Data structure and algorithms

documentation for assignment 2

Table of Contents

Introduction

Theme for assignment: Binary Decision Diagrams

Task: Create a program, where a data structure called BDD (Binary Decision Diagram) with a focus for representation of Boolean functions can be created.

Functions that I have done in my task:

BinaryDecisionDiagram BDD_create(String boolean_function, String order);

BinaryDecisionDiagram BDD_create_with_best_order(String boolean_function) ;

char BDD_use(BinaryDecisionDiagram bdd, String values);

Programming language : Java.

Code editor : IntelliJ IDEA.

Processor: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz

RAM: 16.0 System type: 64-bit operating system, x64-based processor

OS: Windows 11

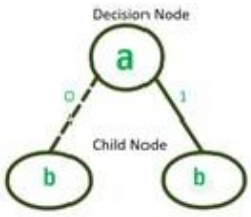
Binary Decision Diagram

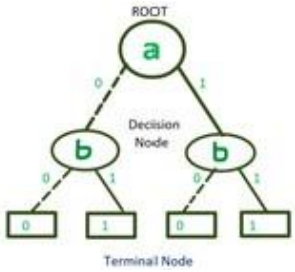
Binary Decision Diagram (BDD) is an effective way to represent the Switching function. It is a Data-Structure used to represent a Boolean Function and can be represented as a compressed form of sets or relations.

BDD is a data structure used to represent a boolean function. It is a directed acyclic graph (DAG) where nodes represent variables and edges represent the boolean function that connects them. BDDs are widely used in computer science, particularly in formal verification and automated reasoning.

BDDs have several properties that make them useful for representing boolean functions. First, they are canonical, meaning that there is only one BDD representation for a given boolean function. Second, BDDs can efficiently handle large and complex boolean functions by representing them compactly. Finally, BDDs can perform a variety of operations on boolean functions, including checking satisfiability, equivalence, and validity.

In summary, BDDs are a powerful tool for representing and manipulating boolean functions, and have many practical applications in computer science.





A binary decision diagram is a rooted, directed, acyclic graph. Nonterminal nodes in such a graph are called decision nodes; each decision node is labeled by a Boolean variable and has two child nodes, referred to as low child and high child.
Shannon decomposition is a technique used in binary decision diagrams (BDDs) to reduce the size of a BDD representing a boolean function. The technique is based on the Shannon expansion theorem, which states that any boolean function can be expressed as the sum of two functions, one where a particular variable is true and one where it is false.

In BDDs, the Shannon decomposition involves splitting a node into two nodes representing the two functions resulting from the Shannon expansion. Specifically, given a node representing a boolean function f(x), the Shannon decomposition involves creating two child nodes: one node representing f(x) when x is true and another node representing f(x) when x is false

The Shannon decomposition can be applied recursively to reduce the size of a BDD. For example, if a BDD has a node representing a function f(x,y,z), the Shannon decomposition can be applied to split the node into two nodes. These nodes can then be further decomposed until a minimal BDD is reached.

Overall, the Shannon decomposition is a useful technique for reducing the size of BDDs, which can help improve the efficiency of algorithms that use BDDs for boolean function manipulation.

Functions

BinaryDecisionDiagram BDD_create

This function creates a Binary Decision Diagram (BDD) based on a given boolean function and an order of variables. The BDD is a data structure used in computer science and logic to represent and manipulate boolean functions. The BDD is constructed recursively by dividing the boolean function into smaller sub-functions until the individual variables are reached.

Reduction occurs during creation.

The function takes two parameters, boolean_function which is the boolean function to be represented and order which is the order of variables used in the function. The function starts by creating an empty BDD and splitting the order string into individual variables. It then splits the boolean function into its constituent parts and looks for any duplicates. If duplicates are found, it increments the count of the existing node, otherwise it creates a new node and adds it to a hash table.

The unique parts of the boolean function are then combined to form a new function string. A root node is created using the first variable in the order as the variable for the node and the newly formed function string as the label. The BDD's root is set to this node and the number of variables in the BDD is set to the length of the order. The BDD's nodes hash table is initialized, and the newly created node is added to it.

The function then recursively creates the sub-trees of the BDD by calling create_subtree function with the root node, the order array, and an index of 0. create_subtree function is responsible for creating the BDD's sub-trees and is called recursively until all variables are processed. Finally, the height of the BDD is calculated by calling the count function with the root node and the BDD is returned.

