Ordered Binary Decision Diagrams

Tobias Berka

July 15, 2009

1 Introduction

Ordered binary decision diagrams (introduced in [3]) are primarily a means for representing and manipulating Boolean functions, i.e. functions of the form $\phi : \{0,1\}^n \to \{0,1\}$, which are of prime interest to computer scientists with applications in switching circuits, VLSI design, formal verification and automated theorem proving. The first goal for the task at hand is to avoid the exponential size complexity of naive truth value tables for a large class of functions encountered in practice. Historically, they are an extension of (unordered) binary decision diagrams (BDDs) first introduced in [4] and popularized by [1]. The formal inspiration for BDDs is the Shannon expansion. If we write $\overline{x}$ for $\neg x$, multiplication for conjunction and addition for disjunction, a function

$$\begin{array}{rcl} f : \{0,1\}^n & \to & \{0,1\}, \\ (x_1, x_2, ..., x_n) & \mapsto & f(x_1, x_2, ..., x_n), \end{array}$$

can be expanded into

$$\begin{array}{rcl} f(x_1, x_2, ..., x_n) & = & \overline{x}_1 \cdot f(0, x_2, ..., x_n) + x_1 \cdot f(1, x_2, ..., x_n), \\ f(x_1, x_2, ..., x_n) & = & \overline{x}_2 \cdot f(x_1, 0, ..., x_n) + x_2 \cdot f(x_1, 1, ..., x_n), \\ & \vdots & \\ f(x_1, x_2, ..., x_n) & = & \overline{x}_n \cdot f(x_1, x_2, ..., 0) + x_n \cdot f(x_1, x_2, ..., 1), \end{array}$$

which can further be applied, e.g.

$$\begin{array}{rcl} f(x_1, ..., x_n) & = & \overline{x}_1 \cdot f(0, x_2, ..., x_n) + x_1 \cdot f(1, x_2, ..., x_n), \\ & = & \overline{x}_1 \cdot (\overline{x}_2 \cdot f(0, 0, ..., x_n) + x_2 \cdot f(0, 1, ..., x_n)) \\ & & + x_1 \cdot (\overline{x}_2 \cdot f(1, 0, ..., x_n) + x_2 \cdot f(1, 1, ..., x_n)) \\ & \vdots & \end{array}$$

This process could be continued until all occurrences of the variables $x_1, ..., x_n$ have been replaced by constants and the corresponding evaluations of the function f can be inserted into the formula, thus giving us a canonical formula for

the function (with exponential size complexity). But instead of computing the full decomposition, a BDD is a graph representing the decomposition rather than its result. An OBDD augments the BDD by imposing an arbitrary ordering on the variables and applying the Shannon expansion strictly by this order from the least to the greatest variable.

Formally, an OBDD for an n-ary Boolean function $f : \{0,1\}^n \to \{0,1\}$ is a graph with vertices V with the following functions:

- We have two functions $lo, hi : V \to V$ assigning a "low" and "high" successor to every non-terminal node.
- There is a function $val : V \to \{0,1\}$ assigning a value to every terminal node.
- A function $var : V \to \{1, ..., n\}$ assigns a variable index to every vertex, where every non-terminal node v satisfies the ordering constraint

$$var(v) < var(lo(v)) \wedge var(v) < var(hi(v)).$$

- For every node $v \in V$ with variable index $i = var(v)$ there is a recursive evaluation function

$$\begin{aligned} f_v : \{0,1\}^n &\to \{0,1\} \\ f_v(\vec{x}) &= \begin{cases} val(v), & v \text{ is a terminal node,} \\ \overline{x}_i \cdot f_{lo(v)}(\vec{x}) + x_i \cdot f_{hi(v)}(\vec{x}), & \text{otherwise.} \end{cases} \end{aligned}$$

For equivalence testing of Boolean functions, we define a simple isomorphism on OBDDs: G and G' are isomorphic if there exists a bijection σ satisfying the conditions:

- if $t \in G$ is a terminal vertex, then $\sigma(t) = t' \in G'$ is also a terminal vertex and $val(t) = val(t')$,
- if $v \in G$ is a non-terminal vertex, then $\sigma(v) = v' \in G'$ is also a non-terminal vertex with $\sigma(lo(v)) = lo(v')$ and $\sigma(hi(v)) = hi(v')$.

