Lecture 12: Binary Decison Diagrams in Detail

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

Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) are a canonical form for boolean formulas.

OBDDs are often substantially more compact than traditional normal forms.

Moreover, they can be manipulated very efficiently.

- R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions 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 is a rooted, directed tree with two types of vertices, *terminal vertices* and *nonterminal vertices*.

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

- $low(v)$ corresponding to the case where the variable v is assigned 0, and
- $high(v)$ corresponding to the case where the variable v is assigned 1.

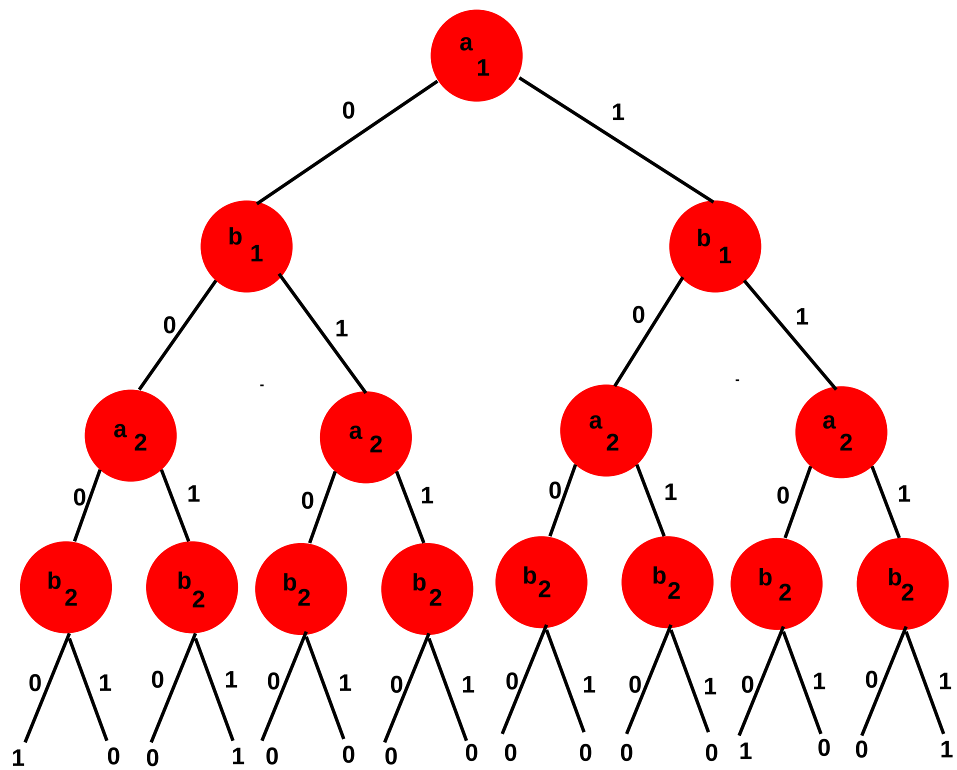
Each terminal vertex v is labeled by $value(v)$ which is either 0 or 1.

Binary Decision Trees (Cont.)

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

$$f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2),$$

is shown in the figure below:



Binary Decision Trees (Cont.)

We can decide if a truth assignment satisfies the formula as follows:

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

In the comparator example, the assignment

$$a_1 \leftarrow 1,\ \ a_2 \leftarrow 0,\ \ b_1 \leftarrow 1,\ \ b_2 \leftarrow 1$$

leads to a leaf vertex labeled 0, so the formula is false for this assignment.

A More Concise Representation

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

However, there is usually a lot of redundancy in such trees.

In the comparator example, there are eight subtrees with roots labeled by b_2, but only three are distinct.

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*.

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 v is labeled by a variable $var(v)$ and has two successors, $low(v)$ and $high(v)$.

Each terminal vertex is labeled by either 0 or 1.

A binary decision diagram with root v determines a boolean function $f_v(x_1, \ldots, x_n)$ in the following manner:

1. If v is a terminal vertex:
   (a) If $value(v) = 1$ then $f_v(x_1, \ldots, x_n) = 1$.
   (b) If $value(v) = 0$ then $f_v(x_1, \ldots, x_n) = 0$.
2. If v is a nonterminal vertex with $var(v) = x_i$ then $f_v(x_1, \ldots, x_n)$ is given by

$$\bar{x}_i \cdot f_{low(v)}(x_1, \ldots, x_n) \; + \; x_i \cdot f_{high(v)}(x_1, \ldots, x_n)$$

Canonical Form Property

In practical applications, it is desirable to have a *canonical representation* for boolean functions.

