DJBDD

Contents

Introduction

This document gives a summary explanation of some topics of the Binary Decision Diagram library developed in Java 7.

Me, Diego J. Romero-López is the author of this BDD package.

This document is short (in-progress) manual working as a launchpad for developers that want to use this package.

Highlights

We don't use a explicit *reduce operation*. Our BDD is reduced when constructed. So we enforce the properties of **uniqueness** and **non-redundantness** of the vertices when constructing the BDD.

Our BDDs are based on the tables T and U described in [1]. Thus, the operations implemented in this library are described in [1]. We could say that [1] is a big influence of our library is in this paper of Andersen. You are encouraged to read it.

What be use is some heuristic reducing methods that try to guess the best variable ordering, that is, the variable ordering that makes the BDD minimal.

Data structures

Vertex

The vertex is composed of the following attributes:

- **`index`**: an unique index for this vertex. The way we get the indices, they are not repeated anytime.
- **`variable`**: an index to the variable that contains this vertex.
- **`low`**: a reference to the low vertex.
- **`high`**: a reference to the high vertex.
- **num_parents**: reference counting of this vertex.
- **num_rooted_bdds**: number of BDDs that have this vertex as root

BDD Tree

The basic structure we use is a vertex tree with some enhancements.

This tree contains three hash tables that reference to vertices. This way, we will have to use vertex reference counters before deciding on deleting the vertex.

Table T

The first hash table contains a hash whose keys are the indices of each vertex and its values, references to its vertices.

It is used to keep count of the vertices and as vertex cache, easing the transversal of the tree.

Table U

The uniqueness table. This hash has as key a concatenation of the attributes that makes unique one vertex, that is:

- `variable`: variable index of the vertex.
- `lowid`: id of the low descendant. If our vertex is a leaf, thus making the high descendant a null, we replace it with a constant.
- `highid`: id of the high descendant. If our vertex is a leaf, thus making the high descendant a null, we replace it with a constant.

This table is used to enforce the uniqueness of each vertex when adding new vertices in the construction of the BDD or when using the apply operation.

Table V

Contains the vertices in groups by variables. That is, the key is the index variable and the value is a set of vertices.

At this moment this structure is not used for anything. We have some plans to use it in a heuristic variable ordering algorithm.

Operations

Construction

The construction uses the apply operation and a grammar parser[1] which is faster than the recursive process. This algorithm constructs the BDD making use of the apply operation between the internal formulas and variables of the formula. The basic BDDs that contains one variable (or a complemented variable) will be constructed with a private static factory with the name of `BDD.factoryFromVarible`.

If you want to change that, set `BDD.USE_APPLY_IN_CREATION` to `false`.

Apply

The apply algorithm is based on the implementation given by [3].

```
# Constructs a new unique key to one vertex
makeUniqueKey(var_index, low, high):
        if(low==null && high==null):
            return var_index+"-"+"NULL"+"-"+"NULL";
        return var_index+"-"+low.index+"-"+high.index

# Adds a unique vertex to the hash tables
addNewVertex(var_index, low, high):
        # Gets the next key, increments a global counter and gets its value
        index = getNextKey()
        v = new Vertex(index, var_index, low, high)
        T[index] = v
        U[makeUniqueKey(var_index, low, high)] = v
        V[var_index].add(v)
        return v

# Adds a non-redundant vertex to the hash tables
addVertex(var, low, high):
        vertexUniqueKey = makeUniqueKey(var_index, low, high)
        if(uniqueKey  in U):
            return U[uniqueKey]
        return addNewVertex(var, low, high)

# Apply algorithm to two vertices
applyVertices(Vertex v1, Vertex v2):
```