Using the recursive definition of the evaluation function f_v, it is easy to show that if G and G' are isomorphic, then $(\forall v \in G)\ (f_v = f_{\sigma(v)})$. Hence testing graph isomorphy suffices to to determine function equivalence.

So far, an OBDD does not give us any advantage in terms of the size complexity of the resulting structure if it is naively constructed via the Shannon expansion. Therefore, we now introduce the notion of a reduced OBDD (ROBDD), which gives us a size advantage for many practically relevant Boolean functions and allows us to make an even stronger statement regarding syntactic testing of function equivalence. An OBDD G is reduced if it satisfies two conditions:

- there is no vertex v with $lo(v) = hi(v)$ (no redundant checks),

- there are no distinct vertices v, v' such that the subgraphs rooted at v and v' are isomorphic.

For any Boolean function there exists an ROBDD, which is unique up to isomorphisms. Furthermore, any other OBDD for this function contains more vertices. On one hand, the size of the resulting ROBDD depends on the Boolean function we wish to represent. While certain classes of functions have been proven to have rather compact ROBDDs (e.g. symmetric n-ary functions have a size of the order $O(n^2)$), others have been shown to be at least exponentially large, notably multiplication circuits. On the other hand, the size of the function depends on the ordering of the variables. The basic problem has been pointed out by Byrant (in [**?**]) in his original paper, and the problem of determining the optimal ordering for an arbitrary function has been shown to be NP-complete (in [2]).

Needless to say, the reduction of an OBDD to form its canonical ROBDD is of key importance for the practical application of BDDs. It is often referred to as "reduce" primitive and runs in $O(\|G\|)$ using an optimal implementation (see e.g. [5]). It is based on three rules of elimination:

1. Eliminate all but one terminal for both possible output values and redirect incoming edges accordingly.

2. Eliminate a duplicate non-terminal v of w if $var(v) = var(w)$ and $lo(v) = lo(w)$ and $hi(v) = hi(w)$ and route all incoming edges to w.

3. Eliminate a duplicate non-terminal v if $lo(v) = hi(v)$ and route all incoming edges to its successor $lo(v)$.

The algorithm given by Byrant is based on an algorithm for checking tree isomorphism. It assigns a label to every vertex from the terminals upwards to the root, such that equivalent nodes are given the same label. Following elimination rule 1, terminal vertices are assigned identical labels if they have the same value. For non-terminal vertices, we enforce elimination rules 2 and 3 by checking the necessary equalities on the node IDs[1]. The superfluous nodes are deleted as part of the run. But in order to give the algorithm in full detail, we need a number of auxiliary ADTs[2] and data structures. Using Pascal as notation, we define a datatype for the vertex as thus:

```
1 TYPE
2     Value = (0, 1, X);
3     VariableIndex = 1..n+1;
4     Vertex = RECORD
5         low, high : Vertex;
6         index     : VariableIndex;
7         val       : Value;
8         id        : Integer;
```

[1] Since the algorithms proceeds bottom-to-top, the IDs of the vertices reached via lo and hi have already been assigned.

[2] ADT ... abstract data-type.

```
9           mark        : Boolean;
10      END;
```

But apart from a data structure for the vertex data, we also need two ADTs: a vertex list and a vertex map, which allows us to access vertices based on a pair of integer IDs. We will assume that we have fully defined the following composite types:

```
1   TYPE
2       VertexStack = RECORD;   { a simple variable sized list }
3           { ... }             { operated as a stack          }
4       END;
5       IntegerPair = RECORD; { used as key for the vertex map }
6           a, b : Integer;
7       END;
8       VertexMap = RECORD;     { a map of the form ℕ² → V }
9           { ... }
10      END;
```

Furthermore, we assume that we have implemented the following procedures and functions to operate on the collection data types:

```
1   PROCEDURE VertexStackPush(s : VertexStack, v : Vertex);
2   FUNCTION VertexStackPop(s : VertexStack) : Vertex;
3   FUNCTION VertexStackEmpty(s : VertexStack) : Boolean;
4   PROCEDURE VertexMapPut(m : VertexMap,
5                          key0, key1 : Integer, v : Vertex);
6   PROCEDURE VertexMapSort(m : VertexMap);
7   FUNCTION VertexMapNextKey(m : VertexMap) : IntegerPair;
8   FUNCTION VertexMapRemove(m : VertexMap,
9                            key : IntegerPair) : Vertex;
10  FUNCTION VertexMapEmpty(m : VertexMap) : Boolean;
```

