Faculty of Informatics and Information Technologies STU, Bratislava Ilkovičova 2, 842 16 Bratislava 4

Assignment 2 – Binary decision diagram

Data Structures and algorithms

Frederik Duvač
2nd semester
Date: 29.4. 2023

Table of contents

1. Theoretical base

1.1. Technical background

Hardware on which the source was compiled and even tested is Lenovo Legion S7, 12th Gen Intel(R) Core™ i5-12500H, 16GB RAM. The whole project is implemented in Java 19 programming language.

1.2. Introduction

A binary decision diagram (BDD) is a data structure used in computer science and engineering to represent boolean functions. In its full version it represents a binary tree. It is a directed acyclic graph (DAG). Each node represents one particular decision. Node handles a boolean variable from order of creation and boolean function, and each edge represents a truth assignment for that variable (0 or 1).

The whole decision goes from the base root node which represents the entire domain, through inside nodes representing one particular decision to the leaf nodes which represent the outputs of the boolean function.

Binary decision diagrams could be used in many applications such as hardware design, software verification, and artificial intelligence. They are also useful in applications involving large boolean functions, as they provide an efficient way to represent and manipulate these functions. They are alternatives for the truth table, vector or Karnaugh map.

1.3. Algorithm explanation

The algorithm which was used to create this binary decision diagram is called Shannon expansion or decomposition. This algorithm works with the order of boolean variables used to create BDD. For example if we have a boolean function AB+A!BC one of the possible order of variables is ABC.

The main idea behind Shannon decomposition is to decompose a boolean function represented by the node into two smaller functions by isolating a single variable and considering its two possible values (0 and 1). This creates two smaller boolean functions, the first one for the left child node in which the variable is to be false, and the second one for the right child node in which the variable is to be true. This process is repeated for the next variable from the order for each of the children in the next level until the leaf nodes are reached. The leaf nodes have only true or false potential values.

One of the advantages of Shannon decomposition is that it provides a way to decompose a large boolean function into smaller manageable pieces. By representing each piece as a smaller boolean function, we can optimise the decomposition by applying possible reductions on redundant nodes and the overall function can be represented as a compact BDD. This can significantly simplify the whole BDD structure and reduce the space complexity of storing and time complexity of evaluating the function.

Truth table, vector and Karnaugh map are representations whose size is 2^N, N is the number of variables in the boolean function. The same problem has also BDD where adding one extra variable means adding the whole next level into BDD. This represents the full BDD but it can be reduced by deleting redundant nodes and merging the nodes which have the same functionality.

2. Implementation of Binary Decision Diagram

Binary decision diagram is represented by the **BDD** class which has attributes such as root **Node** of the binary tree, count of the variables, counter for nodes, order and boolean function. There are also two precreated leaf nodes which are used later in creating BDD.

```
public class BDD {
    private Node root;
    private final int varCount;
    private int nodeCount;
    private String order;
    private final String booleanFunction;
    private final Node trueNode = new Node("1", "LEAF");
    private final Node falseNode = new Node("0", "LEAF");

    public BDD(String booleanFunction, String order) {
        this.booleanFunction = booleanFunction.toUpperCase();
        this.order = order.toUpperCase();
        varCount = order.length();
        this.nodeCount = 0;
    }
```

Each **Node** is represented by its own class and it contains attributes such as list of parents, left and right child, boolean function and variables.

```
public class Node {
    private final List<Node> parents = new ArrayList<>();
    private Node left, right;
    private final String booleanFunction;
    private final String variable;

    public Node(String booleanFunction, String variable) {
        this.booleanFunction = booleanFunction;
        this.variable = variable;
    }
```

Method **BDD_create** is the main function which is responsible for creation of reduced BDD. Firstly the root node is made and added to the level list. Then the program loops through all variables contained in order and for each variable it loops through all nodes in the same level. Each node is decomposed using the Shannon algorithm, so it means that for each node on this level, left and right children are created. These child nodes are added to the **levelHashMap**.