1 This parser uses Antlr3 (http://www.antlr.org/wiki/display/ANTLR3/ANTLR+3+Wiki+Home)

```
# Hash key of the computation of the subtree of these two vertices
String key = "1-" + v1.index + "+2-" + v2.index
if( key in G ):
    return G[key]

if(v1.isLeaf() && v2.isLeaf()):
        # op is the boolean operation between two leaf vertices
        # that is true and false
    if(op(v1,v2)):
        return True # Constant leaf vertex true
    return False  # Constant leaf vertex false

var = -1
low = null
high = null
// v1.index < v2.index
if (!v1.isLeaf() and (v2.isLeaf() or v1.variable < v2.variable)):
    var = v1.variable;
    low = applyVertex(v1.low, v2)
    high = applyVertex(v1.high, v2)
else if (v1.isLeaf() or v1.variable > v2.variable):
    var = v2.variable
    low = applyVertex(v1, v2.low)
    high = applyVertex(v1, v2.high)
else:
    var = v1.variable
    low = applyVertex(v1.low, v2.low)
    high = applyVertex(v1.high, v2.high)

// Respect the non-redundant property:
// "No vertex shall be one whose low and high indices are the same."
if(low.index == high.index):
    return low

// Respect the uniqueness property:
// "No vertex shall be one that contains same variable,
// low, high indices as other."
Vertex u = addVertex(var, low, high)
G[key] = u
return u;
```

```
# Main call to apply algorithm
apply(operation, bdd1, bdd2):
      # Operation is global
      # Cache to avoid repeated computations
      G = {}
      String function = bdd1.function + " "+operation+" "+bdd2.function
      # Fill this.T with vertices of bdd1 and bdd2
      root = appyVertex(bdd1.root, bdd2.root)
      # Construction of new BDD
      bdd = new BDD(function, root)
      return bdd
```

Restrict

Restrict operation assigns boolean values to some variables, thus restricting the path from the tree and obtaining a new one.

```
# Get a new root vertex of a BDD based on this BDD
# with a boolean assignement on some variables.
restrictFromVertex(v, assignement):
        if(v.isLeaf()):
            # There is only one true (index 1) and one false vertex (index 0)
            return T[v.index]
        if(v.variable in assignement):
            boolean value = assignement[v.variable]
            if(value):
                return restrictFromVertex(v.high, assignement)
            else:
                return restrictFromVertex(v.low, assignement)
        else:
            low = restrictFromVertex(v.low, assignement)
            high = restrictFromVertex(v.high, assignement)
            if(low.index == high.index)
                return low
            return addVertex(v.variable, low, high)

# Get a new BDD based on this BDD with a boolean assignement on some variables.
restrict(bdd, assignement):
        restrictedBDD = restrictFromVertex(bdd.root, assignement);
```

```
        rfunction = bdd.function
        for(pair : assignement.pairs()):
            variable = VARIABLES[pair.key]
            value = pair.value
            rfunction = rfunction.replace(variable, value)
        return new BDD(rfunction, restrictedBDD)
```

Swap

This operation swaps tow levels of the tree. Our aim is to obtain some orphan vertices, thus they can be deleted.

This operation is ignored in most of the papers and they point to [6] for more information. This paper can be complemented with the notes of the same author from [7]. I recommend read these papers and later, read another one that will clear completely your doubts about the implementation [8].