This simplifies tasks like checking equivalence of two formulas and deciding if a given formula is satisfiable or not.

Such a representation must guarantee that two boolean functions are logically equivalent if and only if they have isomorphic representations.

Canonical Form Property (Cont.)

Two binary decision diagrams are *isomorphic* if there exists a bijection h between the graphs such that

- terminals are mapped to terminals and nonterminals are mapped to nonterminals,
- for every terminal vertex v, $value(v) = value(h(v))$, and
- for every nonterminal vertex v:
  - $var(v) = var(h(v))$,
  - $h(low(v)) = low(h(v))$, and
  - $h(high(v)) = high(h(v))$.

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 subtrees or redundant vertices in the diagram.

Canonical Form Property (Cont.)

The first requirement is easy to achieve:

- We impose total ordering $<$ on the variables in the formula.
- We require that if vertex u has a nonterminal successor v, then $var(u) < var(v)$.

Canonical Form Property (Cont.)

The second requirement is achieved by repeatedly applying three transformation rules that do not alter the function represented by the diagram:

Remove duplicate terminals: Eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one.

Remove duplicate nonterminals: If nonterminals u and v have $var(u) = var(v)$, $low(u) = low(v)$ and $high(u) = high(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.

Remove redundant tests: If nonterminal vertex v has $low(v) = high(v)$, then eliminate v and redirect all incoming arcs to $low(v)$.

Canonical Form Property (Cont.)

The canonical form may be obtained by applying the transformation rules until the size of the diagram can no longer be reduced.

Bryant shows how this can be done by a procedure called *Reduce* in linear time.

Ordered Binary Decision Diagrams

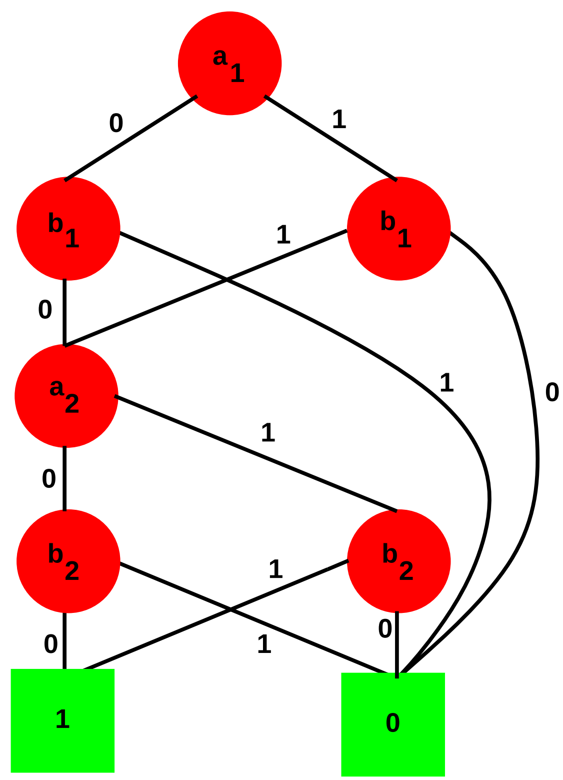
The term *ordered binary decision diagram* (OBDD) will be used to refer to the graph obtained in this manner.

If OBDDs are used as a canonical form for boolean functions, then

- checking equivalence is reduced to checking isomorphism between OBDDs, and
- satisfiability can be determined by checking equivalence with the trivial OBDD that consists of only one terminal labeled by 0.

