Implementing Binary Decision Diagram

Minh Pham E. Allen Emerson

Department of Computer Sciences
The University of Texas at Austin
Austin, TX, 78712

Abstract

This paper presents the classic implementation of a simple BDD package. The focus of the implementation is simplicity. Most of the implementation is written in ANSI C. C++ functions are provided for operator overloading.

1 Introduction

Stephen Cook, in his seminal paper, proved that checking the satisfiability of Boolean expressions is NP-complete [5]. Binary Decision Diagrams (BDDs) developed as a solution to this problem in the many practical cases [4]. A BDD is a representation of a Boolean expression in the form of a directed acyclic graph.

Many useful operations, e.g. checking satisfiability, that is NP-complete or co-NP-complete in the general case can be performed on BDDs in less-than-exponential time in many practical cases.

For its merits, BDD is now widely used in a number of fields, particularly model checking and hardware verification. A variety of BDD implementations have emerged.

This paper presents one implementation that is intended for novice to the field. It is a simplified version of the classic pointer-based BDD implementation. The implementation is based on Henrik Andersen's notes [2]; the API is based on that of the BuDDy package [7]. It is written almost entirely in C, with C++ operator functions, to serve as a starting point for more optimized and sophisticated implementations.

2 If-then-else Normal Form

Andersen defines the if-then-else operator $x \rightarrow y_0, y_1$ as follow:

$$x \rightarrow y_0, y_1 = (x \vee y_0) \wedge (x \vee y_1)$$

Then, $t \rightarrow t_0, t1$ would evaluate to true if and only if t is true and t_0 is true, or t is false and t_1 is true. All Boolean operators can be expressed using only the if-then-else operator and the constants 0 and 1 [2] .

An if-then-else normal form is a Boolean expression expressed using only the if-then-else operator and the constants 0 and 1 [2].

It can be shown that any Boolean expression is equivalent to an expression in INF [2]. We show below the equivalent if-then-else (ITE) operator to each of the 16 possible Boolean operators on two variables.

No.	**Boolean**	**ITE operator**	**No.**	**Boolean**	**ITE operator**
0	0	0	8	$f \mp g$	$ITE(f, 0, \bar{g})$
1	$f \wedge g$	$ITE(f, g, 0)$	9	$f \Leftrightarrow g$	$ITE(f, g, \bar{g})$
2	$f \nRightarrow g$	$ITE(f, \bar{g}, 0)$	10	$\bar{g}$	$ITE(g, 0, 1)$
3	f	f	11	$f \Leftarrow g$	$ITE(f, 1, \bar{g})$
4	$f \nLeftarrow g$	$ITE(f, 0, g)$	12	$\bar{f}$	$ITE(f, 0, 1)$
5	g	g	13	$f \Rightarrow g$	$ITE(f, g, 1)$
6	$f \oplus g$	$ITE(f, \bar{g}, g)$	14	$f \bar{\wedge} g$	$ITE(f, \bar{g}, 0)$
7	$f \vee g$	$ITE(f, 1, g)$	15	1	1

Figure 1: Implementation of Boolean functions in terms of ITE operators

By recursively replacing each variable in the INF through Shannon expansion, we end up with a tree of expressions in INF, also called a decision tree.

By merging all expressions in the decision tree with identical right-hand-side portions, we end up with a directed acyclic graph. Such a graph is a Binary Decision Diagram.

3 Binary Decision Diagram

A Binary Decision Diagram (BDD) is a directed acyclic graph, with each non-terminal node representing a Boolean variable. Two terminal nodes, 1 and 0, represent respectively the Boolean functions 1 and 0.

Each non-terminal node has two edges. The high edge corresponds to the then part, and the low edge corresponds to the else part of an if-then-else normal form. If we choose an ordering of variables such that from the root to the terminal node, each variable appears in the same orders on all paths, then the BDD is ordered (an OBDD).

