Lecture 12: Binary Decision Diagrams in Detail

- Quantified Boolean Formulas
- Logical Operations on OBDDs
- Variable Ordering Problem
- Ordered Binary Decision Diagrams
- Binary Decision Trees
Binary Decision Diagrams

Ordered binary decision diagrams (OBDs) are a canonical form for Boolean formulas.

Moreover, they can be manipulated very efficiently.

OBDs are often substantially more compact than traditional normal forms.

On Computers, C-35(8), 1986.

Binary Decision Trees

To motivate our discussion of binary decision diagrams, we first consider binary decision trees.
A binary decision tree for the two-bit comparator, given by the formula

\[(\bar{q} \leftrightarrow \bar{r}) \lor (\bar{t} \leftrightarrow \bar{d}) = (\bar{q}, \bar{r}, \bar{t}, \bar{d})_f\]

is shown in the figure below:
We can decide if a truth assignment satisfies the formula as follows:

1. Traverse the tree from the root to a terminal vertex.
2. If variable $a$ is assigned 1, the next vertex on the path will be labeled high.
3. If variable $a$ is assigned 0, the next vertex on the path will be labeled low.
4. The value that labels the terminal vertex will be the value of the function for this assignment.

In the comparator example, the assignment

\[ q \rightarrow q, q \rightarrow q, q \rightarrow 0, q \rightarrow 1, q \rightarrow 1 \]

leads to a leaf vertex labeled 0, so the formula is false for this assignment.
Binary decision trees do not provide a very concise representation for boolean functions. However, there is usually a lot of redundancy in such trees. Thus, we can obtain a more concise representation by merging isomorphic subtrees. This results in a directed acyclic graph (DAG) called a binary decision diagram (BDD).
Binary Decision Diagrams

More precisely, a binary decision diagram is a rooted, directed acyclic graph with two types of vertices, terminal vertices and nonterminal vertices. Each nonterminal vertex is labeled by a variable and has two successors, low and high. Each terminal vertex is labeled by either 0 or 1.

A binary decision diagram with root $a$ determines a boolean function $f$ in the following manner:

1. If $a$ is a terminal vertex:
   - If $f$ is true, then $f = (a)\text{true}$ (q)
   - If $f$ is false, then $f = (a)\text{false}$ (p)

2. If $a$ is a nonterminal vertex with $u$ predecessors:
   - If $f = 0$, then $f = (u_1, \ldots, u_n)^f$ (a)\text{false}
   - If $f = 1$, then $f = (u_1, \ldots, u_n)^f$ (a)\text{true}

where $u_i$ are the variables associated with each predecessor.
In practical applications, it is desirable to have a canonical representation for Boolean functions. A canonical representation must guarantee that two Boolean functions are logically equivalent if and only if they have isomorphic representations. This simplifies tasks like checking equivalence of two formulas and deciding if a given formula is satisfiable or not.

Canonical Form Property
Two binary decision diagrams are isomorphic if there exists a bijection $\eta$ between the graphs such that

$$(((a)\eta)\eta)\eta = (((a)\eta)\eta)\eta$$

and

$$(((a)\eta)\eta)\eta = (((a)\eta)\eta)\eta$$

and

$$(((a)\eta)\eta)\eta = (((a)\eta)\eta)\eta$$

for every nonterminal vertex $\omega$.

For every terminal vertex $\omega$, value $\text{value} = (a)\omega$.

Terminals are mapped to terminals and nonterminals are mapped to nonterminals, such that

Canonical Form Property (Cont.)
Bryant showed how to obtain a canonical representation for Boolean functions by placing two restrictions on binary decision diagrams:

First, the variables should appear in the same order along each path from the root to a terminal.

Second, there should be no isomorphic subroutines or redundant vertices in the diagram.
The first requirement is easy to achieve:

1. We require that if vertex \( v \) has a nonterminal successor \( w \), then \( v \succ w \).
2. We impose total ordering \( \succ \) on the variables in the formula.

Canonical Form Property (Cont.)
The second requirement is achieved by repeatedly applying three transformation rules that do not:

**Remove duplicate nonterminals:** Eliminate all but one nonterminal vertex with a given label and redirect all arcs to the eliminated vertex to the remaining one.

**Remove duplicate terminals:** Eliminate all but one terminal vertex with a given label and redirect all incoming arcs to the other vertex.

**Remove redundant tests:** If nonterminal vertex \( (a)_\text{high} = (n)_\text{low} = (n) \) and \( (a)_\text{low} = (a)_\text{high} \), then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.

**Remove duplicate terminals:** Eliminate all but one terminal vertex with a given label and redirect all incoming arcs to the other vertex.

**Remove redundant tests:** If nonterminal vertex \( (a)_\text{high} = (n)_\text{low} = (n) \) and \( (a)_\text{low} = (a)_\text{high} \), then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.
Bryant shows how this can be done by a procedure called Reduce in linear time. The canonical form may be obtained by applying the transformation rules until the size of the diagram can no longer be reduced.
The term ordered binary decision diagram (OBDD) will be used to refer to the graph obtained in this manner.

If OBDDs are used as a canonical form for boolean functions, then checking satisfiability is reduced to checking isomorphism between OBDDs, and checking equivalence is reduced to checking isomorphism between OBDDs. Only one terminal labeled by 0 is used in this manner.

Ordered Binary Decision Diagrams
If we use the ordering of $q^2 > q^1 > q_1 > q_0$, we obtain the OBDD below:

**OBDD for Comparator Example**
With the ordering $a_1 > b_2 > a_2 > q_2$, we get

The size of an OBDD depends critically on the variable ordering.

