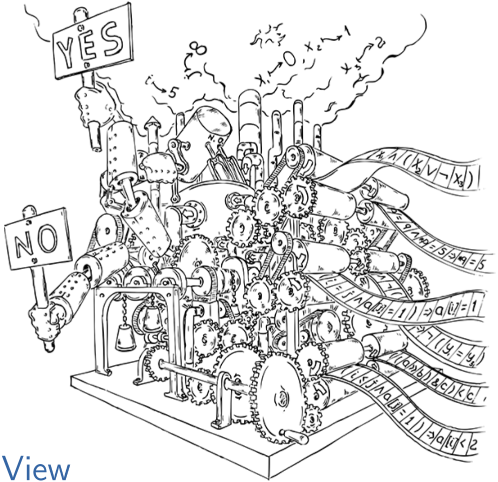
Propositional Encodings

Chapter 11



Decision Procedures

An Algorithmic Point of View

D.Kroening O.Strichman

Revision 1.0

Outline

- Let T be a first-order Σ-theory such that:
  - T is quantifier-free.
  - There exists a decision procedure, denoted DP_T, for the conjunctive fragment of T.

- Example 1:
  - T is equality logic.
  - DP_T is an algorithm based on union-find.

- Example 2:
  - T is disjunctive linear arithmetic.
  - DP_T is the Simplex algorithm.

Input: a conjunction of equalities and disequalities φ.

Example:

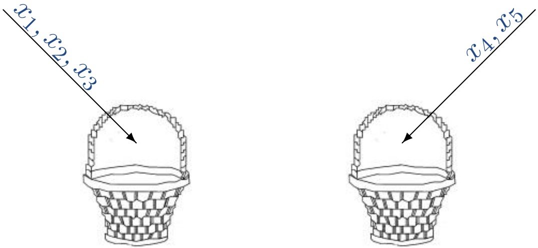
$$\varphi : \quad x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge x_5 \neq x_1$$

Algorithm

1. Define an equivalence class for each variable.
2. For each equality $x = y$, merge the classes of x and y.
3. For each disequality $x \neq y$:
   if x is in the same class as y, return 'UNSAT'.
4. Return 'SAT'.

Can be implemented efficiently with a union-find algorithm.

$$\varphi : \quad x_1 = x_2 \land x_2 = x_3 \land x_4 = x_5 \land x_5 \neq x_1$$



Equivalence class 1 Equivalence class 2

Q: Is there a disequality between members of the same class?

We will now study a framework that combines

- DP_T, and
- a SAT solver,

in various ways, in order to construct a decision procedure for T.

This method is

- modular,
- efficient,
- competitive (all state-of-the-art SMT solvers work this way).

The two main engines in this framework work in tight collaboration:

- The SAT solver chooses those literals that need to be satisfied in order to satisfy the Boolean structure of the formula, and
- The theory solver DP_T checks whether this choice is consistent in T.

Let l be a Σ-literal.

- Denote by $e(l)$ the Boolean encoder of this literal.

Let φ be a Σ-formula,

- Denote by $e(\varphi)$ the Boolean formula resulting from substituting each Σ-literal in φ with its Boolean encoder.

For a Σ-formula φ, the resulting Boolean formula $e(\varphi)$ is called the propositional skeleton of φ.

- Example I: Let $l := x = y$ be a Σ-literal. Then $e(x = y)$, a Boolean variable, is its encoder.

- Example II: Let

$$\varphi := \; x = y \vee x = z$$

be a Σ-formula. Then

$$e(\varphi) := \; e(x = y) \vee e(x = z)$$

is its Boolean encoder.

Let T be equality logic. Given an NNF formula

$$\varphi := \; x = y \land ((y = z \land x \neq z) \lor x = z) \; , \qquad (1)$$

we begin by computing its propositional skeleton:

$$e(\varphi) := \; e(x = y) \land ((e(y = z) \land e(x \neq z)) \lor e(x = z)) \; . \qquad (2)$$

Note that since we are encoding *literals* and not *atoms*, $e(\varphi)$ has no negations and hence is trivially satisfiable.

