Foundations of Artificial Intelligence

Lorenzo Bozzoni

December 8, 2023

Contents

0.1 Model checking

Two algorithms:

1. Reasoning with truth tables
2. Exploting satisfiability (DPLL)

0.1.1 Reasoning with truth tables

Reasoning with truth tables is a form of semantic reasoning, in the sense that it directly exploits the definition of entailment: $\alpha \vDash \beta$ holds when β is true in every model that makes α true In PL, a model is an assignment of truth values (1 or 0, true or false, T or $\bot$) to every propositional symbol that appears in α or β (or both) Therefore, with n symbols we have $2n$ different models, which correspond to the rows of the truth table. For every model (row), we compute the truth values of α and β (by recursively computing the truth values of all the subsentences of α and β). Then we have that $\alpha \vDash \beta$ if, and only if, every model (row) that assigns 1 to α also assigns 1 to β.

Example:
Let us consider:

- When it rains and it is windy Alice, wears a raincoat
- It rains, but Alice does not wear a raincoat
- Therefore, it is not windy

Propositional representation:

- $R \wedge W \rightarrow RCA$
- $R \wedge \neg RCA$
- $\neg W$

We want to prove $a, b \vDash c$. Here is the truth table:

R	W	RCA	R ∧ W	R ∧ W ⇒ RCA	¬RCA	R ∧ ¬RCA	¬W
0	0	0	0	1	1	0	1
0	0	1	0	1	0	0	1
0	1	0	0	1	1	0	0
0	1	1	0	1	0	0	0
1	0	0	0	1	1	1	1
1	0	1	0	1	0	0	1
1	1	0	1	0	1	1	0
1	1	1	1	1	0	0	0

This reasoning procedure is sound and complete. It always terminates, making reasoning in PL decidable. However, it is inefficient when many propositional symbols are involved, because it has to compute a table of size $2^n \times M$, where n is the number of propositional symbols and M is the number of subsentences that appear in the premises and the conclusion. Another drawback of reasoning with truth tables is that it is very unnatural, in the sense that the reasoning process is very far from the most usual forms of human reasoning. This may be problematic in those applications in which an artificial agent must be able to justify the conclusions of its reasoning processes in a way that is easily understandable for a human being.

Certain applications of PL require an agent to establish whether a set of sentences α is or is not satisfiable (i.e., whether there is an assignment of truth values to the symbols of α that makes α true).

Note: in general, set α may be finite or infinite; if it is finite, i.e., $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$, then α set is logically equivalent to the conjunction of its elements, $\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n$. In standard AI applications, α is always finite (although it may be very large): therefore, when it is convenient we can regard α as a single sentence with no loss of generality. The problem of establishing the satisfiability of a set of propositional sentences is known as SAT. Many interesting problems, including establishing propositional entailment, can be reduced to SAT. A (rather inefficient) solution of SAT is given by truth tables: α is satisfiable if, and only if, it has truth value 1 in at least one row of its truth table. A more efficient method is provided by the DPLL algorithm.

0.1.2 DPLL

From Davis-Putnam-Logemann-Loveland. Establish whether a set of sentences α is or is not satisfiable. Preprocessing: convert every sentence in **CNF (Conjunctive Normal Form)**. Body of the procedure: from an empty assignment, incrementally try to build a model of α (i.e., an assignment of truth values to the propositional symbols)

- if a model is built, α is satisfiable
- if the algorithm terminates without being able to build a model, α is unsatisfiable

CNF (Conjunctive Normal Form) represents a sentence as a conjunction of clauses, where a clause is a disjunction of literals and a literal is either a propositional symbol or the negation of a symbol. Every sentence of PL can be transformed in an equivalent sentence in CNF:

- $A \to B \vee C$ becomes $\neg A \vee B \vee C$
- $C \to \neg D$ becomes $\neg C \vee \neg D$

A CNF sentence is often considered as a set of clauses (in logical conjunction), which are in turn considered as sets of literals (in logical disjunction): $(\neg A \vee B \vee C) \wedge (\neg C \vee \neg D)$ becomes $\{\{\neg A, B, C\}, \{\neg C, \neg D\}\}$.

Unit clause: clause with only one literal; e.g. $\neg C$. Two literals are **complementary** if they refer to the same propositional symbol but have different "signs", like $\neg C$ and C.