Variable Ordering Problem
For an $n$-bit comparator:

- For any variable ordering, the number of vertices will be $3n + 2$.
- If we use the ordering $a_1 > a_2 > \cdots > a_n$, the number of vertices is $3n + 1$.
- If we use the ordering $a_1 > a_2 > \cdots > a_n$, the number of vertices is $3n + 1$.

Moreover, there are boolean functions that have exponential size OBDDs for any variable ordering.

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

An example is the middle output (output) of a combinational circuit to multiply two $n$-bit integers.

Variable Ordering Problem (Cont.)
Heuristics for Variable Ordering

Heuristics have been developed for finding a good variable ordering when such an ordering exists. The intuition for these heuristics comes from the observation that OBDDs tend to be small when related variables are close together in the ordering. The variables appearing in a subcircuit are related in that they determine the subcircuit’s output. Hence, these variables should usually be grouped together in the ordering.

During a depth-first traversal of the circuit diagram, this may be accomplished by placing the variables in the order in which they are encountered during a depth-first traversal of the circuit diagram.
A technique, called \textit{dynamic reordering}, appears to be useful if no obvious ordering heuristic applies.

When this technique is used, the OBDD package internally reorders the variables periodically to reduce the total number of vertices in use.
Logical Operations on OBDDs

When the graph is not in canonical form, we apply Reduce to obtain the OBDD for

\[ q \rightarrow 1^x | f \]

If the graph is in canonical form, we apply Minimize to obtain the OBDD for

\[ q \rightarrow 1^x | f \]

For any vertex \( v \) in which has a pointer to a vertex \( w \) such that \( v \) \( q \rightarrow 1^x \) and \( w \) \( q \rightarrow 0 \), we replace the pointer by

\[ \text{low} \]

If \( q \) is 0 and by \( \text{high} \) if \( q \) is 1.

For any vertex \( v \) in which has a pointer to a vertex \( w \) such that \( v \) \( q \rightarrow 1^x \), we replace the pointer by

\[ \text{low} \]

If \( q \) is 1.

This function is denoted by \( q \rightarrow 1^x | f \) and satisfies the identity

\[ \frac{x}{q} | f \]

We begin with the function that restricts some argument \( x \) of the boolean function to a constant value.
The key idea for efficient implementation of these operations is the Shannon expansion.

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

All 16 two-argument logical operations can be implemented efficiently on boolean functions that are represented as OBDDs.
Logical Operations (Cont.)

Bryant gives a uniform algorithm called APPLY for computing all 16 logical operations.

Let \( \cdot, + \) be an arbitrary two argument logical operation, and let \( f \) and \( \tilde{f} \) be two boolean functions.

To simplify the explanation of the algorithm we introduce the following notation:

\[
\text{root}(f) \quad \text{and} \quad \text{root}(\tilde{f})
\]

where \( \text{root}(f) \) and \( \text{root}(\tilde{f}) \) are the roots of the OBDDs for \( f \) and \( \tilde{f} \).
We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

For the rest of this section, we consider several cases depending on the relationships between $a$ and $\omega$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.

We consider several cases depending on the relationships between $a$ and $\omega$.  

If $a$ and $\omega$ are both terminal vertices, then $\omega = \omega \cdot a = \omega \cdot a$.

If $a$ is a terminal vertex and $\omega$ is the root of the resulting OBDD, then $\omega = \omega \cdot a$.
Logical Operations OBDDs (Cont.)

If $x > \overline{x}$, then the required computation is similar to the previous case.

If $x > \overline{x}$, then the required computation is similar to the previous case.

and the OBDD for $f \times f$ is computed recursively as in the second case:

$$\left( f \times \begin{cases} 1 \rightarrow x \end{cases} f \right) \cdot x + \left( f \times \begin{cases} 0 \rightarrow x \end{cases} f \right) \cdot \overline{x} = f \times f$$

In this case the Shannon Expansion simplifies to

$$\text{since } f = \begin{cases} 1 \rightarrow x \end{cases} f = \begin{cases} 0 \rightarrow x \end{cases} f$$

If $x > \overline{x}$, then

Logical Operations OBDDs (Cont.)
By using dynamic programming, it is possible to make the algorithm polynomial.

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

Logical Operations (Cont.)
Several extensions have been developed to decrease the space requirements for OBDDs.

Consequently, checking whether two functions are equal can be implemented in constant time.

Two functions in the collection are identical if and only if they have the same root.

As before, the graph contains no isomorphic subgraphs or redundant vertices.

The same variable ordering is used for all of the formulas in the collection.

A single OBDD can be used to represent a collection of boolean functions:

Sever extending have been developed to decrease the space requirements for OBDDs.
Another useful extension adds labels to the arcs in the graph to denote Boolean negation. This makes it unnecessary to use different subgraphs to represent a formula and its negation.

OBDDs with hundreds of thousands of vertices can be manipulated efficiently.

OBDD Extensions (Cont.)
OBDDs and Finite Automata

Logical operations on boolean functions can be implemented by standard constructions from elementary automata theory.

The DFA provides a canonical form for the original boolean function.

This is a finite language. Finite languages are regular. Hence, there is a minimal DFA that accepts the language.

An \( n \)-argument boolean function can be identified with the set of strings in \( \{0,1\}^n \) that evaluate to \( 1 \).

OBDDs can also be viewed as deterministic finite automata.