Documentation for the implementation of binary decision diagram

Olivér Izsák
Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
xizsak@stuba.sk

4. máj 2021

1. Introduction

Binary decision diagram is data structure that is responsible for interpreting a given boolean function. In my program I created my own implementation of a binary decision diagram. My implementation consist of creating the diagram, reducing the diagram and using the diagram to get the value of a given a function.As my input boolean function I choose a boolean expression.

1.1 Boolean function (BF function)

The input function into my BDD is a boolean expression which can consist of the following characters : +(addition), . (multiplication), {a,b,c...} (the letters of the english alphabet). I have not implemented parentheses' „()" due to lack of time, since it would complicate the implementation. Additionally my implementation uses the uppercase letters as positive value and lowercase as negative eg.:
A=A,
a=!A
A=1,
a=0
Also my BF class accepts a String containing the boolean expression, which has to be in Disjunctive Normal Form. After accepting the expression it modifies the String using 2 other methods: createList(), andcheckVariables(). createList method creates an arraylist of strings and inserts parts of the boolean function which are separated by a + sign. Eg.: a+b.d+z.e, where „a" would be put into arrayslot 0, „b.d" into arrayslot 1, and „z.e" to arrayslot 2.
Meanwhile checkVariables method goes through the string containing the boolean expression and takes out all the unique variables in it, and puts it into an string arraylist which will hold the unique variables.

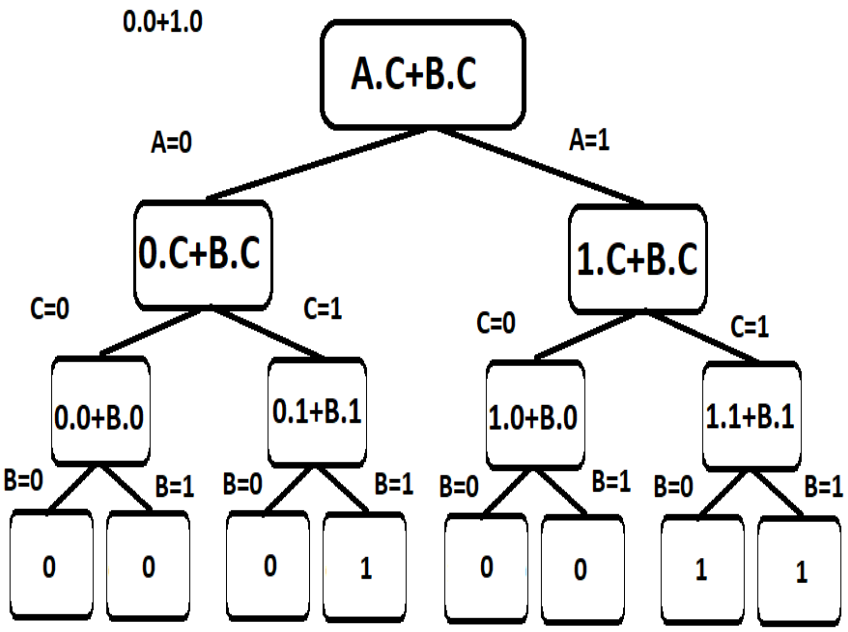
1.2 Binary Decision Diagram

My implementation of the BDD object has the following variables:

- int amountOfNodes – the amount of individual nodes
- int amountOfVariables – the amount of unique variables
- Node root – the root of the binary decision diagram.
- BF funct – the boolean expression that was given as input into the BDD.
- Node node0 – A specific node which has null value for left and right child, and only holds the value 0 as a string.
- Node node1 - A specific node which has null value for left and right child, and only hold the value 0 as a string.