Conversion to CNF, consider $A \leftrightarrow (B \vee C)$:

1. Eliminate $\leftrightarrow$, replacing $\alpha \leftrightarrow \beta$ with $(\alpha \to \beta) \vee (\beta \to \alpha)$: $(A \to (B \vee C)) \wedge ((B \vee C) \to A)$
2. Eliminate $\to$, replacing $\alpha \to \beta$ with $\neg\alpha \vee \beta$: $(\neg A \vee B \vee C) \wedge (\neg(B \vee C) \vee A)$
3. Move $\neg$ inwards using deMorgan's rule ($\neg(\alpha \wedge \beta)$) is equivalent to $(\neg\alpha \vee \neg\beta)$ and ($\neg(\alpha \vee \beta)$) is equivalent to $(\neg\alpha \wedge \neg\beta)$: $(\neg A \vee B \vee C) \wedge ((\neg B \wedge \neg C) \vee A)$
4. Apply distributivity law and flatten: $(\neg A \vee B \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A)$

Essentially a backtracking (depth-first) search over models with some extras:

- Early termination: stop when
  - all clauses are satisfied $(A \vee B) \wedge (A \vee \neg C)$ is satisfied by $\{A = 1\}$
  - any clause is falsified: $(A \vee B) \wedge (A \vee \neg C)$ is falsified by $\{A = 0, B = 0\}$
- Pure literals heuristic: if all occurrences of a symbol in yet-unsatified clauses have the same sign, then give the symbol that value:
  - in $(A \vee B) \wedge (A \vee \neg C) \wedge (C \vee \neg B)$. A is pure and positive, so set it to true
  - if there is a model, then there is a model also with $\{\dots, A = 1, \dots\}$
- Unit clause heuristic: if a clause has a single literal, set the corresponding symbol to satisfy clause:
  - in $(A \vee B) \wedge \neg C$. $\neg C$ must be set to true $\{C = 0\}$

Pure literals and unit clauses often propagate to new pure literals and unit clauses: $(\neg A \vee B \vee C) \wedge (A \vee \neg C)$ with pure literal $\{B = 1\}$ becomes $(A \vee \neg C)$, then A is pure literal ...

The actual DPLL algorithm:

```
function DPLL(clauses, symbols, model) returns true or false
if every clause in clauses is true in model then return true
if some clause in clauses is false in model then return false
P, value <- FIND-PURE-SYMBOL(symbols, clauses, model)
if P is non-null then return DPLL(clauses, symbols - P, model U {P=value})
P, value ←FIND-UNIT-CLAUSE(clauses, model)
if P is non-null then return DPLL(clauses, symbols - P, model U {P=value})
P ← First(symbols); rest ← Rest(symbols)
return or(DPLL(clauses, rest, model U {P=true}),
DPLL(clauses, rest, model U {P=false}))
```

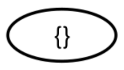
Its python implementation:

```python
def dpll(clauses, symbols, model, branching_heuristic=no_branching_heuristic):
    """See if the clauses are true in a partial model."""
    unknown_clauses = []  # clauses with an unknown truth value
    for c in clauses:
        val = pl_true(c, model)
        if val is False:
            return False
        if val is None:
            unknown_clauses.append(c)
    if not unknown_clauses:
        return model
    P, value = find_pure_symbol(symbols, unknown_clauses)
    if P:
        return dpll(clauses, remove_all(P, symbols), extend(model, P, value), branching_heuristic)
    P, value = find_unit_clause(clauses, model)
    if P:
        return dpll(clauses, remove_all(P, symbols), extend(model, P, value), branching_heuristic)
    P, value = branching_heuristic(symbols, unknown_clauses)
    return (dpll(clauses, remove_all(P, symbols), extend(model, P, value), branching_heuristic) or
            dpll(clauses, remove_all(P, symbols), extend(model, P, not value), branching_heuristic))
```

Example of DPLL: check if the set of sentences $\alpha = \{A \rightarrow B \vee C, \neg C, A \wedge B \rightarrow D, C \rightarrow \neg D\}$ is satisfiable. Convert in CNF:

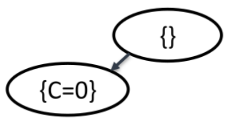
- $A \rightarrow B \vee C$ becomes $\neg A \vee B \vee C$
- $\neg C$ remains $\neg C$
- $A \wedge B \rightarrow D$ becomes $\neg A \vee \neg B \vee D$
- $C \rightarrow \neg D$ becomes $\neg C \vee \neg D$

¬A ∨ B ∨ C ¬C ¬A ∨ ¬B ∨ D ¬C ∨ ¬D

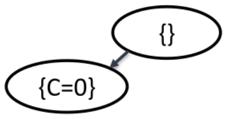


¬A ∨ B ∨ C ¬C ¬A ∨ ¬B ∨ D ¬C ∨ ¬D unit clause ¬C

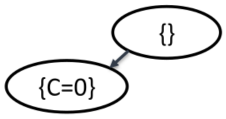
¬C



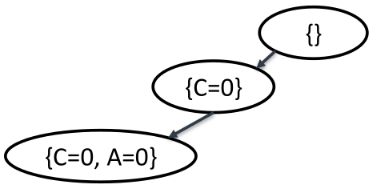
	¬A ∨ B ∨ C	¬C	¬A ∨ ¬B ∨ D	¬C ∨ ¬D	unit clause ¬C
¬C					