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := e(\varphi) .$$

As a second step, we pass $\mathcal{B}$ to a SAT solver.

Assume that the SAT solver returns the satisfying assignment

$$\begin{aligned} \alpha := \{ & e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE},\ e(x \neq z) \mapsto \text{TRUE}, \\ & e(x = z) \mapsto \text{FALSE}\} . \end{aligned}$$

- Denote by $\hat{T}h(\alpha)$ the conjunction of the literals corresponding to this assignment.

$$\hat{T}h(\alpha) := \; x = y \wedge y = z \wedge x \neq z \wedge \neg(x = z) \; .$$

- The decision procedure DP_T now has to decide whether $\hat{T}h(\alpha)$ is satisfiable.

  $\hat{T}h(\alpha)$ is not satisfiable, which means that the negation of this formula is a tautology.

Thus $\mathcal{B}$ is conjoined with $e(\neg \hat{T}h(\alpha))$, the Boolean encoding of this tautology:

$$e(\neg \hat{T}h(\alpha)) := \ (\neg e(x = y) \vee \neg e(y = z) \vee \neg e(x \neq z) \vee e(x = z)) \ .$$

- This clause contradicts the current assignment, and hence blocks it from being repeated.
- Such clauses are called **blocking clauses**.
- We denote by t the formula – also called the **lemma** – returned by DP_T (in this example $t := \neg \hat{T}h(\alpha)$).

After the blocking clause has been added, the SAT solver is invoked again and suggests another assignment, for example

$$\alpha' := \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE},\ e(x = z) \mapsto \text{TRUE},\\ e(x \neq z) \mapsto \text{FALSE}\}\ .$$

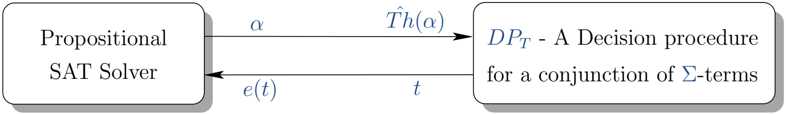
The corresponding Σ-formula

$$\hat{Th}(\alpha') := x = y \land y = z \land x = z \land \neg(x \neq z) \tag{3}$$

is satisfiable, which proves that φ, the original formula, is satisfiable.

Indeed, any assignment that satisfies $\hat{Th}(\alpha')$ also satisfies φ.

Overview



The information flow between the two components of the decision procedure.

There are many improvements to this basic procedure.

We will later consider several of them:

1. Invoke DP_T after partial assignments.
2. Theory propagation: learn propositional constraints based on the theory literals.
    - When the partial assignment is not contradictory.
3. Generalize blocking clauses

Consider the partial assignment

$$\alpha := \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE}\}\ . \tag{4}$$

DP_T receives:

$$\hat{T}h(\alpha) := x = y \wedge y = z\ , \tag{5}$$

... and infers $x = z$.

DP_T now performs Theory Propagation: informs the SAT solver of

$e(x = z) \mapsto \text{TRUE}$ and $e(x \neq z) \mapsto \text{FALSE}$.

Plan

We will now formalize three versions of the algorithm:

1. Simple
2. Incremental
3. DPLL(T)

- $lit(\varphi)$ – the set of literals in a given NNF formula φ.
- $lit_i(\varphi)$ – the i-th distinct literal in φ (assuming some predefined order on the literals).
- α – For a given encoding $e(\varphi)$, denotes an assignment (either full or partial), to the encoders in $e(\varphi)$.

Notation

- $Th(lit_i, \alpha)$ – For an encoder $e(lit_i)$ that is assigned a truth value by α, denotes the corresponding literal:

$$Th(lit_i, \alpha) \doteq \begin{cases} lit_i & \alpha(lit_i) = \text{TRUE} \\ \neg lit_i & \alpha(lit_i) = \text{FALSE}\ . \end{cases} \tag{6}$$

- $Th(\alpha) \doteq \{Th(lit_i, \alpha) \mid e(lit_i) \text{ is assigned by } \alpha\}$
- $\hat{T}h(\alpha)$ – a conjunction over the elements in $Th(\alpha)$.