We can reduce the size of an OBDD further by merging all identical nodes and eliminating all redundant tests. An OBDD that has been thus reduced is a reduced OBDD (ROBDD). ROBDDs are canonical, meaning for any Boolean function f there is one and only one ROBDD that represents it. We can thus check in constant time whether an ROBDD is constantly true or false, whereas such an operation is exponential for Boolean expressions [2].

4 Construction and Manipulation of BDD

To construct a BDD that represent a Boolean formula, we will begin with small BDDs that each represents either an input variable or a truth value. We will then build up the BDD by applying Boolean operations on BDDs themselves.

In this section we present the representation of a BDD, constructing the most basic elements of the BDD, and the two main operations on BDD: RESTRICT and ITE.

4.1 Using Node to Represent Graph

One way to think of a BDD is a tree composed of nodes that represent boolean variables. Such a tree would begin at some root node and grow down to one or both sinks. Such a way of thinking is useful in many circumstances. However, for the purpose of implementation, we can equate the BDD and its root node, since we can access all parts of a bdd by traversing down from its root node. This is similar to a linked list. In this implementation, instead of having a node struct and a separate BDD struct that simply holds the value of the root node, we will call nodes BDDs.

We realize that this naming scheme may cause some confusion. However, we would like to emphasize the fact that every node can also be thought of as a BDD that begins at that node.

```
struct bdd {
public:
    int index;          // lower index = closer to the top
    bdd* high;          // pointer to the THEN bdd
    bdd* low;           // pointer to the ELSE bdd
    bdd* next;          // pointer to the next bdd in the bucket
```

Figure 2: Data members of the BDD structs

The *index* variable represents the subscript in the variable ordering $x_0 < x_1 < x_2 < ....$ By convention, we designate the nodes in the same tree with the lower index to be nearer to the top.

The *high* and *low* variables, respectively, point to the *then* and *else* part of an if-then-else operator, with the *if* clause being the variable represented by this node. This variable is also known as the splitting variable.

The *next* variable points to the next node in a hash bucket. We discuss hashing in the next section.

4.2 MAKENODE and ITHVAR

In order to maintain canonicity, there must be no identical nodes in the BDD. Two nodes are identical if they represent the same variable, and point to the same high and low nodes. We use a hash map to speed up the look-up process. By hashing on their variable indices and pointer addresses of the their child nodes, two identical nodes would necessarily hash to the same value.

In case two different nodes also hash to the same value, we handle collision by chaining the colliding node onto existing nodes to form buckets. Thus, each node has a *next* pointer to the next node in the bucket.

The NODEHASH macro uses the variable *bddnodesize.* This variable starts at 0 and can be changed at initialization of the package. Higher *bddnodesize* consumes more memory (since it is the size of the array underlying the lookup table), but reduces the possibility of hash collision.

The function ITHVAR returns a BDD that represents the ith variable: it has one node that points to the 1 (high) and 0 (low) sinks. ITHVAR is

```
#define PAIR(a,b)       ((unsigned int)((((unsigned int)a)+((
    unsigned int)b))*(((unsigned int)a)+((unsigned int)b)+((
    unsigned int)1))/((unsigned int)2)+((unsigned int)a)))
#define TRIPLE(a,b,c)   ((unsigned int)(PAIR((unsigned int)c,PAIR
    (a,b))))
#define NODEHASH(lvl, l, h)      (TRIPLE((lvl),(l),(h)) %
    bddnodesize);
```