This **levelHashMap** is responsible for making the reduction of the duplicate nodes with the same functionality. Before setting the left or right child to the parent node, the

program asks the **levelHashMap** if it contains the same function yet. If it contains then parent's child is the node from the **levelHashMap**, otherwise it will create the new child node, which is added to **nextLevel** list and also put into **levelHashMap**, increments node counter and sets up pointers on each other. The same process is done for left and right children. If the child is zero or one the pointers are set to one of the prepared true or false nodes and those nodes are final so the decomposition won't be executed on leaves.

Now after this reduction the reductions of the redundant nodes can be made. The same level with nodes is looped through and for each node program looks whether the node has the left and the right children the same, if so we need to delete this node by check all the possible parent of this node and for all parents set the correct pointer on new child, and also set correct parent pointer for child and we decrement the node counter.

Lastly the **level** needs to be resigned to **nextLevel** and the new iteration through order variables could be done.

```
private BDD BDD_create(String booleanFunction, String order) {
   List<Node> level = new ArrayList<>();
   root = new Node(booleanFunction,
String.valueOf(order.charAt(0)));
   level.add(root);
   nodeCount++;

   for (int i = 0; i < order.length(); i++) {
       String s = String.valueOf(order.charAt(i));
       List<Node> nextLevel = new ArrayList<>();
       Map<String, Node> levelHashMap = new HashMap<>();

       for (Node actualNode : level) {
           Set<String> functions = new
HashSet<>(List.of(actualNode.getBooleanFunction().split("\\+")));
           Set<String> rightChild = new HashSet<>();
           Set<String> leftChild = new HashSet<>();

           for (String function : functions) {
               if (function.contains("!" + s)) {
                   if (function.equals("!" + s)) {
                       leftChild.add("1");
                   } else {
                       leftChild.add(function.replace("!" + s,
""));
                   }
               } else if (function.contains(s)) {
                   if (function.equals(s)) {
                       rightChild.add("1");
                   } else {
                       rightChild.add(function.replace(s, ""));
                   }
               } else {
```

```
                leftChild.add(function);
                rightChild.add(function);
            }
        }

        String left = "";
        String right = "";
        if (leftChild.contains("1")) {
            left = "1";
        } else {
            for (String string : leftChild) {
                left = left.concat(string + "+");
            }
            if (left.length() != 0) {
                left = left.substring(0, left.length() - 1);
            } else {
                left = "0";
            }
        }
        if (rightChild.contains("1")) {
            right = "1";
        } else {
            for (String string : rightChild) {
                right = right.concat(string + "+");
            }
            if (right.length() != 0) {
                right = right.substring(0, right.length() - 1);
            } else {
                right = "0";
            }
        }

        if (i < order.length() - 1) {
            String variable = String.valueOf(order.charAt(i +
1));

            if (left.equals("0")) {
                actualNode.setLeft(falseNode);
            } else if (left.equals("1")) {
                actualNode.setLeft(trueNode);
            } else {
                Node leftNode;
                if (levelHashMap.containsKey(left)) {
                    leftNode = levelHashMap.get(left);
                    actualNode.setLeft(leftNode);
                    leftNode.addParent(actualNode);
                } else {
                    leftNode = new Node(left, variable);
```

```
                    nodeCount++;
                    actualNode.setLeft(leftNode);
                    leftNode.addParent(actualNode);
                    nextLevel.add(leftNode);
                    levelHashMap.put(left, leftNode);
                }
            }
            if (right.equals("0")) {
                actualNode.setRight(falseNode);
            } else if (right.equals("1")) {
                actualNode.setRight(trueNode);
            } else {
                Node rightNode;
                if (levelHashMap.containsKey(right)) {
                    rightNode = levelHashMap.get(right);
                    actualNode.setRight(rightNode);
                    rightNode.addParent(actualNode);
                } else {
                    rightNode = new Node(right, variable);
                    nodeCount++;
                    actualNode.setRight(rightNode);
                    rightNode.addParent(actualNode);
                    nextLevel.add(rightNode);
                    levelHashMap.put(right, rightNode);
                }
            }
        } else {
            if (left.equals("0"))
actualNode.setLeft(falseNode);
            else actualNode.setLeft(trueNode);

            if (right.equals("0"))
actualNode.setRight(falseNode);
            else actualNode.setRight(trueNode);
        }
    }

    if (i > 0) {
        for (Node actualNode : level) {
            if (actualNode.getLeft() == actualNode.getRight())
{
                for (Node parent : actualNode.getParents()) {
                    if (actualNode == parent.getLeft()) {
                        parent.setLeft(actualNode.getLeft());
                        actualNode.getLeft().addParent(parent);
                    } else {
                        parent.setRight(actualNode.getRight());
```

```
actualNode.getRight().addParent(parent);
                        }
                    }
                    nodeCount--;
                }
            }
        }
        level = nextLevel;
    }

    if (root.getLeft() == root.getRight()) {
        root.getLeft().addParent(null);
        root = root.getLeft();
        nodeCount--;
    }

    if (root.getVariable().equals("LEAF")) {
        nodeCount++;
    } else {
        nodeCount += 2;
    }

    return this;
}
```