Let

$$lit_1 = (x = y),\ lit_2 = (y = z),\ lit_3 = (z = w)\ , \tag{7}$$

and let α be a partial assignment such that

$$\alpha := \{e(lit_1) \mapsto \text{FALSE},\ e(lit_2) \mapsto \text{TRUE}\}\ .$$

Then

$$Th(lit_1, \alpha) := \neg(x = y),\ Th(lit_2, \alpha) := (y = z)\ ,$$

and

$$Th(\alpha) := \{\neg(x = y), (y = z)\}\ .$$

Conjoining these terms gives us

$$\hat{Th}(\alpha) := \neg(x = y) \wedge (y = z)\ .$$

Notation

- T – a Σ-theory.
- DP_T a decision procedure for the conjunctive fragment of T.
- Let DEDUCTION be a procedure based on DP_T, which receives a conjunction of Σ-literals as input, and
  - decides whether it is satisfiable, and,
  - if the answer is negative, returns constraints over these literals.

function LAZY-BASIC(φ)
    $\mathcal{B} := e(\varphi)$;
    while (TRUE) **do**
        $\langle \alpha, res \rangle :=$ SAT-SOLVER($\mathcal{B}$);
        if $res =$ "Unsatisfiable" **then return** "Unsatisfiable";
        else
            $\langle t, res \rangle :=$ DEDUCTION($\hat{Th}(\alpha)$);
            if $res =$ "Satisfiable" **then return** "Satisfiable";
            $\mathcal{B} := \mathcal{B} \wedge e(t)$;

The clause t that is returned by DEDUCTION:

... should not be too strong.

Strategies:

1. $\varphi \rightarrow t$.
   - ... DEDUCTION needs to deal with φ rather than $Atoms(\varphi)$.
2. t is T-valid

   Example:

   $$x = y \wedge y = z \longrightarrow x = z$$

This guarantees soundness.

The clause t that is returned by DEDUCTION:

... should not be too weak

- it should at least block α.

This guarantees termination (?)

The clause t that is returned by DEDUCTION:

... should not lead to divergence

- $Atoms(t) \subseteq Atoms(\varphi)$, or
- $Atoms(t)$ is finite.

  Example:

  $T =$ equality logic.
  $Atoms(t) = \{x_i = x_j \mid x_i, x_j \in var(\varphi)\}$
  Some of the predicates in t do not appear in φ.

This guarantees termination.

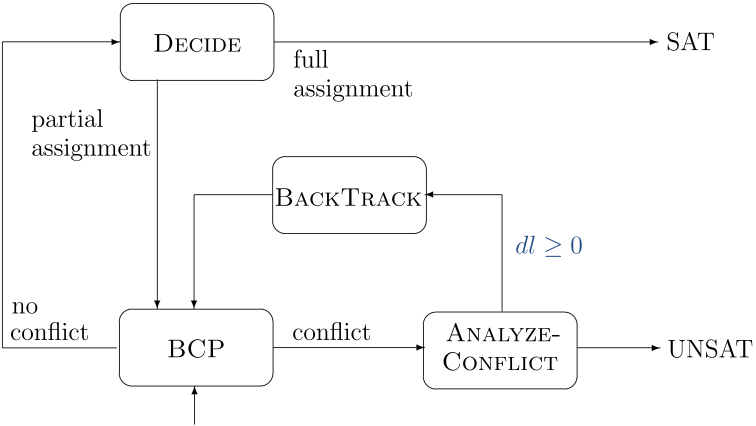
- Let $\mathcal{B}^i$ be the formula $\mathcal{B}$ in the i-th iteration of the loop.
- The constraint $\mathcal{B}^{i+1}$ is strictly stronger than $\mathcal{B}^i$ for all $i \geq 1$, because clauses are added but not removed between iterations.
- As a result, any conflict clause that is learned while solving $\mathcal{B}^i$ can be reused when solving $\mathcal{B}^j$ for $i < j$.
- This is a special case of incremental satisfiability.