Overall, the function takes a boolean function and an order of variables, creates a BDD based on the given inputs using a recursive algorithm, and returns the completed BDD.

```
public BinaryDecisionDiagram BDD_create(String boolean_function, String order) {   //creating BDB returning pointer to created BDD
BinaryDecisionDiagram bdd =new BinaryDecisionDiagram();
String[] order_s = order.split( regex: "");  //separate letters
String[] functionss = boolean_function.split( regex: "\\+"); //parts of function

    HashMap<String, Node> unique_parts = new HashMap<>();  //look for duplicate
for (String func : functionss) {
    func = func.trim();
    func = func.replace( target: ".", replacement: "");
    if (unique_parts.containsKey(func)) {
        Node  c = unique_parts.get(func);
        c.count += 1;
    } else {
        Node newn = new Node( right_true: null, left_false: null, variable: null,func);
        newn.count=1;
        unique_parts.put(func, newn);
    }
}
String[] formulas = unique_parts.keySet().toArray(new String[0]);
String whole_f=String.join( delimiter: "+",formulas);

Node root =new Node( right_true: null, left_false: null,order_s[0],whole_f);
bdd.Root = root;
bdd.num_of_variables = order_s.length;
bdd.nodes = new HashMap<>();
nodes.put(whole_f,root);
bdd.order = order;
create_subtree(root,order_s, i: 0,bdd);
bdd.height= count(bdd.Root);
return bdd;
```

The function create_subtree is a helper function for the BDD_create function. It takes a Node object, an array of strings representing the variable order, an integer index, and a BinaryDecisionDiagram object as arguments. The function recursively creates the binary decision diagram (BDD) starting from the given Node object and adding nodes for each level in the variable order.

The function first retrieves the variable at the current level from the order array. It then splits the formula stored in the Node object into its constituent parts using the + operator. The parts of the formula are then analyzed to determine which parts belong in the left and right subtrees.

The left subtree consists of the parts of the formula that evaluate to false when the current variable is set to false. The function creates a HashSet to store these parts and removes the current variable from any parts that contain it. The resulting parts are then joined back together using the + operator to form the formula for the left subtree. If the resulting formula is empty, the false leaf node of the BDD is used. If the resulting formula contains the 1 literal, the true leaf node is used. Otherwise, the function checks if a node with the same formula already exists in the nodes HashMap of the BDD. If not, a new Node object is created with the current variable as its label and the formula for the left subtree as its formula. The new Node object is then added to the nodes HashMap and the create_subtree function is called recursively on this new node.

The right subtree is constructed similarly, except that the parts of the formula that evaluate to true when the current variable is set to true are used. The resulting formula is then used to create a new Node object or retrieve an existing one from the nodes HashMap.

After constructing the left and right subtrees, the function assigns them to the left_false and right_true properties of the current Node object, respectively. The function then calls the reduce_S function to perform a reduction step on the current Node and its descendants.

The reduction step compares the Node object with its children and grandchildren to check for redundant nodes. If the current Node object has the same left and right children, it is replaced with its left child. If the current Node object has a grandchild with the same formula and variable label as itself, it is replaced with that grandchild. The reduction step continues recursively on the children of the reduced node until no further reductions are possible

```
private void create_subtree(Node node,String[] order,int i,BinaryDecisionDiagram bdd)
{
    String bukva= order[i];
    String no_var= '!'+bukva;

    String formula= node.formula;
    String[] parts= formula.split( regex: "\\+");

    HashSet<String> left_tree = new HashSet<>();
    HashSet<String> right_tree = new HashSet<>();

    for(int j=0; j< parts.length;j++) //decomposition part
    {
        String one=parts[j];

        if(one.contains(no_var)&& one.length()>2)
        {
            one=one.replaceAll(no_var, replacement: "");
            if(one.equals(""))
            {
                left_tree.add("1");
            }
            else
            {
                left_tree.add(one);
            }
        } else if (one.contains(no_var)&& one.length()==2) {
            one= one.replaceAll(no_var, replacement: "1");
            left_tree.add(one);
        } else if (one.contains(bukva)&& one.length()>1) {
            one=one.replaceAll(bukva, replacement: "");
```

```
                one=one.replaceAll(bukva, replacement: "");
                if(one.equals(""))
                {
                    right_tree.add("1");
                }else
                {
                    right_tree.add(one);
                }
            } else if (one.contains(bukva) && one.length()==1)
            {
                one=one.replaceAll(bukva, replacement: "1");
                right_tree.add(one);
            }
            else
            {
                left_tree.add(one);
                right_tree.add(one);
                //   System.out.println(bukva+"---");

            }
        }

        String whole_f0= String.join( delimiter: "+",left_tree);
        String whole_f1 =String.join( delimiter: "+",right_tree);
        Node left;
        Node right;
        if(whole_f0.equals(""))    //creating nodes
        {
            left= bdd.false_0;
        } else if (whole_f0.contains("1")) {
            left=bdd.true_1;
        }else
        {
            left=bdd.nodes.get(whole_f0+order[i+1]);

            if(left==null)
            {
                left=new Node( right_true: null, left_false: null,order[i+1], whole_f0);
                bdd.nodes.put(whole_f0+order[i+1],left);
                create_subtree(left,order, i: i+1,bdd);
            }
        }
        if(whole_f1.equals(""))
        {
            right=bdd.false_0;
        } else if (whole_f1.contains("1")) {
            right=bdd.true_1;
        }else
        {
            right=bdd.nodes.get(whole_f1+order[i+1]);
            if(right==null)
            {
                right=new Node( right_true: null, left_false: null,order[i+1], whole_f1);
                bdd.nodes.put(whole_f1+order[i+1],right);
                create_subtree(right,order, i: i+1,bdd);
            }
        }
        node.left_false=left;
        node.right_true=right;
        reduce_S(node);
    }
```