2. BDD_create

The function BDD_create is responsible for creating all the nodes of the binary decision diagram using a recursive function (BBDCreateHelper), which accepts a string (containing a reduced and modified boolean function), a Node which is the parent node,a BF object that contains the current variable that is being modified to 1 or 0, and a boolean which determines whether the new node being made is left or right.Additionally every node has a depth, and when the last depth is reached, that is where the final value is stored. Instead of storing the string containing a modified expression consisting of zeroes and ones, it will be converted into a string containing a 1 or a 0 (See picture 1.). The BBDCreatehelper function uses some other functions to create the diagram. Such as the shannonDecomposition method. Which basically goes through a given boolean expression and replaces the current variable with a 0 on the left side and 1 on the right side. For the last part where the nodes with the last depth are replaced with 0 or 1 only, 2 other methods are helping. One is getAdditions, which is responsible for taking apart a given boolean expression and sorts the variables into an array depending whether there is a multiplication between the variables or an addition. While the other method is stringToInt, which converts a specific boolean expression where all variables have been replaced by ones and zeroes into a regular equation (Eg.: A.b+c -> 1.1+0) and calculates it, this way it can calculate whether the result of the boolean expression in this case is 0 or 1.



Picture:. 1: Nodes.in the Binary Decision Diagram after creation

3. BDD_reduce_

The method BDD_create is responsible for reducing the binary decision diagram by deleting all duplicate nodes and also replacing the nodes with the last depth with node0 and node1, depending on their value.

The method returns the amount of nodes deleted in the reduction process.For the BDD reduction I use 2 other methods.

1.Method is the postOrderTraversalReduction, which accepts the following parameters: the root of the BDD, an arraylist of string for storing boolean expression combinations, an integer representing the depth of the current node and an arraylist of nodes, for storing the current node we are checking.This recursive method works the following way:

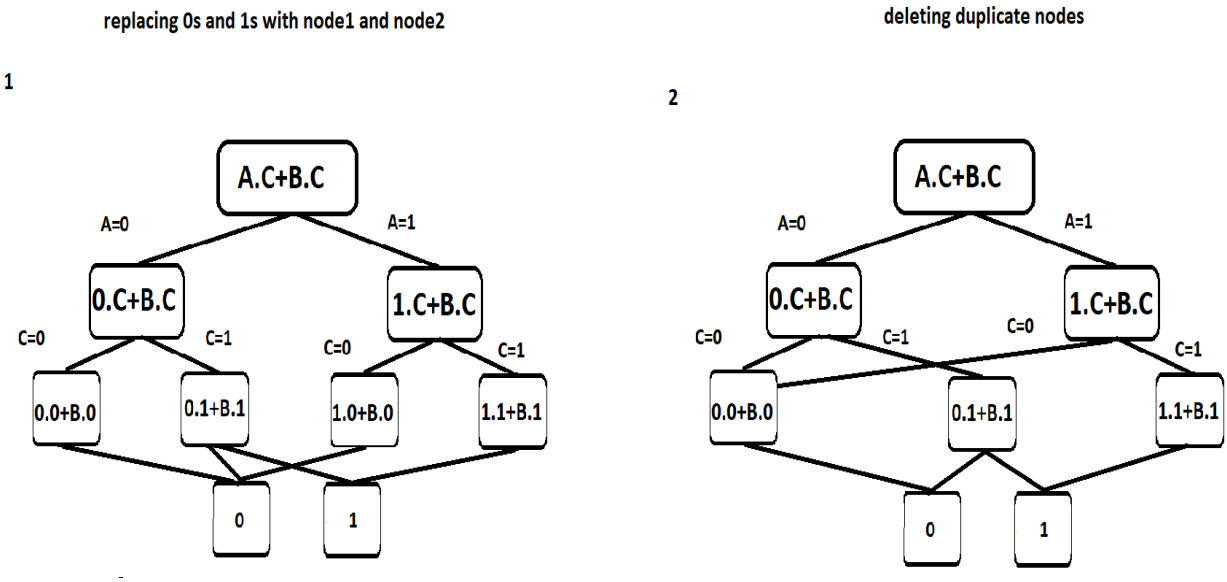
The first if statement does the following - it goes through the BDD in post order traversal checking the the string value of a last depth node.If the node is a left one , then it takes its neighbour (right node of left's parent) and joins them together in a string variable called combination. After that it checks whether this combination has already been checked, by inspecting the arraylist of strings whether it contains the same combination (it can either be 00,01,10,11 – in the last depth nodes only). If the list doesn't containg the combination then it puts it in the list, and goes to the next nodes, this time checking whether the new nodes has identical combination, if they have identical combination, then this new found node if deleted and is replaced by the first one we found. And this goes on recursively until all nodes combinations have been found and replaced. 2. The second if statement does the same thing but for nodes that have a depth below n and n-1 (since the first if statement only works for the last depth nodes, and their parent nodes).

