Review

• Goals and Organization
• History and Motivation
• Introduction to OCaml

Outline

• Propositional Logic
• Truth Table Method
• Davis Putnam (DP)
• Davis Putnam Logemann Loveland (DPLL)
• More OCaml
• Stålmarck’s Method

Review of Propositional Logic: Syntax

We use a formal inductive definition to define the set of well-formed formulas in propositional logic.

- $U$ = the set of all expressions.
- $B$ = the set of expressions consisting of a single propositional symbol.
- $F$ = the set of formula-building operations:
  - $E_\neg(\alpha) = (\neg\alpha)$
  - $E_\land(\alpha, \beta) = (\alpha \land \beta)$
  - $E_\lor(\alpha, \beta) = (\alpha \lor \beta)$
  - $E_\rightarrow(\alpha, \beta) = (\alpha \rightarrow \beta)$
  - $E_\leftrightarrow(\alpha, \beta) = (\alpha \leftrightarrow \beta)$

The set of well-formed formulas is the set of all expressions generated by $F$ from $B$. 

Outline

- Propositional Logic
- Truth Table Method
- Davis Putnam (DP)
- Davis Putnam Logemann Loveland (DPLL)
- More OCaml
- Stålmarck’s Method

Sources:
Review of Propositional Logic: Semantics

Intuitively, given a wff $\alpha$ and a value (either true or false) for each propositional symbol in $\alpha$, we can determine the value of $\alpha$.

Let $v$ be a function from $B$ to $\{0, 1\}$, where 0 represents false and 1 represents true. Recall that in the inductive definition of wffs, $B$ contains the propositional symbols.

Now, we define $\tau$, a function from $W$ to $\{0, 1\}$ as follows

- For each propositional symbol $A_i$, $\tau(A_i) = v(A_i)$.
- $\tau(\neg \alpha) = 1 - \tau(\alpha)$
- $\tau(\land \alpha, \beta) = \min(\tau(\alpha), \tau(\beta))$
- $\tau(\lor \alpha, \beta) = \max(\tau(\alpha), \tau(\beta))$
- $\tau(\rightarrow \alpha, \beta) = \max(1 - \tau(\alpha), \tau(\beta))$
- $\tau(\leftrightarrow \alpha, \beta) = 1 - |\tau(\alpha) - \tau(\beta)|$

Definitions

If $\alpha$ is a wff, then a truth assignment $v$ satisfies $\alpha$ if $\tau(\alpha) = 1$.

A wff $\alpha$ is satisfiable if there exists some truth assignment $v$ which satisfies $\alpha$.

Suppose $\Sigma$ is a set of wffs. Then $\Sigma$ tautologically implies $\alpha$, $\Sigma \models \alpha$, if every truth assignment which satisfies each formula in $\Sigma$ also satisfies $\alpha$.

Particular cases:

- If $\emptyset \models \alpha$, then we say $\alpha$ is a tautology or $\alpha$ is valid and write $\models \alpha$.
- If $\Sigma$ is unsatisfiable, then $\Sigma \models \alpha$ for every wff $\alpha$.
- If $\alpha \models \beta$ (shorthand for $\{\alpha\} \models \beta$) and $\beta \models \alpha$, then $\alpha$ and $\beta$ are tautologically equivalent.
- $\Sigma \models \alpha$ if and only if $\land (\Sigma) \rightarrow \alpha$ is valid.
- Satisfiability and validity are dual notions: $\alpha$ is unsatisfiable if and only if $\neg \alpha$ is valid.

Propositional Logic in OCaml

We now take a look at Harrison’s encoding of propositional logic in OCaml, including a simple truth-table based tautology checker.

- formulas.ml
- prop.ml
- propexamples.ml

CNF

We next want to look at some algorithms that require their input to be in conjunctive normal form (CNF).

Recall that a propositional formula is in conjunctive normal form if it is a conjunction of disjunctions of literals, where a literal is either a propositional symbol or the negation of a propositional symbol. For example:

$$(p \lor \neg q) \land (\neg p \lor r) \land (\neg p \lor q \lor r)$$

Each conjunct is called a clause. In the above formula, the clauses are:

$$(p \lor \neg q), (\neg p \lor r), \text{ and } (\neg p \lor q \lor r).$$

A propositional symbol occurs positively if it occurs unnegated in a clause.

A propositional symbol occurs negatively if it occurs negated in a clause.

For example, in the above formula, $r$ occurs only positively, while $p$ and $q$ occur both positively and negatively.
**Boolean Gates**

Consider an electrical device having \( n \) inputs and one output. Assume that to each input we apply a signal that is either 1 or 0, and that this uniquely determines whether the output is 1 or 0.

The behavior of such a device is described by a Boolean function:

\[
F(X_1, \ldots, X_n) = \text{the output signal given the input signals } X_1, \ldots, X_n.
\]

We call such a device a **Boolean gate**.

The most common Boolean gates are **AND**, **OR**, and **NOT** gates.

**Boolean Circuits**

The inputs and outputs of Boolean gates can be connected together to form a **combinational Boolean circuit**.