BinaryDecisionDiagram BDD_create_with_best_order(String boolean_function)

This function is used to create a Binary Decision Diagram (BDD) with the best possible order for a given boolean function. It takes in a boolean function as a string and first creates a set of all the variables present in the function. It then creates an array of strings that represents the order of the variables in the BDD.

The function then iterates through all possible variable orders and creates a BDD for each order. The height of each BDD is calculated and stored along with the order used to create it in a HashMap.

I check the order of characters N times, where N is the number of characters.

Finally, the function checks which BDD has the smallest height, indicating that it is the most efficient one, and returns that BDD.

In summary, this function helps to find the best variable order to reduce the size of the BDD, resulting in a more efficient representation of the boolean function.

```
3 usages
public BinaryDecisionDiagram BDD_create_with_best_order(String boolean_function) //find the best order for bdd
{
    HashMap<String,BinaryDecisionDiagram> bdd_comp = new HashMap<>();
    BinaryDecisionDiagram bdd = new BinaryDecisionDiagram();
    Set<Character> variable = new HashSet<>();
    for(int i=0;i<boolean_function.length();i++)
    {
        char c = boolean_function.charAt(i);
        if(Character.isLetter(c))
        {
            variable.add(c);
        }
    }
    String[] order = new String[variable.size()];

    int index =0;
    for(Character c: variable)
    {
        order[index++]=c.toString();
    }
    int num_of_var = order.length;
    int i=0;
    while (i<num_of_var)
    {
        order = new_array(order,i);
        String order_s = String.join( delimiter: "",order);
        bdd = bdd.BDD_create(boolean_function, order_s);
        bdd.order = order_s;
        bdd_comp.put(bdd.order,bdd);
        i++;
```

```
BinaryDecisionDiagram small_bdd=null;
for(BinaryDecisionDiagram temp : bdd_comp.values())
{
    if(small_bdd == null|| temp.height< small_bdd.height)
    {
        small_bdd = temp;

    }
} //System.out.println("With best order: "+small_bdd.height);
return small_bdd;
```

.

char BDD_use(BinaryDecisionDiagram bdd, String values);

This function takes in a binary decision diagram (bdd) and a string of values (values) and returns a single character that represents the value of the boolean function for the given set of input values.

The function first starts at the root of the binary decision diagram (node = bdd.Root) and gets the order of the variables (order = bdd.order). It then checks whether the Root node is null or values string is null, and returns '0' if either one is null.

Next, the function iterates through each character in the values string and checks whether the current node is the false node (bdd.false_0) or the true node (bdd.true_1). If it is, the function returns '0' or '1' respectively.

If the current node is neither the false nor the true node, it checks whether the current node's formula contains the variable represented by the current character in the order string. If it does not, the function increments the index j to move to the next character in the values string and the corresponding variable in the order string.

Once the function has found the correct node for the current variable, it checks whether the value of the variable is '0' or '1'. If it is '0', the function moves to the left child of the current node (node.left_false). If it is '1', the function moves to the right child of the current node (node.right_true).

