H250: Honors Colloquium – Introduction to Computation

Representing Propositional Formulas

Marius Minea
marius@cs.umass.edu
Colloquium Topics (tentative)

- Binary decision diagrams
- Conjunctive normal form
- SAT checking
- Axiomatization of logic
- Resolution theorem proving, unification
- Fixpoint theorem
- Representing and exploring search spaces
- Number Theory
- Combinatorics
- Linear recurrences
- Recursive tree traversals
- Grammars and parsing
- Automata properties and testing
- Automata learning
Logistics

Grading:

70% homeworks ($\sim$1 problem / week)

30% final project (discuss/agree topic by Columbus Day)

Q&A: Campuswire
Today

*Binary Decision Diagrams*

Conjunctive Normal Form

Tseitin Transformation
Review: Propositional Formulas

Boolean operators: $\neg, \land, \lor$.
We’ll rewrite $\rightarrow, \leftrightarrow, \oplus$ in these terms.

Interesting Questions:

Are two formulas $A$ and $B$ equivalent? $A \leftrightarrow B$

Is a formula a tautology? (true in all truth assignments)

Is a formula satisfiable? (in at least one truth assignment)

Is a formula a contradiction? (has no satisfying assignment)
Representing Propositional Formulas

Ideally, a representation should be

*canonical* (a formula is represented in a single way)
two formulas are equal precisely if they have same representation

*simple* and *compact* (easy to implement and store)

*easy to process* (simple, efficient algorithms)

*Binary decision diagrams* are such a representation for Boolean formulas (Bryant, 1986)
Review: Truth tables

Truth tables show the value of a formula for all truth assignments (all interpretations).

Two formulas are *equivalent* if they have the *same truth table*

But: $2^n$ combinations (lines) for a formula with $n$ propositions

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a →(b →c)</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>(a →b) →c</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Boole expansion / Shannon decomposition wrt a variable

By fixing the value of a variable, the formula simplifies

Let \( f = (a \lor b) \land (a \lor c) \land (\neg a \lor \neg b \lor c) \).
We assign values T and F to variable \( a \) (two halves of truth table)

\[
\begin{align*}
  f|_{a=T} &= T \land T \land (\neg b \lor c) = \neg b \lor c \\
  f|_{a=F} &= b \land c \land T = b \land c
\end{align*}
\]

**Boole expansion**
(or **Shannon decomposition**)

\[
f = x \land f|_{x=T} \lor \neg x \land f|_{x=F}
\]

expresses a Boolean function \( f \) with respect to a variable \( x \)

In code (ML): if-then-else with variable as condition

```ml
if a then not b || c else b && c
```
Binary decision tree

Continuing with subformulas, we obtain a decision tree: assigning variables and following branches (true/false), we obtain the function value (\( T/F \), or 0/1)

\[
\begin{align*}
  f|_{a=T} &= T \land T \land (\neg b \lor c) = \neg b \lor c \\
  f|_{a=F} &= b \land \neg c \land T = b \land c \\
  f|_{a=T,b=F} &= T, \quad f|_{a=T,b=T} = c, \text{ etc.}
\end{align*}
\]

With a fixed variable ordering, the tree is unique: canonical still inefficient: up to \( 2^n \) possible combinations, like the truth table.
From decision tree to decision diagram

$f(x_1, x_2, x_3) = (\neg x_1 \land x_2 \land x_3) \lor (x_1 \land \neg x_2 \land x_3) \lor (x_1 \land x_2 \land x_3)$

e.g., $f(T, F, T) = T$, $f(F, T, F) = F$, etc.

*leaf*/terminal nodes: function value (0 or 1, i.e, F or T)
*inner*/nonterminal nodes: $x_i$ (variables the function depends on)
branches: *low*(n) / *high*(n) : assignment F/T of node variable

We define 3 transformation rules for a more compact form: *binary decision diagram*. 
Reduction nr. 1: Merge leaf nodes

We keep a single copy for nodes 0 and 1
Reduction nr. 2: Merge nodes with same structure

If \( \text{low}(n_1) = \text{low}(n_2) \) and \( \text{high}(n_1) = \text{high}(n_2) \), we merge \( n_1 \) and \( n_2 \). Two nodes with equal results on the false branch and equal results on the true branch yield the same value.
Reduction nr. 3: Eliminate useless tests

Eliminate nodes with the same result on false and true branches
The three reductions are used to define a BDD.
In practice, we want to avoid the decision tree due to its size.
We directly apply expansion with respect to a variable.
Practical BDD construction

Don’t start from a complete binary tree

Build BDD directly, *recursively, decomposing* with respect to a variable:

\[ f = x_1 \land f|_{x_1=T} \lor \neg x_1 \land f|_{x_1=F} \]

**build** \( f|_{x_1=T} \) and \( f|_{x_1=F} \)

**merge** any common nodes

BDD libraries: use recursive processing with *lookup/hashings*
if BDD already exists, it is found and used, not duplicated
equivalence check is effectively pointer comparison!
Example BDD construction

\[ f(x_1, x_2, x_3) = (\neg x_1 \land x_2 \land x_3) \lor (x_1 \land \neg x_2 \land x_3) \lor (x_1 \land x_2 \land x_3) \]

Choose a variable: \( x_1 \). Compute \( f|_{x_1=F} \) and \( f|_{x_1=T} \)

Build BDD for the two functions: directly, if simple (T, F, \( p \), \( \neg p \)), else \textit{recursively}, choosing \textit{a new variable}:

\[
\begin{align*}
  f_1 &= f|_{x_1=F} = x_2 \land x_3 \\
  f_1|_{x_2=F} &= F \\
  f_1|_{x_2=T} &= x_3 \\
  f|_{x_1=T} &= x_3
\end{align*}
\]

Add decision node for \( x_2 \)

Add decision based on \( x_1 \)

The BDD rooted at \( x_3 \) is common, we keep one copy.
Use of BDDs

Represent Boolean functions efficiently
small in many practical cases (still worst-case exponential)

Check equivalence of two Boolean functions

Both are needed in CAD software for circuit design (logic synthesis)
apply optimizations
then check that result is correct (equivalent to original)
(two equivalent circuits/functions must have same BDD)

Frequent use in formal verification (check system correctness),
efficiently represent large state spaces, etc.

Construct BDDs interactively:
http://formal.cs.utah.edu:8080/pbl/BDD.php
Can we represent a propositional formula in a “systematic” way?

Can we represent propositional formulas uniquely?

Ideally, a representation should be

*simple* and *compact* (easy to implement and store)

*easy to process* (simple, efficient algorithms)

*canonical* (a formula is represented in a single way)

two formulas are equal precisely if they have same representation