The 2nd if statement also uses another outside method called midFunctionReduction, which is a similiar method to stringToInt method, however this one works on boolean expression that still contains variables. It goes through the given boolean function and reduces it or calculate its value Eg.: 1.0+A.1 will be reduced to A. 1.1+A.B.C ->will be reduced to 1, it doesn't matter the value of A,B,C, since 1.1 means the outcome will always be 1. Same goes for 0.A.B.C + 0.A -> will be calculated as 0. And using this function it deletes all the other duplicate of nodes that have lower depth than n and n-1. (See picture 2. and screenshot 1.)

2.Method which is postOrderTraversalReductionOAZ, is repsonsible for replacing all the nodes with last depth, which are containing either 1 or a 0, to node1 and node0 respectively. This saves memory, since there are only 2 possible outcomes at last depth. There is no need to have more nodes than two.

```
public void postOrderTraversalReduction(Node node,ArrayList<String> nodes,int n,ArrayList<Node> crn)
{ if (node == null)
    return;
  postOrderTraversalReduction(node.getLeft(), nodes, n, crn);
  postOrderTraversalReduction(node.getRight(), nodes, n, crn);
  if (node.getDepth() == amountOfVariables && n == node.getDepth())
  {
    String combination = node.getFunction();
    if (node.isLeft())
    {
      combination = combination + node.getParent().getRight().getFunction();
      if (nodes.get(nodes.size() - 1).equals("|"))
      { if (nodes.get(nodes.size() - 2).equals(combination))
        { if (node.getParent().isLeft())
          { node.getParent().getParent().setLeft(crn.get(0));
          amountOfNodes -= 1; }
          else
          { node.getParent().getParent().setRight(crn.get(0));
            amountOfNodes -= 1;
          }
        }
      }
      else if (!(nodes.contains(combination)))
      { nodes.add(combination);
        crn.add( index: 0,node.getParent());
        nodes.add("|");
      }
    }
    }
  }
  else if (node.getDepth() == n && node.getDepth()!=amountOfVariables-1)
  { String combination = node.getFunction();
  String temp = midFunctionReduction(combination);
  combination = temp;
  if (nodes.get(nodes.size() - 1).equals("|"))
    { if (nodes.get(nodes.size() - 2).equals(combination))
      { if (node.isLeft())
        { node.getParent().setLeft(crn.get(0));
          amountOfNodes -= 1; }
        else
        { node.getParent().setRight(crn.get(0));
          amountOfNodes -= 1;
        }
      }
    }
    else if (!(nodes.contains(combination)))
    { nodes.add(combination);
      crn.add( index: 0,node);
      nodes.add("|"); }
  }
}
```

Screenshot 1. :postOrderTraversalReduction recursive method.



Picture 2. : Illustration for what the BDD_reduce method does.

4. BDD_USE

This method is responsible for going through the already created BDD and inserting a combination of ones and zeroes representing the values of the variables in the already defined boolean expression and calculating the result of that specific combination. The method accepts a binary decision diagram parameter which has a boolean expression already defined in it, and accepts a string parameter which consists of ones and zeroes. The order of the ones and zeroes is the same as the order of unique variables appearing in the defined boolean expression eg.: If your boolean expression was A+C+B.E.D+A then typing 01001 means, A=0,C=1, B=0, E=0, D=1. If the given string value is not equal to the unique variables in the boolean expression, or if a char other than 1 or 0 is present in the string, the method will return -1, meaning there was an error in the string. However if the input is correct it will return the outcome of that boolean expression 1 or 0.

```
public int BDD_use(BDD bdd,String value)
{ Node node =bdd.getRoot();
  if(amountOfVariables==value.length())
  { for (int i = 0; i < amountOfVariables; i++)
    { if (value.charAt(i) == '0')
      {
        node = node.getLeft();
      }
      else if (value.charAt(i) == '1')
      {
        node = node.getRight();
      }
      else
      {
        System.out.println("error, use only numbers 1 or 0");
        return -1;
      }
    }
  }
  else {
    System.out.println("Error, wrong input for string");return -1;}

  if(node.getFunction().equals("1")) {return 1;}
  else
     return 0;
```