Finally, the function checks whether the current node is the false node or the true node. If it is, the function returns '0' or '1' respectively. If it has gone through all the variables and still not found the true or false node, the function returns -1.

```
char BDD_use(BinaryDecisionDiagram bdd, String values)//find value for boolean funciton
{   Node node = bdd.Root;
    String order = bdd.order;
    if(bdd.Root == null && values == null)
    {   return 0;
    }for(int j=0; j<values.length();j++) {
        char val = values.charAt(j);
        char bukva = order.charAt(j);
        if (node == bdd.false_0) {
            return '0';
        } else if (node == bdd.true_1) {
            return '1';
        }while ((!node.formula.contains(String.format("%c",bukva))))
        {   j++;
            val=values.charAt(j);
            bukva=order.charAt(j);
        }if(val=='0')
        {
            node=node.left_false;
        } else if(val=='1')
        {
            node=node.right_true;
        }if(node==bdd.false_0)
        {
            return '0';
        }else if(node==bdd.true_1)
        {
            return '1';
        }
    }return (char) -1;
}
```

Testing

As written in a task , testing must be done for the correct execution of the functions and then calculate the time dependence, percentage of reduction and memory in BDD_create and BDD_create_with_best_order.

While testing you can create random number of functions, I am doing a test on 100 random Boolean functions with the difference in the number of variables, so for example for 3 variables I created 100 random Boolean functions and then I test them for correctness

So, to test the functionality of my function BDD create and correctness of BDD use, I wrote a helper function that calculates the value of the Boolean function for all possible combinations of zeros and ones and compare it with the value of BDD use function ,if they are equal, then the function is correct.

```
1 usage
public static void testing_correctness(int number) {
    BinaryDecisionDiagram bdd = new BinaryDecisionDiagram();
    int correct = 0;
    String[] functions = new String[100];
    List<String> combinations = generateBooleanCombinations(number);
    for (int i = 0; i < functions.length; i++) {
        functions[i] = generateDNF(number);
        for (int j = 0; j < combinations.size(); j++) {
            //для звичайного
            bdd = bdd.BDD_create_with_best_order(functions[i]);
            char result = bdd.BDD_use(bdd, combinations.get(j));
            char result2 = evaluateBooleanFunction(functions[i], bdd.order, combinations.get(j));
            if (result == result2) {
                correct++;
            }
            if (number < 4) {
                System.out.println(functions[i] + "result of BDD_use: " + result + " = " + result2 + " combination: " + combinations.get(j));
            }
        }

    }
    System.out.println("Num of functions * num of combinations: " + correct + " " + combinations.size());
    System.out.println("Correctness is: " + (correct / combinations.size()) + "%");
}
```

```java
public static char evaluateBooleanFunction(String booleanFunction, String variablesOrder, String variablesValues) {
    char[] order = variablesOrder.toCharArray();
    boolean[] values = new boolean[variablesValues.length()];
    for (int i = 0; i < variablesValues.length(); i++) {
        values[i] = variablesValues.charAt(i) == '1';
    }String[] parts = booleanFunction.split( regex: "[+]");
    boolean result = false;
    for (String part : parts) {
        boolean partResult = true;
        for (int i = 0; i < part.length(); i++) {
            char variable = part.charAt(i);
            boolean negated = false;
            if (variable == '!') {
                negated = true;
                variable = part.charAt(++i);
            }int index = -1;
            for (int j = 0; j < order.length; j++) {
                if (order[j] == variable) {
                    index = j;
                    break;
                }
            }if (index >= 0) {
                boolean value = values[index];
                if (negated) {
                    value = !value;
                }
                partResult = partResult && value;
            }
        }result = result || partResult;
    }
    return result ? '1' : '0';
```

The next stage is to calculate average size for BDD, time complexity, and memory.
So, I start with asking for how many variables I will do testing(my test allows testing for a maximum of variables).After generating 100 boolean functions for common number of variables I start to create BDD with random order and with best order, also while creating I measure the time for each created bdd and then make an average value. Also while creating bdd I measure memory usage.After creating each bdd I save sizes of all bdd`s and then I will do average size for simple bdd and bdd with best order.

```java
long start2 =0;
long end2=0;
long time2=0;
long end = 0;
long time = 0;
int correct = 0;
int size_reduction=0;
int size_reduction2=0;
int all = 0;
int all2=0;
for (int j = 0; j < functions.length; j++) {
    size_reduction = 0;
    size_before = 0;
    start = 0;
    end = 0;
    String order = order(functions[j]);
    start = System.nanoTime();
    bdd = bdd.BDD_create(functions[j], order);
    end = System.nanoTime();
    start2=System.nanoTime();
    bdd2 =bdd.BDD_create_with_best_order(functions[j]);
    end2 =System.nanoTime();
    time2+= end2-start2;
    size_reduction2 = bdd2.height;
    size_reduction = bdd.height;
    time += end - start;
    size_before = (Math.pow(2, (bdd.num_of_variables + 1)) - 1);
    all_s += size_before;
    all += size_reduction;
    all2+= size_reduction2;
}
```