Figure 3: Hash functions realized as C macros

```
bdd** BDDTable;

bdd* bdd_makenode(int var, bdd* high, bdd* low) {
     unsigned int hash = NODEHASH(var, high, low);
     // no matching bucket found
     if (BDDTable[hash] == 0) {
          BDDTable[hash] = new bdd(var, high, low);
          return BDDTable[hash];
     } else {
          bdd* node = BDDTable[hash];      // first node in bucket
          // search bucket for the matching node
     while (node->next && (node->index != var || node->high !=
         high || node->low != low))
               node = node->next;
          // no matching node in bucket
          if (node->index != var || node->high != high || node->
              low != low) {
               // make new node and add to bucket
               bdd* newNode = new bdd(var, high, low);
               node->next = newNode;
               return newNode;
          } else {
               // found matching node in bucket
               return node;
          }
     }
}
```

Figure 4: MAKENODE implemented in C/C++

implemented in term of MAKENODE to ensure that the BDD it returns is unique.

```
bdd* bdd_ithvar(int i) {
    numvars++;
    BDDTable[0]->index = numvars;
    BDDTable[1]->index = numvars;
    return bdd_makenode(i, bdd_true(), bdd_false());
}
```

Figure 5: ITHVAR implemented in C/C++

We must readjust the index of the sinks everytime a new variable is requested. This adjustment is to preserve correct behavior of querying operations, as discussed later in this paper.

The next two sections present the implementation of the two most important operations in constructing BDDs, RESTRICT and ITE.

4.3 RESTRICT

RESTRICT computes a BDD that results from assigning all instances of the variable *var* with the truth value *val*. RESTRICT also makes use of MAKENODE to ensure the BDD returned is unique.

```
bdd* bdd_restrict(bdd* subtree, int var, bool val) {
    if (subtree->index > var) {
        return subtree;
    } else if (subtree->index < var) {
        return bdd_makenode(subtree->index,
                    bdd_restrict(subtree->high, var, val),
                    bdd_restrict(subtree->low, var, val));
    } else { /* (subtree->index == var) */
        if (val) {
            return bdd_restrict(subtree->high, var, val);
        } else {
            return bdd_restrict(subtree->low, var, val);
        }
    }
}
```

Figure 6: RESTRICT implemented in C/C++

4.4 ITE

The ITE operation computes and returns the BDD that is the result of applying the if-then-else operator to three BDDs that serve as the *if*, *then*, and *else* clauses, respectively.

```
bdd* bdd_ite(bdd* I, bdd* T, bdd* E) {
    // Base cases
    if (I == bdd_true())  return T;
    if (I == bdd_false()) return E;
    if (T == E)           return T;
    if (T == bdd_true() && E == bdd_false()) return I;

    // General cases
    // splitting variable must be the topmost root
    int split_var = I->index;
    if (split_var > T->index) { split_var = T->index; }
    if (split_var > E->index) { split_var = E->index; }

    bdd* Ixt = bdd_restrict(I, split_var, true);
    bdd* Txt = bdd_restrict(T, split_var, true);
    bdd* Ext = bdd_restrict(E, split_var, true);
    bdd* pos_ftor = bdd_ite(Ixt, Txt, Ext);

    bdd* Ixf = bdd_restrict(I, split_var, false);
    bdd* Txf = bdd_restrict(T, split_var, false);
    bdd* Exf = bdd_restrict(E, split_var, false);
    bdd* neg_ftor = bdd_ite(Ixf, Txf, Exf);

    bdd* result = bdd_makenode(split_var, pos_ftor, neg_ftor);
    return result;
}
```

Figure 7: The ITE operator implemented in C/C++

ITE terminates when one of the base cases is encountered. Otherwise, it recursively descend down the BDD tree while building itself up using the Shannon cofactor expansion.

```
bdd* bdd_and(bdd* lhs, bdd* rhs) {
    return bdd_ite(lhs, rhs, bdd_false());
}
```