- Hence, invoking an incremental SAT solver in line 4 can increase the efficiency of the algorithm.

- A better option is to integrate DEDUCTION into the DPLL-SAT algorithm, as shown in the following algorithm.

- This algorithm uses a procedure ADDCLAUSES, which adds new clauses to the current set of clauses at run time.

- Before seeing this algorithm let us first recall DPLL...


Decide
SAT
full
assignment
partial
assignment
BackTrack
$dl \geq 0$
no
conflict
BCP
conflict
Analyze-
Conflict
UNSAT

```
function DPLL
    if BCP() = "conflict" then return "Unsatisfiable";
    while (TRUE) do
        if ¬DECIDE() then return "Satisfiable";
        while (BCP() = "conflict") do
            bcktk-level := ANALYZE-CONFLICT();
            if bcktk-level < 0 then return "Unsatisfiable";
            else BackTrack(bcktk-level);
```

2. Integration into DPLL

```
function Lazy-DPLL
    AddClauses(cnf(e(φ)));
    if BCP() = "conflict" then return "Unsatisfiable";
    while (TRUE) do
        if ¬Decide() then                                ▷ Full assignment
            ⟨t, res⟩ := Deduction(T̂h(α));
            if res = "Satisfiable" then return "Satisfiable";
            AddClauses(e(t));
        while (BCP() = "conflict") do
            backtrack-level := Analyze-Conflict();
            if backtrack-level < 0 then return "Unsatisfiable";
            else BackTrack(backtrack-level);
```

- Let $\{x_1 \geq 10, x_1 < 0\} \subset Atoms(\varphi)$.

- Assume currently $e(x_1 \geq 10) = e(x_1 < 0) = \text{TRUE}$.

- Now any call to DEDUCTION results in a contradiction.

- So far: DEDUCTION after finding a full assignment α.
    - Time taken to complete the assignment is wasted.
    - The refutation of α may be due to other reasons.
        - additional assignments that include
          $e(x_1 \geq 10) = e(x_1 < 0) = \text{TRUE}$
          are not ruled out.

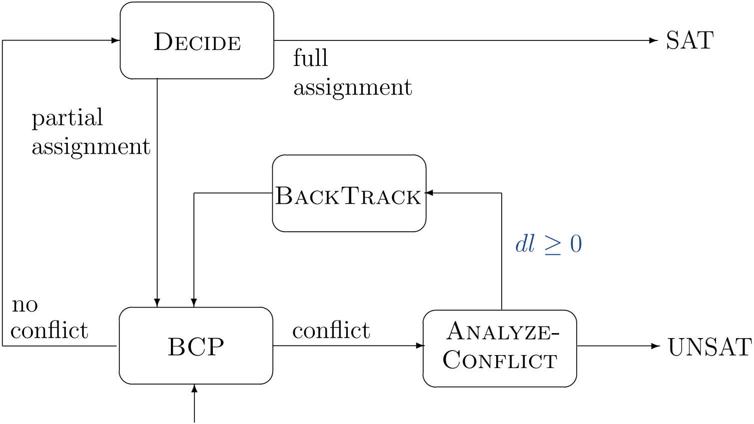
Solution: Early call to Deduction.

