Outline

- Ocaml
- Decision Procedures for Propositional Logic
  - Truth Table Method
  - Davis Putnam (DP)
  - Davis Putnam Logemann Loveland (DPLL)

Sources:
Harrison, John. *Introduction to Logic and Automated Theorem Proving*. Unpublished manuscript. Used by permission.

Ocaml

Ocaml is available from http://www.ocaml.org

Some of the useful resources available from this page include

- O’Caml manual (the *Index of Values* is especially helpful).
- Online tutorials
- Tuareg emacs Caml mode

If you have access to a machine on which you can install Ocaml, I suggest you do so.

If not, I will make a guest account available on one of my machines. Watch the email list for details (instructions for getting on the email list are on the course webpage).
Ocaml

We begin with a brief tour of Jason Hickey’s introduction to Objective Caml, available on the web at:


Harrison’s Ocaml Libraries

John Harrison has created a superb platform for playing with automated theorem proving in Ocaml. A complete set of files is available from his webpage at http://www.cl.cam.ac.uk/users/jrh/atp/index.html.

However, I will be providing a modified version of these files for use in the class. They will be available on the webpage as part of the first assignment.

The files we will use in this lecture are:

- startup.ml (among other things, loads lib.ml, intro.ml, formulas.ml)
- prop.ml (propositional reasoning)
- propexamples.ml (tools for generating propositional satisfiability problems)
- defcnf.ml (tools for conversion to CNF)
- dp.ml (DP and DPLL algorithms)
- bdd.ml (BDD package implemented in Ocaml)

Some Useful Ocaml Routines

Directives

- `#use "filename.ml"`
  - Load filename.ml as if it had been typed into the toplevel.

Built-in operations

- `+`, `+`, `/`
  - Addition of integers, floats, and Nums

- `@`
  - List concatenation

- `↑`
  - String concatenation

- `failwith "message"`
  - Raise a Failure exception with the given message.

Some Useful Ocaml Routines

From lib.ml

- `**`
  - Function composition.

- `hd l`, `tl l`
  - First element of a list `l`, everything but the first element.

- `itlist f l b`
  - Iterate over list `l` (from last element to first), applying binary function `f`. `b` provides the initial second argument to `f`.

- `end_itlist f l`
  - Same as above, except start with the last element of `l` instead of `b`.

- `partition p l`
  - Divide list `l` into a pair of lists based on predicate `p`.

- `filter p l`
  - Keep only elements of list `l` which satisfy predicate `p`.
Some Useful Ocaml Routines

From lib.ml (continued)

- \textit{length} \( l \)
  \text{Length of list} \( l \)

- \textit{find} \( p \) \( l \)
  \text{Return the first element of} \( l \) \text{satisfying predicate} \( p \).

- \textit{map} \( f \) \( l \)
  \text{Return a new list formed by applying function} \( f \) \text{to every element of} \( l \).

- \textit{smap} \( f \) \( l \)
  \text{Like map, but duplicates are removed}

- \textit{allpairs} \( f \) \( a \) \( b \)
  \text{Return a new list formed by applying binary function} \( f \) \text{to every pair of}
  \text{elements from lists} \( a \) \text{and} \( b \).

- \textit{assoc} \( a \) \( l \)
  \( l \) must be a list of pairs. \( \text{assoc} \ a \ l \) \text{returns the second part of the pair whose}
  \text{first part is} \( a \).

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.

- \textit{formulas.ml}
- \textit{prop.ml}
- \textit{propexamples.ml}

Some Useful Ocaml Routines

From lib.ml (continued)

- \textit{setify} \( l \)
  \text{Remove duplicates from a list.}

- \textit{union, intersect, subtract}
  \text{Set operations for lists.}

- \textit{insert} \( x \) \( l \)
  \text{Add an element to a list treated as a set (i.e. a duplicate element is not inserted).}

- \textit{mem} \( x \) \( l \)
  \text{Check if} \( x \) \text{appears in list} \( l \).

- \textit{implode, explode}
  \text{Convert list of chars to string and vice versa.}

In general, to find the definition of a routine, you should look in Harrison’s files first, and then check the “Index of Values” in the Ocaml manual.

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 \textit{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 \textit{positively} if it occurs unnegated in a clause.

A propositional symbol occurs \textit{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

The formula corresponding to the above circuit is:

$$(D \land (A \land B)) \lor ((A \land B) \land \neg C).$$

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.

$$(D \land (A \land B)) \lor (A \land B) \land \neg C.$$ 

A satisfying assignment for this formula gives the values that must be applied to the inputs of the circuit in order to set the output of the circuit to true.

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.

Converting to CNF: Example

\[(A \land B) \leftrightarrow E\]
\[((-A \land B) \rightarrow E) \land (E \rightarrow (A \land B))\]
\[(-A \lor -B \lor E) \land (-E \lor (A \land B))\]
\[(-A \lor -B \lor E) \land (-E \lor A) \land (-E \lor B)\]
\[(-A \lor -B \lor E) \land (-E \lor A) \land (-E \lor B)\]
\[(-C \lor F) \land (-F \lor C)\]
\[(-D \lor -E \lor G) \land (-G \lor D) \land (-G \lor E)\]
\[(-D \lor -E \lor G) \land (-G \lor D) \land (-G \lor E)\]
\[(-E \lor -F \lor H) \land (-H \lor E) \land (-H \lor F)\]
\[(-E \lor -F \lor H) \land (-H \lor E) \land (-H \lor F)\]
\[(G \lor H \lor \neg I) \land (I \lor \neg G) \land (I \lor \neg H)\]
\[(I)\]
**Davis-Putnam Algorithm**

The 1-literal rule

Also called unit propagation.

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

- Remove any instances of \(-p\) 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 \(\Delta \cup \{-p\}\) for satisfiability.
## Experimental Results

<table>
<thead>
<tr>
<th>Problem</th>
<th>tautology</th>
<th>dptaut</th>
<th>dpiltaut</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>