Figure 8: The AND operator implemented in term of ITE

5 Querying BDD

5.1 SATCOUNT

SATCOUNT returns the number of truth assignments that satisfy the Boolean expression represented by the BDD. Since this number can be very large for some BDDs, a return type of double is used.

```
double bdd_satcount_rec(bdd* subtree) {
    if (subtree == bdd_false()) {
        return 0;
    } else if (subtree == bdd_true()) {
        return 1;
    } else {
        bdd* low = subtree->low;
        bdd* high = subtree->high;
        double size = 0.0, s = 1.0;
        s = pow(2.0, (low->index) - (subtree->index) - 1);
        size += s * bdd_satcount_rec(low);
        s = pow(2.0, (high->index) - (subtree->index) - 1);
        size += s * bdd_satcount_rec(high);
        return size;
    }
}

double bdd_satcount(bdd* subtree) {
    return pow(2.0, (subtree->index) - 1) * bdd_satcount_rec(
        subtree);
}
```

Figure 9: SATCOUNT implemented in C/C++

SATCOUNT assumes that the index of each node will correspond to the variable subscript in the variable ordering. This is true for non-terminal nodes, since their index is correctly assigned when they are constructed using

ITHVAR. Terminal nodes are a special case: their index is here assumed to be always one larger than the largest subscript in the variable ordering, but that largest subscript cannot be determined in advance.

Therefore, we need to readjust the index of the terminal nodes every time we check out an input variable.

5.2 SATONE

SATONE returns one truth assignment, expressed as a BDD, that satisfies the given BDD. We see that a truth assignment can be found for the variable at every node that would satisfy the given BDD, except when that node is the 0 sink.

From this observation, we need only descend down the BDD towards the 1 sink, by avoiding all paths that lead to the 0 sink.

```
bdd* bdd_satone_rec(bdd* subtree) {
    assert(subtree != 0);
    if (subtree == bdd_false() || subtree == bdd_true()) {
        return subtree;
    } else if (subtree->low == bdd_false()) {
        return bdd_makenode(subtree->index, bdd_false(),
            bdd_satone_rec(subtree->high));
    } else { /* subtree->high == bdd_false() */
        return bdd_makenode(subtree->index, bdd_satone_rec(
            subtree->low), bdd_false());
    }
}

bdd* bdd_satone(bdd* subtree) {
    assert(subtree != 0);
    if (subtree == bdd_false())
        return 0;
    else
        return bdd_satone_rec(subtree);
}
```

Figure 10: SATONE implemented in C/C++

Related Works

There is a very large amount of literature on efficiency topics related to BDDs. Listed below are only some that the authors believe would be of interest to a novice in BDD-related topics.

Bryant et al. discussed various efficiency improvements that are now the standard in a classic BDD implementation. These include complemented edge, hashing, memoization, caching and garbage collection [3].

It is known that the size and complexity of BDDs, and consequently, the efficiency of operations on them, depend heavily on the variable ordering chosen. The variable ordering problem is discussed at length in many textbooks, including an excellent one by Christoph Meinel and Thorsten Theobald [1].

Another approach to BDD implementation that discard the use of pointers in favor of simple arrays is discussed in Geert Jansen's paper. This approach is shown to outperform a pointer-based approach in memory and CPU efficiency, at the expense of more complicated garbage collection [6]. The BuDDy package follows this approach [7].

Conclusion

Presented in this paper is a highly simplified pointer-based implementation of a BDD package. For the sake of simplicity, error-checking and garbage-collection, as well as many efficiency considerations, were omitted from this implementation. These, however, would be important considerations in a serious BDD package, which the authors hope an invested reader would now be more inclined to explore.

References

[1] *Algorithms and Data Structures in VLSI Design.* Springer-Verlag, 1990.

[2] Henrik Reif Andersen. An introduction to binary decision diagrams. In *In Lecture notes for Efficient Algorithms and Programs.*

[3] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEE Design Automation Conference.*

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*.

[5] Stephen Arthur Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*.

[6] Geert Jansen. Design of a pointerless bdd package. Technical report, IBM T.J. Watson Research Center, 2001.

[7] Jørn Lind-Nielsen. Buddy: a bdd package. `http://sourceforge.net/projects/buddy/`.