Foundations of Artificial Intelligence

8. Satisfiability and Model Construction

Davis-Putnam-Logemann-Loveland Procedure, Phase Transitions, GSAT

Wolfram Burgard, Maren Bennewitz, and Marco Ragni

Albert-Ludwigs-Universität Freiburg
Contents

1 Motivation

2 Davis-Putnam-Logemann-Loveland (DPLL) Procedure

3 “Average” complexity of the satisfiability problem

4 GSAT: Greedy SAT Procedure
Motivation

- Usually:
  - Given: A logical theory (set of propositions)
  - Question: Does a proposition logically follow from this theory?
  - Reduction to unsatisfiability, which is coNP-complete (complementary to NP problems)

- Sometimes:
  - Given: A logical theory
  - Wanted: Model of the theory
  - Example: Configurations that fulfill the constraints given in the theory
  - Can be “easier” because it is enough to find one model
The DPLL Procedure

DPLL Function

Given a set of clauses $\Delta$ defined over a set of variables $\Sigma$, return “satisfiable” if $\Delta$ is satisfiable. Otherwise return “unsatisfiable”.

1. If $\Delta = \emptyset$ return “satisfiable”

2. If $\square \in \Delta$ return “unsatisfiable”

3. **Unit-propagation Rule:** If $\Delta$ contains a unit-clause $C$, assign a truth-value to the variable in $C$ that satisfies $C$, simplify $\Delta$ to $\Delta'$ and return $\text{DPLL}(\Delta')$.

4. **Splitting Rule:** Select from $\Sigma$ a variable $v$ which has not been assigned a truth-value. Assign one truth value $t$ to it, simplify $\Delta$ to $\Delta'$ and call $\text{DPLL}(\Delta')$
   
   a. If the call returns “satisfiable”, then return “satisfiable”.
   
   b. Otherwise assign the other truth-value to $v$ in $\Delta$, simplify to $\Delta''$ and return $\text{DPLL}(\Delta'')$. 

(University of Freiburg)
Example (1)

\[ \Delta = \{\{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\} \]
\[ \Delta = \{ \{ a, b, \neg c \}, \{ \neg a, \neg b \}, \{ c \}, \{ a, \neg b \} \} \]

1. **Unit-propagation rule:** \( c \mapsto T \)
\[ \Delta = \{\{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\} \]

1. Unit-propagation rule: \( c \mapsto T \)
   \[ \{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\} \]
Example (1)

\[ \Delta = \{\{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\} \]

1. Unit-propagation rule: \( c \mapsto T \)
   \( \{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\} \)

2. Splitting rule:
Example (1)

\[ \Delta = \{ \{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\} \]

1. Unit-propagation rule: \(c \mapsto T\)
   \[\{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}\]

2. Splitting rule:

2a. \(a \mapsto F\)
   \[\{\{b\}, \{\neg b\}\}\]
Example (1)

\[ \Delta = \{ \{ a, b, \neg c \}, \{ \neg a, \neg b \}, \{ c \}, \{ a, \neg b \} \} \]

1. Unit-propagation rule: \( c \mapsto T \)
   \[ \{ \{ a, b \}, \{ \neg a, \neg b \}, \{ a, \neg b \} \} \]

2. Splitting rule:

2a. \( a \mapsto F \)
   \[ \{ \{ b \}, \{ \neg b \} \} \]

3a. Unit-propagation rule: \( b \mapsto T \)
   \[ \{ \Box \} \]
\[\Delta = \{\{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\}\]

1. Unit-propagation rule: \(c \mapsto T\)
   \[\{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}\]

2. Splitting rule:
   2a. \(a \mapsto F\)
       \[\{\{b\}, \{\neg b\}\}\]
   2b. \(a \mapsto T\)
       \[\{\{\neg b\}\}\]

3a. Unit-propagation rule: \(b \mapsto T\)
    \[\{\Box\}\]
Example (1)

\[ \Delta = \{ \{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\} \]

1. Unit-propagation rule: \( c \mapsto T \)
   \( \{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\} \)

2. Splitting rule:

2a. \( a \mapsto F \)
    \( \{\{b\}, \{\neg b\}\} \)

2b. \( a \mapsto T \)
    \( \{\{\neg b\}\} \)

3a. Unit-propagation rule: \( b \mapsto T \)
    \( \{\square\} \)