As a final prerequisite, we require a procedure to gather all vertices of a tree in depth-first order:

```
1   PROCEDURE GatherVertices(v     : Vertex,
2                            vlist : ARRAY[1..n+1] OF VertexStack);
3   BEGIN
4       v.mark ← not v.mark;
5       VertexStackPush(vlist[v.index], v); { add v to the list }
6       IF v.index ≤ n THEN BEGIN  { handle non-terminal vertex }
7           IF v.mark ≠ v.low.mark THEN
8               GatherVertices(v.low, vlist);
9           IF v.mark ≠ v.high.mark THEN
10              GatherVertices(v.high, vlist);
11      END;
12  END;
```

Armed with these tools, we can now describe the reduction algorithm in full detail:

```
1  FUNCTION Reduce(v : Vertex) : Vertex;
2  VAR u        : Vertex;
3      nextid   : Integer;
4      old, key : IntegerPair;
5      subgraph : ARRAY[1..|G|] OF Vertex;
6      vlist    : ARRAY[1..n+1] OF VertexStack;
7      map      : VertexMap;
8  BEGIN
9      GatherVertices(v, vlist); { insert vertices into vlist }
10     nextid ← 0;
11     FOR i ← n+1 DOWNTO 1 DO BEGIN
12         Q ← ∅;
13         WHILE not VertexStackEmpty(vlist[i]) DO
14             u ← VertexStackPop(vlist[i]);
15             IF u.index = n+1 THEN
16                 { for terminals u we use val(u) as key }
17                 VertexMapPut(map, u.value, −1, u);
18             ELSE IF u.low.id = u.high.id THEN
19                     { elimination rule 3 is met }
20                     u.id = u.low.id;
21                 ELSE    { non−terminals u use the key (lo(u),hi(u)) }
22                     VertexMapPut(map, u.low.id, u.high.id, u);
23         { this operation dominates the complexity }
24         VertexMapSort(map);
25         old.a ← −1; old.b ← −1;
26         WHILE not VertexMapEmpty(map) DO BEGIN
27             key ← VertexMapNextKey(map);
28             u ← VertexMapRemove(map, key);
29             IF key = old THEN
30                 { elimination rule 2 is met }
31                 u.id ← nextid;
32             ELSE BEGIN
33                 { no elimination rule applies }
34                 nextid ← nextid + 1;
35                 u.id ← nextid;
36                 subgraph[nextid] ← u;
37                 u.low ← subgraph[u.low.id];
38                 u.high ← subgraph[u.high.id];
39                 old ← key;
40     END; END; END;
41     RETURN subgraph[v.id];
42 END;
```

Clearly, this algorithm is completed within $O(\|G\|)$ loop iterations, because every vertex is visited exactly once. The complexity of a single loop iteration depends critically on the performance of the sorting operation of the vertex map. Hence, it is immediatly clear that a sorting method such as bucket sort should

be used. Unfortunately, the bucket sort algorithm does not lend itself easily as the number of keys is quadratic. The solution is to use an array of lists L (i.e. the buckets) of size $\|G\|$, where every vertex v is in the bucket L[v.low.id], and an array of vertices R with $\|G\|$ elements. It is immediatly clear that only members of the same bucket are candidates for deletion according to elimination rule 2. We then proceed as thus:

- We insert all vertices v $\in$ vlist[i] into the bucket L[v.low.id].
- For all indices j $\in \{1, ..., \|G\|\}$ and v $\in$ L[j]:
  - If R[v.high.id] $= \emptyset$, then v will be retained. We set R[v.high.id] $\leftarrow$v and update the ID and other data structures as before.
  - If R[v.high.id] $=$ w, then v will be replaced by w and we set v.id $\leftarrow$w.id.

In a full implementation, we will use an additional list of indices to keep track of which buckets contain any elements. Consequently, if we have r non-empty buckets we require $O(r)$ steps within the list. Thus the total complexity of the algorithm is $O(\|G\|r)$. In literature, it is commonly acknowledged that r is sufficiently small and the complexity is generally said to be linear.

References

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 1978.

[2] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[3] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[4] C. Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38:985–999, 1959.

[5] D. Sieling and I. Wegener. Reduction of OBDDs in Linear Time. *Inf. Process. Lett.*, 48(3):139–144, 1993.