.

BCP$(\phi, \mathbf{v})$

1. Repeatedly search for unit clauses, and set unassigned literal to required value.
2. If a literal is assigned to a conflicting value, then return false else return true.

Davis-Putnam-Logemann-Loveland Algorithm (1/2)

Input: A propositional formula ϕ on a finite set V and a partial assignment $\mathbf{v}$ on V such that only variables unassigned by $\mathbf{v}$ occur in ϕ.

Output: true if ϕ is satisfiable, false otherwise.

```
DPLL($\phi, \mathbf{v}$) {
  1 if every clause of $\phi$ has a true literal, return true;
  2 if any clause of $\phi$ has all false literals, return false;
  3 if BCP($\phi, \mathbf{v}$) returns false, then return false;
  4 choose an $p \in V$ that is unassigned in $\mathbf{v}$;
  5 assign $p$ to true, that is, let $\mathbf{v}(p) = $ true;
  6 if DPLL($\phi|_p, \mathbf{v}$) returns true, then return true;
  7 assign $p$ to false, that is, let $\mathbf{v}(p) = $ false;
  8 if DPLL($\phi|_{\neg p}, \mathbf{v}$) returns true, then return true;
  9 unassign $\mathbf{v}(p)$; # backtracking takes place here
  10 return false; }
```

Davis-Putnam-Logemann-Loveland Algorithm (2/2)

Example

See `https://en.wikipedia.org/wiki/DPLL_algorithm` for a detailed example.

Remarks

- DPLL($\phi, \mathbf{v}$) terminates and is correct for the same reasons as Naive_SAT($\phi, \mathbf{v}$)
- Many modern SAT solvers are based on DPLL.

References

- `http://yices.csl.sri.com/old/download-yices1-full.html` (The `yices` software)
- `http://www.cs.cornell.edu/gomes/papers/SATSolvers-KR-Handbook.pdf` SAT solvers handbook
- `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem` (SAT)
- `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories` (SMT)
- `https://en.wikipedia.org/wiki/DPLL_algorithm` (DPLL)
- `http://0a.io/boolean-satisfiability-problem-or-sat-in-5-minutes/`

This lecture follows partly a presentation by Hans Zantema (Eindhoven University of Technology), another by Luciano Serafini (Fondazione Bruno Kessler, Trento) and another by David L. Dill (Stanford University).