Screenshot 2.: BDD_use method

5. Testing

Testing was done in 3 different ways:

1. manual testing. Taking a 5 variable boolean expression, creating the binary decision diagram, reducing it, then calling BDD_use on every possible combination of ones and zeroes, and then checking each result manually.

Result:

```
Boolean function: A.B+C.E+D
Amount of unique variables: 5
Amount of nodes created: 63
Reduction:
Amount of unique variables: 5
Amount of nodes created: 12
Use:
Amount of errors: 0
Node reduction percentage: 80.95238095238095%
```

Screenshot 3.: manual testing

2. random testing. This test was done with 13 variables and 2000 different binary decision diagrams were created, and each one had a unique random generated boolean expression given as parameter.After the BDD_create, all of them were reduced by BDD_reduce. After the reduction each reduced diagram went through a loop which called the BDD_use method on every possible combination of ones and zeroes for the given boolean expression, and compared the outcome of the reduced diagram with the diagram that was created first.

Result:

```
Random boolean expression created: J+G.a+D+I.M.I.H+E.E.M.b.M.D+I.H+E+F+E+K+a+K.M.I.c+L.a.E+L.E.M+D.D.J.a.M.K+a.b+F.E.D.I.L.J.G+c
Amount of nodes: 16383.0 Amount of nodes after reduction: 37
reduction percentage: 99.774%
Random boolean expression created: I.a+I+c+L+c+g.a.f.J+b.a.g+d.H.I.c.d.e.a+a+b.b+I+b.f.g+L+J.c.M+e.J.f.e+f.a+K
Amount of nodes: 16383.0 Amount of nodes after reduction: 36
reduction percentage: 99.78%
Random boolean expression created: L+L+a.M.J.J.a.a.J+K.M.M+M.H.b.J+H.H.L+b+J.c+K.I.c.K+d.M.b+F.a+a+c.H+b+b+G.e
Amount of nodes: 16383.0 Amount of nodes after reduction: 54
reduction percentage: 99.67%
Reduction successful
Node reduction Percentage: 99.721%
```

Screenshot 4.: random testing all methods

3. random testing with time. This test was also done with 13 variables and 2000 different binary decision diagrams created with unique random generated boolean expressions.Then BDD_reduce called on each of them. At the same time measuring the time for both the BDD_create and the BDD_reduce method. I did not measure the BDD_use method since it's

comparatively takes a lot less time than creation or reduction, since its just normal search function.
It can be seen in the result that reduce methods takes more time then create methods, since when it comes to reduction the algorithm goes through the diagram many times.
Result:

```
  All create time :125.40034700000021
Average create time: 0.06270017350000011
All reduce time: 1132.4540361000013
Average reduce time: 0.5662270180500006
Average total time: 0.6289271915500008
Total time spent : 1257.8543831000015
```

Random testing with time

6. Conclusion

In conclusion my implementation of the binary decision diagram seems efficient when it comes to node reduction. Since it's average node reduction is above 99% if working with large amount of variables. However when it comes to smaller variables this percentage obviously becomes smaller. Since more variables in the boolean expression means more possibilities for duplicate nodes in the diagram.
One thing that could have been done to make the reduction better was to add an if statement for specific boolean expression when a boolean expression contains a large amount of multiplication and small amount of addition, or no additions at all, such as : A.B.C.D. Where if any one the variables is not 1 then the outcome is 0 and the outcome is only 1 if every variable is equals to 1.Since I have not done a specific if statement for this. Expressions like that will take a lot of time to proccess for my algorithm, even though with the if statement added it coud be done a lot faster.
One other mistake was that I didn't have a good and efficient idea to check the solutions for each and every outcome when BDD_use method was applied, that is why I only compared it the reduced solutions to the first created one.