Our implementation is similar to [8] so you should understand it without any problems.

```
swapVertex(Vertex v, int varJ):
        swapWasMade = false
        varI = v.variable

        low = v.low
        high = v.high

        A = null
        B = null
        if (!low.isLeaf()):
            A = low.low
            B = low.high
        else:
            A = low
            B = low

        C = null
        D = null
        if (!high.isLeaf()) :
            C = high.low()
            D = high.high()
        else:
            C = high
            D = high
```

```
newLow = null
newHigh = null

// Case a:
if (low != null && low.variable == varJ &&
          (high == null || high.variable != varJ)):
    newLow = addWithoutRedundant(varI, A, C)
    newHigh = addWithoutRedundant(varI, B, C)
    setVertex(v, varJ, newLow, newHigh)
    swapWasMade = true
// Case b:
else if ((low == null || low.variable != varJ) &&
          (high != null && high.variable == varJ)):
    newLow = addWithoutRedundant(varI, A, B)
    newHigh = addWithoutRedundant(varI, A, C)
    setVertex(v, varJ, newLow, newHigh)
    swapWasMade = true
// Case c:
else if ((low != null && low.variable == varJ) &&
          (high != null && high.variable == varJ)):
    newLow = addWithoutRedundant(varI, A, C)
    newHigh = addWithoutRedundant(varI, B, D)
    this.setVertex(v, varJ, newLow, newHigh)
    swapWasMade = true
// Case d:
else if ((low == null || low.variable != varJ) &&
          (high == null || high.variable != varJ)):
    swapWasMade = false
// Case e:
else if ((low == null || low.variable != varJ) && high == null):
    swapWasMade = false

return swapWasMade
```

```
swap(int level):

    // If is the last level, ignore
    if(level == variables.size()-1)
        return false

    variableI = variables.getVariableInPosition(level)
    variableJ = variables.getVariableInPosition(level+1)

    boolean swapWasMade = false

    verticesOfThisLevel = V[variableI]
    for(Vertex v : verticesOfThisLevel):
        swapWasMade = swapWasMade || swapVertex(v, variableJ)

    variables.swap(variableI, variableJ);
    return swapWasMade
```

Garbage collection

The garbage collection can be called using `BDD.gc`. Note that this garbage collection compacts table T, erasing the pairs `<index,Vertex>` where the vertex is not referenced by other vertices and is not root of any BDD.

The deletion is applied for each vertex while there has been some deletions. Some upgrades to this methods are

- Parallelizing the vertex loop.
- Keeping track of the ancestors of each vector for a more efficient way of repeating the vertex loop.

Dynamic variable ordering

Our methods are based on the variable swapping operation. They are heuristics that try to find the best possible variable ordering, that is, the variable ordering that makes the number of vertices of the BDD minimal.

This is a NP complete problem and there are many possible solutions. We have implemented these algorithms:

- **Sifting Algorithm**, described by Rudell in [6].
- **Windom Permutation Algorithm**, main Rudell method's competitor [6].
- **Random Swapper Reduction**: iterates and in each iteration swaps to levels of vertices.
- **Genetic Algorithm**, based on [9].
- **Memetic Algorithm**, based on the Genetic Algorithm from [9].
- **Iterative Sifting**, our original method, described in [10].

Open tasks

There are some tasks that have not been finished.

Saving/Loading BDDs

Developing a file format to store and load BDDs from disk. Nowadays, the BDDs must be hard-coded, introduced as DIMACS files or C expressions.

Parallel algorithm

One of my aims is making this implementation run on shared memory environments.

High-efficient implementation

Other of our objectives is implementing this same library in C++ or D to make the BDD construction and dynamic reduction more efficient.

Bibliography

[1] An introduction to binary decision diagrams, Henrik Reif Andersen.

[2] Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, Randal E. Bryant. Carnegie Mellon University.

[3] Binary Decision Diagrams. Fabio Somenzi. Department of Electrical and Computer Engineering. University of Colorado at Boulder.

[4] Efficient implementation of a BDD package, Karl S. Brace, Richard L. Rudell, Randal E. Bryant.

[5] Implementation of an Efficient Parallel BDD Package. Tony Stornetta, Forrest Brewer.

[6] *Dynamic variable* ordering for ordered binary decision diagrams. Richard L. Rudell 1993.

[7] BDDs: Implementation Issues & Variable Ordering. Richard Rudell 1993.

[8] Incremental Reduction of Binary Decision Diagrams. R. Jacobi, N. Calazans, C. Trullemans.

[9] Genetic Algorithms for the Variable Ordering Problem of Binary Decision Diagrams, Foundations of Genetic Algorithms (pp. 1-20). Springer Berlin Heidelberg, 2008. W. Lenders & C. Baier.

[10] Iterative Sifting: A new approach to reduce BDD size. Diego J. Romero-López & Elena Ruiz-Larrocha. TBA.