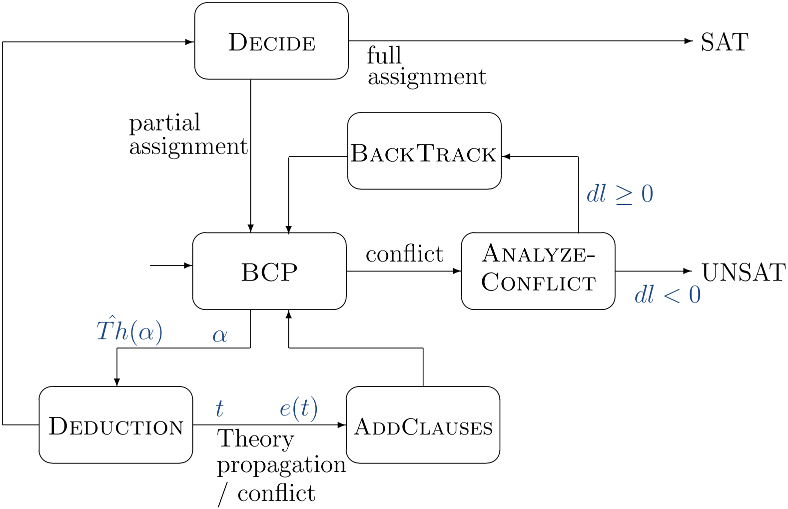
Early call to DEDUCTION can serve two purposes:

1. Contradictory partial assignments are ruled out early.
2. Allows theory propagation.
   - When $e(x_1 \geq 10) = \text{TRUE}$, infer $e(x_1 < 0) = \text{FALSE}$.

This brings us to the next version of the algorithm, called DPLL(T).

3. Reminder: DPLL




DECIDE
SAT
full assignment
partial assignment
BACKTRACK
$dl \geq 0$
BCP
conflict
ANALYZE-CONFLICT
UNSAT
$dl < 0$
$\hat{Th}(\alpha)$
α
DEDUCTION
t
$e(t)$
ADDCLAUSES
Theory propagation / conflict

```
function DPLL(T)
    AddClauses(cnf(e(φ)));
    if BCP() = "conflict" then return "Unsatisfiable";
    while (TRUE) do
        if ¬Decide() then return "Satisfiable";
        repeat
            while (BCP() = "conflict") do
                btrk-level := Analyze-Conflict();
                if btrk-level < 0 then return "Unsatisfiable";
                else BackTrack(btrk-level);
            ⟨t, res⟩:=Deduction(T̂h(α));
            AddClauses(e(t));
        until t ≡ TRUE
```

When $\hat{Th}(\alpha)$ is satisfiable, it is required that:

- $e(t)$ is an asserting clause under α.
  - BCP will now assign a value to an encoder.

- If DEDUCTION cannot find such an asserting clause t, then $t = e(t) = \text{TRUE}$.
  - This can happen when, e.g., all the encoders are assigned.

Various ways to perform theory propagation:

- After every decision / after every assignment
- Partial / Exhaustive theory propagation
  - Exhaustive = propagate everything that is implied.
- Refer only to existing predicates / add auxiliary ones.

Exhaustive theory propagation after each assignment: what does this mean ?

That's right, no possible conflicts on the theory side.

How to check whether a predicate p is implied by $\hat{Th}(\alpha)$?

- Plunging – is $\hat{Th}(\alpha) \wedge \neg p$ satisfiable ?

- Theory-specific propagation.

  Example:

  $T =$ equality logic

  Build the equality graph corresponding to $Th(\alpha)$.

  Infer equalities/disequalities from the graph.

- Theory propagation matters for efficiency, not correctness.

- How much propagation is cost-effective is a subject for research, and depends on T.

3. Theory Propagation – How?

- Normally theory propagation is done by transferring clauses.

- Inefficient
  - Less than 0.5% are actually used.

- Instead – add implied literals directly to the implication stack.
  - This causes a problem in ANALYZE-CONFLICT()
  - Can you see what problem ?

- The problem: $\textsc{Analyze-conflict}()$ requires an antecedent clause for each implication, in order to compute the conflict clause and backtrack level.

- ... but now there are implications without antecedents.

- Solution – DP_T should be able to explain an implication post-mortem, in the form of a clause.

- When $\hat{T}h(\alpha)$ is unsatisfiable, the 'lemma' t blocks α.

- We want to make t stronger.

- How?
  - Analyze the reason for the unsatisfiability.
  - Build the lemma accordingly.

3. Strong Lemmas – An Example