3b. Unit-propagation rule: \( b \mapsto F \)
    \( \{\} \)
Example (1)

\[ \Delta = \{ \{ a, b, \neg c \}, \{ \neg a, \neg b \}, \{ c \}, \{ a, \neg b \} \} \]

1. Unit-propagation rule: \( c \mapsto T \)
   \[ \{ \{ a, b \}, \{ \neg a, \neg b \}, \{ a, \neg b \} \} \]

2. Splitting rule:

2a. \( a \mapsto F \)
   \[ \{ \{ b \}, \{ \neg b \} \} \]

3a. Unit-propagation rule: \( b \mapsto T \)
   \[ \{ \Box \} \]

2b. \( a \mapsto T \)
   \[ \{ \{ \neg b \} \} \]

3b. Unit-propagation rule: \( b \mapsto F \)
   \[ \{ \} \]
Example (2)

\[ \Delta = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\} \]
\[ \Delta = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\} \]

1. Unit-propagation rule: \( d \mapsto T \)
Example (2)

\[ \Delta = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\} \]

1. Unit-propagation rule: \( d \mapsto T \)
   \( \{\{a, \neg b, \neg c\}, \{b\}, \{c\}\} \)
\[ \Delta = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\} \]

1. Unit-propagation rule: \( d \mapsto T \)
   \[ \{\{a, \neg b, \neg c\}, \{b\}, \{c\}\} \]

2. Unit-propagation rule: \( b \mapsto T \)
   \[ \{\{a, \neg c\}, \{c\}\} \]
\[ \Delta = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\} \]

1. Unit-propagation rule: \( d \mapsto T \)
   \( \{\{a, \neg b, \neg c\}, \{b\}, \{c\}\} \)

2. Unit-propagation rule: \( b \mapsto T \)
   \( \{\{a, \neg c\}, \{c\}\} \)

3. Unit-propagation rule: \( c \mapsto T \)
   \( \{\{a\}\} \)
Example (2)

\[ \Delta = \{ \{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\} \]

1. Unit-propagation rule: \(d \mapsto T\)
   \(\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}\)

2. Unit-propagation rule: \(b \mapsto T\)
   \(\{\{a, \neg c\}, \{c\}\}\)

3. Unit-propagation rule: \(c \mapsto T\)
   \(\{\{a\}\}\)

4. Unit-propagation rule: \(a \mapsto T\)
   \(\{\}\)
Example (2)

\[ \Delta = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\} \]

1. Unit-propagation rule: \(d \mapsto T\)
   \[\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}\]

2. Unit-propagation rule: \(b \mapsto T\)
   \[\{\{a, \neg c\}, \{c\}\}\]

3. Unit-propagation rule: \(c \mapsto T\)
   \[\{\{a\}\}\]

4. Unit-propagation rule: \(a \mapsto T\)
   \[\{}\]
Properties of DPLL

- DPLL is complete, correct, and guaranteed to terminate.
- DPLL constructs a model, if one exists.
- In general, DPLL requires exponential time (splitting rule!)
- DPLL is polynomial on Horn clauses, i.e., clauses with at most one positive literal

\[ \neg A_1 \lor \ldots \lor \neg A_n \lor B \iff \bigwedge_{i} A_i \Rightarrow B \]

→ Heuristics are needed to determine which variable should be instantiated next and which value should be used.

→ In all SAT competitions so far, DPLL-based procedures have shown the best performance.
DPLL on Horn Clauses (1)

Note:

1. The simplifications in DPLL on Horn clauses always generate Horn clauses

2. A set of Horn clauses without unit clauses is satisfiable
   - All clauses have at least one negative literal
   - Assign *false* to all variables

3. If the first sequence of applications of the unit propagation rule in DPLL does not lead to the empty clause, a set of Horn clauses without unit clauses is generated (which is satisfiable according to 2.)
4. Although a set of Horn clauses without a unit clause is satisfiable, DPLL may not immediately recognize it.
   
   a. If DPLL assigns \textit{false} to a variable, this cannot lead to an unsatisfiable set and after a sequence of unit propagations we are in \textit{the same situation as in 4}.
   
   b. If DPLL assigns \textit{true}, then we may get an empty clause - perhaps after unit propagation (and have to backtrack) - or the set is still satisfiable and we are in \textit{the same situation as in 4}.
In summary:

1. DPLL executes a sequence of unit propagation steps resulting in
   - an empty clause or
   - a set of Horn clauses without a unit clause, which is satisfiable

2. In the latter case, DPLL proceeds by choosing for one variable:
   - `false`, which does not change the satisfiability
   - `true`, which either
     - leads to an immediate contradiction (after unit propagation) and backtracking or
     - does not change satisfiability

→ Run time is *polynomial* in the number of variables.
We know that SAT is NP-complete, i.e., in the worst case, it takes exponential time.