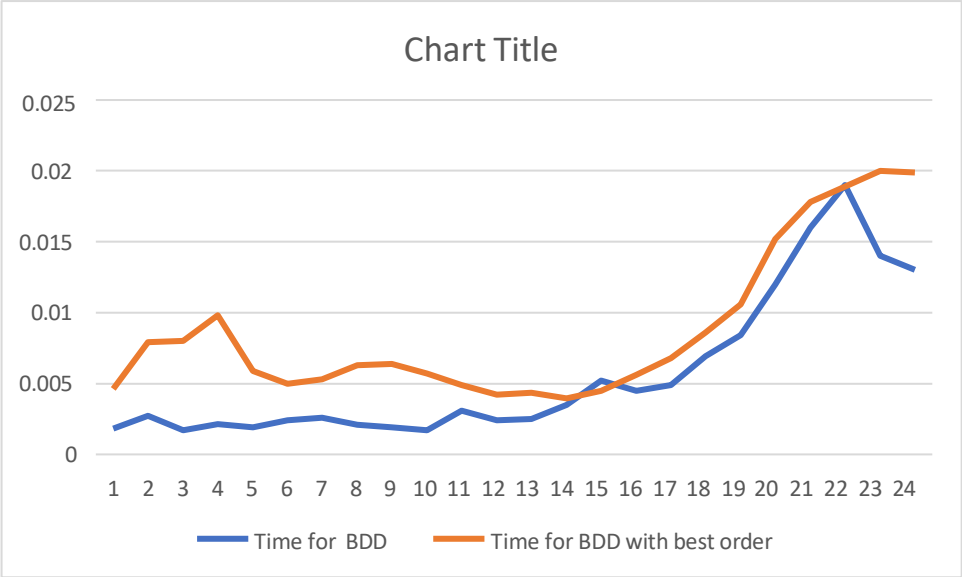
For example, the output to the console will look like this:

```
Testing for correctness press 'c'
 Testing for reduction and  time comparing press 'r':
r
Testing for specific number(press 'x') or from certain number to a maximum(press 'y'):
x
Enter number of variables(max 26):
14
Enter number of functions to generate:
100
Size before reduction for 14 is:32767.0
Average size after reduction: 53
Average percentage rate of BDD reduction: 99%
Average execution time of your BDD functions: 0
Memory usage: 191
For best order:
Size before reduction for14 is:32767.0
Average size after reduction: 50
Average percentage rate of BDD reduction: 99%
Average execution time of your BDD functions: 223
Memory usage: 1047
```

Also, I did some graph of the dependence of time and the number of variables

Time in milliseconds

Number of variables	Time for BDD	Time for BDD with best order
3	0.0018	0.0046
4	0.0027	0.0079
5	0.0017	0.008
6	0.00214	0.0098
7	0.00189	0.0059
8	0.0024	0.00498
9	0.0026	0.0053
10	0.0021	0.0063
11	0.0019	0.0064
12	0.0017	0.0057
13	0.0031	0.0049
14	0.0024	0.0042
15	0.0025	0.00436
16	0.0035	0.00395
17	0.0052	0.0045
18	0.0045	0.0056
19	0.0049	0.0068
20	0.0069	0.0086
21	0.0084	0.0106
22	0.012	0.0152
23	0.016	0.0178
24	0.019	0.0189
25	0.014	0.02
26	0.013	0.0199



Conclusion

When constructing Binary Decision Diagrams (BDDs), it's important to consider several factors such as computational resources, time, and memory usage. Based on the testing conducted, it has been found that finding the best variable ordering for BDD construction takes more time compared to constructing BDDs with a random variable ordering. However, BDDs constructed using the best order have a smaller size compared to BDDs constructed with a random order.

In addition to time and computational resources, memory usage is also an important consideration when constructing BDDs. It's recommended to monitor the memory usage during testing to ensure that the available memory is not exceeded.

Furthermore, the testing has shown that the reduction method employed is effective. By calculating the percentage reduction of the resulting BDDs compared to the BDDs constructed with a random order, it has been demonstrated that the reduction method is useful for optimizing BDD construction and reducing the size of the resulting BDDs.

Overall, the testing provides valuable insights into the trade-offs involved in constructing BDDs. By considering the factors mentioned above, it's possible to construct BDDs that are both efficient and effective for various applications.