A combinational Boolean circuit corresponds to a **directed acyclic graph** (DAG) whose leaves are inputs and each of whose nodes is labeled with the name of a Boolean gate. One or more of the nodes may be identified as outputs.

**Sharing Sub-Expressions**

This formula highlights an inefficiency in the logic representation as compared with the circuit representation. Since we are only concerned with the **satisfiability** of the formula, we can overcome this inefficiency by introducing new propositional symbols.

\[
(D \land (A \land B)) \lor ((A \land B) \land \neg C)
\]

Note that the new formula is *not* tautologically equivalent to the original formula (why?).

But it is equisatisfiable (i.e. the original formula is satisfiable iff the new formula is satisfiable).
Converting to CNF

This same idea is behind a simple algorithm for converting any propositional formula (or an associated Boolean circuit) into an equisatisfiable formula in conjunctive normal form (CNF) in linear time. We will view the formula or circuit as a directed acyclic graph (DAG).

1. Label each non-leaf node of the DAG with a new propositional symbol.
2. Construct a conjunction of disjunctive clauses which relate the inputs of that node to its output (the new propositional symbol)
3. The conjunction of all of these clauses together with a single clause consisting of the symbol for the root node is satisfiable iff the original formula is satisfiable.

Davis-Putnam Algorithm

The code for conversion to CNF is in `defcnf.ml`. We next look at `dp.ml` which contains two variations of the Davis-Putnam algorithm.

Both of these algorithms are decision procedures for satisfiability of propositional formulas in CNF.

The first algorithm, Davis-Putnam (DP) was published in 1960, and is often confused with the later, more popular algorithm presented by Davis, Logemann, and Loveland in 1962, which we will refer to as Davis-Putnam-Logemann-Loveland (DPLL).

We first consider the original DP algorithm.

Davis-Putnam Algorithm

There are three satisfiability-preserving transformations in DP.

- The 1-literal rule
- The affirmative-negative rule
- The rule for eliminating atomic formulas

The first two steps reduce the total number of literals in the formula. The last step reduces the number of variables in the formula.

By repeatedly applying these rules, eventually we obtain a formula containing an empty clause, indicating unsatisfiability, or a formula with no clauses, indicating satisfiability.
**Davis-Putnam Algorithm**

**The 1-literal rule**

Also called *unit propagation*.

Suppose \((p)\) is a unit clause (clause containing only one literal). Let \(\neg p\) denote the negation of \(p\) where double negation is collapsed (i.e. \(\neg \neg q \equiv q\)).

- Remove any instances of \(\neg p\) from the formula.
- Remove all clauses containing \(p\) (including the unit clause itself).

**Davis-Putnam Algorithm**

**The affirmative-negative rule**

Also called *pure literal rule*.

If a literal appears *only positively* or *only negatively*, delete all clauses containing that literal.

Why does this preserve satisfiability?

---

**DPLL Algorithm**

In the worst case, the resolution rule can cause a quadratic expansion every time it is applied.

For large formulas, this can quickly exhaust the available memory.

The DPLL algorithm replaces resolution with a *splitting rule*.

- Choose a propositional symbol \(p\) occurring in the formula.
- Let \(\Delta\) be the current set of clauses.
- Test the satisfiability of \(\Delta \cup \{p\}\).
- If satisfiable, return *true*.
- Otherwise, return the result of testing \(\Delta \cup \{\neg p\}\) for satisfiability.
**Experimental Results**

<table>
<thead>
<tr>
<th>Problem</th>
<th>tautology</th>
<th>dptaut</th>
<th>dpintaut</th>
</tr>
</thead>
<tbody>
<tr>
<td>prime 3</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>prime 4</td>
<td>0.02</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td>prime 9</td>
<td>18.94</td>
<td>2.98</td>
<td>0.51</td>
</tr>
<tr>
<td>prime 10</td>
<td>11.40</td>
<td>3.03</td>
<td>0.96</td>
</tr>
<tr>
<td>prime 11</td>
<td>28.11</td>
<td>2.98</td>
<td>0.51</td>
</tr>
<tr>
<td>prime 16</td>
<td>&gt;1 hour</td>
<td>out of memory</td>
<td>9.15</td>
</tr>
<tr>
<td>prime 17</td>
<td>&gt;1 hour</td>
<td>out of memory</td>
<td>3.87</td>
</tr>
<tr>
<td>ramsey 3 3 5</td>
<td>0.03</td>
<td>0.06</td>
<td>0.02</td>
</tr>
<tr>
<td>ramsey 3 3 6</td>
<td>5.13</td>
<td>8.28</td>
<td>0.31</td>
</tr>
<tr>
<td>mk_adder_test 3 2</td>
<td>&gt;&gt;1 hour</td>
<td>6.50</td>
<td>7.34</td>
</tr>
<tr>
<td>mk_adder_test 4 2</td>
<td>&gt;&gt;1 hour</td>
<td>22.95</td>
<td>46.86</td>
</tr>
<tr>
<td>mk_adder_test 5 2</td>
<td>&gt;&gt;1 hour</td>
<td>44.83</td>
<td>170.98</td>
</tr>
<tr>
<td>mk_adder_test 5 3</td>
<td>&gt;&gt;1 hour</td>
<td>38.27</td>
<td>250.16</td>
</tr>
<tr>
<td>mk_adder_test 6 3</td>
<td>&gt;&gt;1 hour</td>
<td>out of memory</td>
<td>1186.4</td>
</tr>
<tr>
<td>mk_adder_test 7 3</td>
<td>&gt;&gt;1 hour</td>
<td>out of memory</td>
<td>3759.9</td>
</tr>
</tbody>
</table>

**Stålmarck’s Method**

OCaml implementation in *stal.ml*.

