Review

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

- More OCaml
- Stålmarck’s Method
- Binary Decision Diagrams

Sources:

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 Unpublished manuscript.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


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

Ocaml implementation in \textit{stal.ml}.

Breadth-first approach instead of depth-first.

\textbf{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 \textit{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 $R = S_n$. 
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” $p_i \iff p_j \otimes p_k$.

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

$$(E \iff A \land B), (G \iff D \land E), (H \iff E \land \neg C'), (I \iff G \lor H)$$
Stålmarck’s Method: Implementation

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 \( p \iff q \land r \)

If \( r \iff T \), then \( p \iff q \).

If \( p \iff T \), then \( q \iff T \) and \( r \iff T \).

If \( q \iff F \), then \( p \iff F \).

If \( q \iff r \), then \( p \iff q \) and \( p \iff r \).

If \( p \iff \neg q \), then \( q \iff T \) and \( r \iff F \).

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 \((T \iff F)\) 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:

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

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

For $k \geq 0$, a $k$-place Boolean function is a function from $\{0, 1\}^k$ to $\{0, 1\}$. A Boolean function is anything which is a $k$-place Boolean function for some $k$.

Boolean functions can be represented by propositional formulas. However, the representation is not always efficient.

Binary Decision Diagrams (BDDs) are an efficient data structure for representing and performing operations on Boolean functions.

We will briefly describe what BDDs are and look at an implementation which includes a decision procedure in $bdd.ml$. 
Boolean Function Notation

Assume all functions are $n$-place Boolean functions on variables $x_1, \ldots, x_n$.

Identity: $x_i$

Negation: $\overline{f}$

Conjunction: $f \cdot g$

Disjunction: $f + g$
Definitions

Let \( f \) be an \( n \)-place Boolean function.

A \textit{restriction} or \textit{cofactor} of \( f \) is formed by replacing one of its arguments by a constant:

\[
f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).
\]

The \textit{Shannon expansion} of a function around variable \( x_i \) is given by

\[
f = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0}.
\]

The function resulting when some argument \( x_i \) of function \( f \) is replaced by function \( g \) is called a \textit{composition} of \( f \) and \( g \), and is denoted \( f|_{x_i=g} \):

\[
f|_{x_i=g}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n).
\]

Some functions may not depend on all their arguments. The \textit{dependency set} of a function \( f \), denoted \( I_f \), contains those arguments on which the function depends:

\[
I_f = \{ i \mid f|_{x_i=0} \neq f|_{x_i=1} \}.
\]
Binary Decision Trees

An *binary decision tree* is a rooted, directed tree with two types of vertices: *terminal vertices* and *nonterminal vertices*.

Each nonterminal vertex $v$ is labeled by a variable $\text{var}(v)$ and has two successors:

- $\text{low}(v)$ corresponding to the case where $\text{var}(v)$ is assigned 0, and
- $\text{high}(v)$ corresponding to the case where $\text{var}(v)$ is assigned 1.

Terminal vertices $v$ have no children and are labeled by $\text{value}(v) \in \{0, 1\}$. 
Binary Decision Trees

A binary decision tree $T$ defines a Boolean function $f_v$ for each vertex $v$ in the tree, defined as follows

- If $v$ is a terminal vertex, then
  - If $\text{value}(v) = 1$, then $f_v = 1$.
  - If $\text{value}(v) = 0$, then $f_v = 0$.

- If $v$ is a nonterminal vertex and $\text{var}(v) = x_i$, then
  \[ f_v(x_1, \ldots, x_n) = \overline{x_i} \cdot f_{\text{low}(v)}(x_1, \ldots, x_n) + x_i \cdot f_{\text{high}(v)}(x_1, \ldots, x_n). \]

The Boolean function defined by $T$ is the function $f_{\text{root}(T)}$ where $\text{root}(T)$ denotes the root vertex of $T$. 
Example

A binary decision tree for the two-bit comparator, given by the formula

\[ f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_3) \land (x_2 \leftrightarrow x_4), \]

is shown below (left is \textit{low}, right is \textit{high}).
Truth Assignments and Binary Decision Trees

To find the value of the function associated with the tree for a given truth assignment, simply traverse the tree from the root as follows.

- if $\text{var}(v)$ is assigned 0, move to low$(v)$.
- if $\text{var}(v)$ is assigned 1, move to high$(v)$.