	¬A ∨ B ∨ C	¬C	¬A ∨ ¬B ∨ D	¬C ∨ ¬D	unit clause ¬C
¬C	¬A ∨ B	T	¬A ∨ ¬B ∨ D	T	



	¬A ∨ B ∨ C	¬C	¬A ∨ ¬B ∨ D	¬C ∨ ¬D	unit clause ¬C
¬C	¬A ∨ B	T	¬A ∨ ¬B ∨ D	T	pure literal ¬A
¬A					



	¬A ∨ B ∨ C	¬C	¬A ∨ ¬B ∨ D	¬C ∨ ¬D	unit clause ¬C
¬C	¬A ∨ B	T	¬A ∨ ¬B ∨ D	T	pure literal ¬A
¬A	T	T	T	T	early termination

Conclusion: the set of sentences α is satisfiable (because it is satisfied by any complete assignment extending $\{\neg C = 1, \neg A = 1\}$).

Another example of DPLL: is the set of sentences $\alpha = \{A \vee B, A \vee \neg B, \neg A \vee B\}$ satisfiable?

	A ∨ B	A ∨ ¬B	¬A ∨B	split on A (guess A)
A	T	T	B	unit clause B
A, B	T	T	T	termination

At least one branch (with A true) is satisfiable: the set of sentences α is satisfiable If the branch with A true was unsatiasfiable, backtrack and try with the branch with A false Split on several variables, in general What if we would like to prove unsatisfiability? All branches must be unsatisfiable.

The problem of establishing propositional entailment can be reduced to a SAT problem, because $\alpha \vDash \beta$ holds

- if, and only if, every models that satisfies α also satisfies β
- equivalently if, and only if, no model satisfies both α and $\neg\beta$
- equivalently if, and only if, $\alpha \vee \neg\beta$ is unsatisfiable (by refutation or contradiction)

Consider again the previous example in which we want to prove $\{a, b\} \vDash c$.

- $R \wedge W \rightarrow RCA$
- $R \wedge \neg RCA$
- $\neg W$

	¬R ∨ ¬W ∨ RCA	R	¬RCA	W	

	¬R ∨ ¬W ∨ RCA	R	¬RCA	W	unit clause R
R					

	¬R ∨ ¬W ∨ RCA	R	¬RCA	W	unit clause R
R	¬W ∨ RCA	T	¬RCA	W	

	¬R ∨ ¬W ∨ RCA	R	¬RCA	W	unit clause R
R	¬W ∨ RCA	T	¬RCA	W	unit clause ¬RCA
¬RCA					

	¬R ∨ ¬W ∨ RCA	R	¬RCA	W	unit clause R
R	¬W ∨ RCA	T	¬RCA	W	unit clause ¬RCA
¬RCA	¬W	T	T	W	

	¬R ∨ ¬W ∨ RCA	R	¬RCA	W	unit clause R
R	¬W ∨ RCA	T	¬RCA	W	unit clause ¬RCA
¬RCA	¬W	T	T	W	unit clause ¬W
¬W					

	¬R ∨ ¬W ∨ RCA	R	¬RCA	W	unit clause R
R	¬W ∨ RCA	T	¬RCA	W	unit clause ¬RCA
¬RCA	¬W	T	T	W	unit clause ¬W
¬W	T	T	T	⊥	

Conclusion: the set $\{a, b, \neg c\}$ is unsatiasfiable, therefore $\{a, b\} \vDash c$ holds.

0.1.3 Efficiency

Naive implementation of DPLL: solve 100 variables. Improvements:

1. Variable and value ordering (from CSPs)
2. Divide and conquer
3. Caching unsolvable subcases as extra clauses to avoid redoing them

4. Cool indexing and incremental recomputation tricks so that every step of the DPLL algorithm is efficient (typically $O(1)$)
    - Index of clauses in which each variable appears $+ve/-ve$
    - Keep track of number of satisfied clauses, update when variables are assigned
    - Keep track of number of ramaining literals in each clause
5. Real implementation of DPLL: solve 10000000 variables

SAT solvers in practice:

- Circuit verification: does this VLSI circuit compute the right answer?
- Software verification: does this program compute the right answer?
- Software synthesis: what program computes the right answer?
- Protocol verification: can this security protocol be broken?
- Protocol synthesis: what protocol is secure for this task?
- Planning next classes

On the slides there are some examples and exercises.