Breadth-first approach instead of depth-first.

**Dilemma Rule**

Given a set of formulas $\Delta$ and any basic deduction algorithm, $R$, the dilemma rule performs a case split on some literal $p$ by considering the new sets of formulas $\Delta \cup \{\neg p\}$ and $\Delta \cup \{p\}$.

To each of these sets, the algorithm $R$ is applied to yield $\Delta_0$ and $\Delta_1$ respectively.

The original set $\Delta$ is then augmented with $\Delta_0 \cap \Delta_1$.

In 1994, Kunz and Pradhan developed a technique they called recursive learning which is very similar to the dilemma rule.

**Stålmarck’s Method**

Stålmarck’s Method takes as input a set of formulas $\Delta$ and a set of basic deduction rules $S_0$.

Applying $S_0$ to $\Delta$ until no further deductions are possible is called $0$-saturation.

Applying the dilemma rule with $R = S_0$ until no further deductions are possible is called $1$-saturation, and denoted $S_1$. Note that in order to achieve $1$-saturation, the dilemma rule is applied for every variable. This is why Stålmarck’s Method can be classified as a breadth-first strategy.

Repeatedly applying the dilemma rule with $R = S_1$ is called $2$-saturation, and denoted $S_2$.

In general, $S_{n+1}$ or $(n + 1)$-saturation is obtained by applying the dilemma rule with $R = S_n$.

**OCaml Example**

We will go through an example which should help you get started on the homework. The example is contained in *btree.ml*.
Stålmarck's Method

If a set of formulas $\Delta$ is decidable by $n$-saturation, then $\Delta$ is said to be $n$-easy. If, in addition, it is not decidable by $(n-1)$-saturation, it is said to be $n$-hard.

If $\Delta$ contains at most $n$ propositional symbols, then $\Delta$ is clearly $n$-easy. (Why?)

The merit of Stålmarck's method is that many practical problems appear to be $n$-easy for small values of $n$, often just $n = 1$.

If a formula with $m$ connectives is $n$-easy, Stålmarck's Method can decide it in time $O(m^n)$.

Stålmarck's Method: Implementation

Triplets

Stålmarck's Method first translates a formula into a set of “triplets” $p_i \leftrightarrow p_j \odot p_k$.

The conversion to triplets is analogous to the conversion to CNF we discussed last time. The only difference is that the equivalences at each connective are not transformed into clauses; they are left as equivalences.

Example

\[
(E \equiv A \land B), (G \equiv D \land E), (H \equiv E \land \neg C), (I \equiv G \lor H)
\]

Simple Rules

The rules for 0-saturation simply enumerate the new equivalences that can be deduced from a triplet given a set of existing equivalences.

Example

Consider the triplet $p \equiv q \land r$

If $r \equiv T$, then $p \equiv q$.

If $p \equiv T$, then $q \equiv T$ and $r \equiv T$.

If $q \equiv F$, then $p \equiv F$.

If $q \equiv r$, then $p \equiv q$ and $p \equiv r$.

If $p \equiv \neg q$, then $q \equiv T$ and $r \equiv F$.

These rules are called triggers.

In Harrison's OCaml implementation, the triggers are computed automatically.

0-saturation is done by using the triggers to deduce new equivalences until nothing new can be obtained or a contradiction ($T \equiv F$) is derived.

Stålmarck's Method: Implementation

The overall algorithm works as follows.

1. The negated formula is converted to triplets.

2. 0-saturation is performed. If a contradiction is obtained, we are done.

3. Otherwise, 1-saturation is performed: for each variable, the dilemma rule is used with $R = S_0$ to deduce new equivalences. If a contradiction is obtained, we are done.

4. Continue performing additional levels of saturation until a contradiction is obtained.

Note that the algorithm as given does not detect satisfiable formulas, only unsatisfiable formulas.

With some modification, the algorithm can be adapted to detect satisfiability as well.
Stålmarck’s Method: Performance

The procedure is quite effective in many cases. For primality formulas, it is generally comparable to DPLL. For Ramsey formulas, significantly worse. But for adder formulas it is substantially better.

Another class of formulas on which Stålmarck performs well is the so-called urquhart formulas:

\[ p_1 \iff p_2 \iff \cdots \iff p_n \iff p_1 \iff p_2 \iff \cdots \iff p_n. \]

These formulas are all 2-easy, whereas DPLL must search through nearly all possible cases to prove them.