The value that labels the terminal vertex is the value of the function for this assignment.
Truth Assignments and Binary Decision Trees

What is $f(1, 0, 1, 0)$, where

$$f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_3) \land (x_2 \leftrightarrow x_4)?$$
Truth Assignments and Binary Decision Trees

What is $f(1, 0, 1, 0)$, where

$$f(x_1, x_2, x_3, x_4) = (x_1 \leftrightarrow x_3) \land (x_2 \leftrightarrow x_4)$$

The path leads to a terminal vertex labeled with 1, so $f(1, 0, 1, 0) = 1$. 

19-a
A More Concise Representation

Binary decision trees do not provide a very concise representation for Boolean functions.

There is typically a lot of redundancy in such trees.

In the previous example, there are eight subtrees with roots labeled by $x_4$, but only three are distinct.

This observation leads to a natural improvement: merge isomorphic subtrees.

The result is a directed acyclic graph (DAG), called a *binary decision diagram* (BDD).

Note that the function represented is unchanged.
Example

After merging isomorphic subtrees, the example looks like this.
Ordered Binary Decision Diagrams

An *ordered binary decision diagram* (OBDD) has the additional property that for some ordering $\prec$ of the variables $x_1, \ldots, x_n$, $\text{var}(v) \prec \text{var}(\text{low}(v))$ and $\text{var}(v) \prec \text{var}(\text{high}(v))$ for each vertex $v$.

In his original paper, Bryant called these *function graphs*. 

10010
Ordered Binary Decision Diagrams

An ordered binary decision diagram (OBDD) has the additional property that for some ordering $\prec$ of the variables $x_1, \ldots, x_n$, $\text{var}(v) \prec \text{var}(\text{low}(v))$ and $\text{var}(v) \prec \text{var}(\text{high}(v))$ for each vertex $v$.

In his original paper, Bryant called these function graphs.

Our comparator example is an OBDD which uses the variable ordering:

$x_1 \prec x_3 \prec x_2 \prec x_4$. 

\[ 
\begin{array}{c}
\text{OBDD Diagram} \\
\end{array} 
\]
Reduced Binary Decision Diagrams

The representation can be made even more concise by eliminating vertices \( v \) for which \( \text{low}(v) = \text{high}(v) \). A BDD which contains no such vertices is called reduced.

Reduced Ordered Binary Decision Diagrams (ROBDD’s) have become the data structure of choice for representing Boolean functions, and are now the most common type of BDD.

The primary advantage of ROBDD’s is that they are canonical.

**Theorem**

For any \( n \)-place Boolean function \( f \), there is a unique ROBDD (on \( n \) variables) denoting \( f \) and any other OBDD denoting \( f \) contains more vertices.

**Proof**

By induction on the size of \( I_f \).
Canonicity of ROBDD's

Given any OBDD, an equivalent ROBDD can be computed in linear time by applying a procedure called \textit{Reduce}.

The fact that ROBDD’s are canonical make several important Boolean function operations trivial:

- Two Boolean functions are equivalent if they have isomorphic ROBDD’s.
- Satisfiability can be determined by simply checking if the ROBDD has a terminal labeled with 1.
- A tautology is represented by the ROBDD with a single vertex labeled 1.
Example

The ROBDD for the comparator example is:

![ROBDD Diagram]

From now on, when we refer to BDD's, we mean ROBDD's.

Note that the size of a BDD depends very much on the variable ordering.
Variable Ordering

In general, finding an optimal ordering is known to be $NP$-complete.

There are Boolean functions that have exponential size BDD’s for any variable ordering (multiplier).

However, heuristics have been developed for finding a good variable ordering when such an ordering exists.

Heuristics try to group related variables together.

For example, when converting a circuit to a BDD, the variables in a subcircuit are related because together they determine the output of that subcircuit.

Thus, these variables should usually be grouped together.

This can be done by placing the variables in the order in which they are encountered during a depth-first traversal of the circuit.
Dynamic Variable Ordering

A technique called *dynamic reordering* can be useful if no obvious ordering heuristic applies.

When this technique is used, the BDD package internally tries a variety of reorderings and keeps the best one.