OBDD for Comparator Example

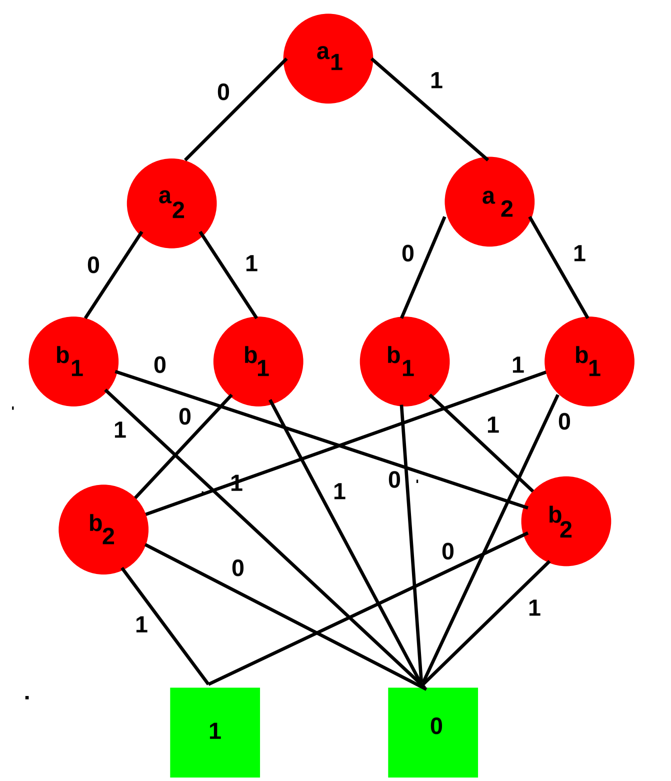
If we use the ordering $a_1 < b_1 < a_2 < b_2$ for the comparator function, we obtain the OBDD below:



Variable Ordering Problem

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

With the ordering $a_1 < a_2 < b_1 < b_2$, we get



Variable Ordering Problem (Cont.)

For an n-bit comparator:

- if we use the ordering $a_1 < b_1 < \ldots < a_n < b_n$, the number of vertices will be $3n + 2$.
- if we use the ordering $a_1 < \ldots < a_n < b_1 \ldots < b_n$, the number of vertices is $3 \cdot 2^n - 1$.

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

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

An example is the middle output (n^{th} output) of a combinational circuit to multiply two n bit integers.

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.

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.

Dynamic Variable Ordering

A technique, called *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

We begin with the function that restricts some argument x_i of the boolean function f to a constant value b.

This function is denoted by $f\mid_{x_i \leftarrow b}$ and satisfies the identity

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

If f is represented as an OBDD, the OBDD for the restriction $f\mid_{x_i \leftarrow b}$ is computed by a depth-first traversal of the OBDD.

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

When the graph is not in canonical form, we apply $Reduce$ to obtain the OBDD for $f\mid_{x_i \leftarrow b}$.

Logical Operations (Cont.)

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

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

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

$$f = \bar{x} \cdot f \mid_{x \leftarrow 0} + x \cdot f \mid_{x \leftarrow 1} .$$

Logical Operations (Cont.)

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

Let $\star$ be an arbitrary two argument logical operation, and let f and f' be two boolean functions.

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

- v and v' are the roots of the OBDDs for f and f'.
- $x = var(v)$ and $x' = var(v')$.

Logical Operations OBDDs (Cont.)

We consider several cases depending on the relationship between v and v'.

- If v and v' are both terminal vertices, then $f \star f' = value(v) \star value(v')$.
- If $x = x'$, then we use the Shannon expansion

$$f \star f' = \bar{x} \cdot (f \mid_{x \leftarrow 0} \star f' \mid_{x \leftarrow 0}) + x \cdot (f \mid_{x \leftarrow 1} \star f' \mid_{x \leftarrow 1})$$

to break the problem into two subproblems. The subproblems are solved recursively.

The root of the resulting OBDD will be v with $var(v) = x$.

$Low(v)$ will be the OBDD for $(f \mid_{x \leftarrow 0} \star f' \mid_{x \leftarrow 0})$.

$High(v)$ will be the OBDD for $(f \mid_{x \leftarrow 1} \star f' \mid_{x \leftarrow 1})$.

Logical Operations OBDDs (Cont.)

- If $x < x'$, then $f' |_{x \leftarrow 0} = f' |_{x \leftarrow 1} = f'$ since f' does not depend on x.

  In this case the Shannon Expansion simplifies to

  $$f \star f' = \bar{x} \cdot (f |_{x \leftarrow 0} \star f') + x \cdot (f |_{x \leftarrow 1} \star f')$$

  and the OBDD for $f \star f'$ is computed recursively as in the second case.
- If $x' < x$, then the required computation is similar to the previous case.

Logical Operations (Cont.)

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.

OBDD Extensions

Several extensions have been developed to decrease the space requirements for OBDDs.

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

- The same variable ordering is used for all of the formulas in the collection.
- As before, the graph contains no isomorphic subgraphs or redundant vertices.

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

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

OBDD Extensions (Cont.)

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.

OBDDs and Finite Automata

OBDDs can also be viewed as *deterministic finite automata*.

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

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

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

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