This is clearly also true for the DPLL-procedure.  
→ Couldn’t we do better in the average case?

For CNF-formulae in which the probability for a positive appearance, negative appearance and non-appearance in a clause is 1/3, DPLL needs on average quadratic time (Goldberg 79)!  
→ The probability that these formulae are satisfiable is, however, very high.
Conversely, we can, of course, try to identify hard to solve problem instances.

Cheeseman et al. (IJCAI-91) came up with the following plausible conjecture:

All NP-complete problems have at least one order parameter and the hard to solve problems are around a critical value of this order parameter. This critical value (a phase transition) separates one region from another, such as over-constrained and under-constrained regions of the problem space.

Confirmation for graph coloring and Hamilton path ... later also for other NP-complete problems.
Phase Transitions with 3-SAT

**Constant clause length model** (Mitchell et al., AAAI-92):
Clause length $k$ is given. Choose variables for every clause $k$ and use the complement with probability 0.5 for each variable.

**Phase transition** for 3-SAT with a clause/variable ratio of approx. 4.3:
Empirical Difficulty

The Davis-Putnam (DPLL) Procedure shows extreme runtime peaks at the phase transition:

Note: Hard instances can exist even in the regions of the more easily satisfiable/unsatisfiable instances!
Notes on the Phase Transition

- When the probability of a solution is close to 1 (under-constrained), there are many solutions, and the first search path of a backtracking search is usually successful.

- If the probability of a solution is close to 0 (over-constrained), this fact can usually be determined early in the search.

- In the phase transition stage, there are many near successes ("close, but no cigar")

  → (limited) possibility of predicting the difficulty of finding a solution based on the parameters

  → (search intensive) benchmark problems are located in the phase region (but they have a special structure)
In many cases, we are interested in finding a satisfying assignment of variables (example CSP), and we can sacrifice completeness if we can “solve” much large instances this way.

Standard process for optimization problems: **Local Search**

- Based on a (random) configuration
- Through local modifications, we hope to produce better configurations
  - Main problem: **local maxima**
Dealing with Local Maxima

As a measure of the value of a configuration in a logical problem, we could use the number of satisfied constraints/clauses.

But local search seems inappropriate, considering we want to find a global maximum (all constraints/clauses satisfied).

By restarting and/or injecting noise, we can often escape local maxima.

Actually: Local search performs very well for finding satisfying assignments of CNF formulae (even without injecting noise).
**Procedure GSAT**

**INPUT**: a set of clauses $\alpha$, $\text{Max-Flips}$, and $\text{Max-Tries}$

**OUTPUT**: a satisfying truth assignment of $\alpha$, if found

```
begin
  for $i := 1$ to $\text{Max-Tries}$
    $T :=$ a randomly-generated truth assignment
    for $j := 1$ to $\text{Max-Flips}$
      if $T$ satisfies $\alpha$ then return $T$
      $v :=$ a propositional variable such that a change in its
      truth assignment gives the largest increase in
      the number of clauses of $\alpha$ that are satisfied by $T$
      $T := T$ with the truth assignment of $v$ reversed
    end for
  end for
return “no satisfying assignment found”
end
```
The Search Behavior of GSAT

- In contrast to normal local search methods, we must also allow sideways movements!
- Most time is spent searching on plateaus.

![Graph showing the behavior of GSAT over iterations.](graph.png)
State of the Art

- SAT competitions since beginning of the 90s
- Current SAT competitions ([http://www.satcompetition.org/](http://www.satcompetition.org/)):
  - In 2010:
    - Largest “industrial” instances: $> 1,000,000$ literals
- Complete solvers are as good as randomized ones on handcrafted and industrial problem
Concluding Remarks

- **DPLL-based SAT solvers prevail:**
  - Very efficient implementation techniques
  - Good branching heuristics
  - Clause learning

- **Incomplete randomized SAT-solvers**
  - are good (in particular on random instances)
  - but there is no dramatic increase in size of what they can solve
  - parameters are difficult to adjust