Uses various techniques to try to find minimum BDD sizes without getting stuck in a local minimum.
Logical Operations on BDD’s

We begin with the operation of restricting some argument $x_i$ of the Boolean function $f$ to a constant value $b$.

Recall the definition of the restriction or cofactor of $f$:

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n).$$

If $f$ is represented by a BDD, the BDD for the restriction $f|_{x_i=b}$ is computed by a depth-first traversal of the BDD.

For any vertex $v$ which has a pointer to a vertex $w$ such that $\text{var}(w) = x_i$, we replace the pointer by $\text{low}(w)$ if $b$ is 0 and $\text{high}(w)$ if $b$ is 1.

After this transformation, $\text{Reduce}$ is applied to ensure that the result is canonical.
Logical Operations

All 16 binary propositional connectives can be implemented efficiently on Boolean functions that are represented as BDD’s.

In fact, the complexity of these operations is linear in the size of the argument BDDs.

The key idea for efficient implementation of these operations is the *Shannon expansion*:

\[
f = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0}.
\]
Logical Operations

Bryant gives a uniform algorithm called Apply for computing all 16 binary operations.

Let \( \circ \) be an arbitrary binary operation, and let \( f \) and \( f' \) be two Boolean functions. To compute \( f \circ f' \):

1. If \( \text{root}(f) \) and \( \text{root}(f') \) are both terminal vertices, then
   \[
   f \circ f' = \text{value}(\text{root}(f)) \circ \text{value}(\text{root}(f')).
   \]

2. If \( \text{var}(\text{root}(f)) = \text{var}(\text{root}(f')) \), then use the Shannon expansion. Let \( x = \text{var}(\text{root}(f)) = \text{var}(\text{root}(f')) \):
   \[
   f \circ f' = x \cdot (f|_{x=1} \circ f'|_{x=1}) + \bar{x} \cdot (f|_{x=0} \circ f'|_{x=0}).
   \]

Notice that this effectively breaks the problem into two subproblems which are solved recursively.

The root of the resulting BDD will be a vertex \( v \) labeled by \( x \).

The first part of this expression computes \( \text{high}(v) \), and the second part of the expression computes \( \text{low}(v) \).
Logical Operations

Computing $f \odot f'$, continued. Let $x = \text{var}(\text{root}(f))$ and $x' = \text{var}(\text{root}(f'))$:

3. If $x < x'$, then $f'|_{x=0} = f'|_{x=1} = f'$ since $f'$ does not depend on $x$. In this case, the Shannon expansion simplifies to:

$$f \odot f' = x \cdot (f|_{x=1} \odot f') + \overline{x} \cdot (f|_{x=0} \odot f').$$

The BDD is then computed recursively as in the second case.

4. If $x > x'$, then $f|_{x'=0} = f|_{x'=1} = f$ since $f$ does not depend on $x'$. In this case, the Shannon expansion simplifies to:

$$f \odot f' = x' \cdot (f \odot f'|_{x'=1}) + \overline{x'} \cdot (f \odot f'|_{x'=0}).$$

The BDD is computed recursively as before.
Logical Operations

By using dynamic programming, it is possible to make the algorithm polynomial.

- A hash table is used to record all previously computed subproblems.
- Before any recursive call, the table is checked to see if the subproblem has been solved.
- If it has, the result is obtained from the table; otherwise, the recursive call is performed.
- The result must be reduced to ensure that it is in canonical form.
BDD Extensions

A single DAG can be used to represent a collection of Boolean functions:

- The same variable ordering is used for all of the functions.
- All identical subgraphs are merged.
- Two functions are identical iff they have the same root.
- Checking equivalence can be done in constant time.

Another useful extension adds labels to the edges in the DAG to denote Boolean negation. This makes it unnecessary to use different subgraphs for a formula and its negation.
BDD Implementation

In Harrison’s implementation, a bdd node consists of a variable and a left and right child. Each bdd node is also indexed by an integer.

A global table, called the *unique table* maps bdd nodes to their index (a similar table is kept for the reverse mapping).

Another table, the *computed table* caches bdd operations that have been done before to avoid repeating work that has already been done.

A basic tautology checker *bddtaut* can be implemented using bdd’s. It’s performance is comparable to DPLL.

For circuit-like examples, a more powerful tautology checker, *ebddtaut*, takes advantage of the structure of the formula in both the translation to the bdd and the variable ordering.