Function **BDD_create_with_best_order** is for creation of possible smaller BDD using the reordering variables in order. This function simply creates N BDDs where N is the number of variables, each BDD has a different order which is changed by the carousel function. The function returns BDD with the smallest number of nodes.

```
private BDD BDD_create_with_best_order(String booleanFunction) {
    String newOrder = this.order;
    BDD minimalBdd = BDD_create(booleanFunction, newOrder);

    do {
        newOrder = carousel(newOrder);

        BDD bdd = new BDD(booleanFunction, newOrder);
        bdd.BDD_create(booleanFunction, newOrder);
        if (bdd.getNodeCount() < minimalBdd.getNodeCount()) {
            minimalBdd = bdd;
            minimalBdd.order = newOrder;
        }
    } while (!newOrder.equals(order));
```

```
    return minimalBdd;
}

private String carousel(String newOrder) {
    String firstChar = String.valueOf(newOrder.charAt(0));
    newOrder = newOrder.substring(1);

    return newOrder.concat(firstChar);
}
```

Method **BDD_use** returns appropriate result value (0 or 1) from created BDD for input vector. If the vector is incorrect the function returns -1.

Firstly it maps each vector value with its suitable variable from the order. It is stored in a hashmap because some of the nodes could be redundant and they are not in BDD so the vector value must be skipped.

In second part the BDD is traversed from the root to the leaf value using vector values, for example if the vector is 010 for the ABC order the tree is traversed to the left, then to the right and lastly to the left child which is the leaf with appropriate return value.

```
private String BDD_use(BDD bdd, String input) {
    if (input.length() == varCount) {
        Map<String, String> vectorMap = new HashMap<>();
        for (int i = 0; i < input.length(); i++) {
            if (input.charAt(i) == '0' || input.charAt(i) == '1')
                    vectorMap.put(String.valueOf(order.charAt(i)),
String.valueOf(input.charAt(i)));
            else return "-1";
        }

        Node result = bdd.getRoot();

           while (result.getLeft() != null && result.getRight() !=
null) {
            String value = vectorMap.get(result.getVariable());

            if (value.equals("0")) {
                result = result.getLeft();
            } else if (value.equals("1")) {
                result = result.getRight();
            } else {
                return "-1";
            }
        }

        return result.getBooleanFunction();
    }
```

```
    return "-1";
}
```

3. Testing

3.1. Console interface

After running the main method program is going to print following command `”Automatic testing -> a | Manual testing -> m | EXIT -> e >> ”`. We can choose between automatic testing and manual testing. If we decide to for automatic testing the next thing he need to set is maximum variables in order `"Maximum variables (min 2, max 26) >> "` and then the number of boolean functions on which the BDDs are going to be created `"Number of boolean functions for each order >> "`. Then we can turn on automatic checking for results of each created BDD `”Do you want to check all BDDs (y / n) >> "`. After creating BDDs there is going to be an additional option to list all average data for each number of variables `”Do you wanna list all average data (y / n) >> ”`.

While BDDs are created the data are printed into the console. For each BDD there is shown the boolean function, number of nodes for non reduced BDD, variables count, order, then the creation with reduction is done, and the number of nodes, reduction by certain percentage and creation time is displayed. Then the program finds the BDD with best order and also number of nodes, additional percentage reduction by, total percentage reduction by and creation time is displayed.

The manual testing provides `"Create with best order -> bo | Create with reduction -> r | Create without reduction -> wr | EXIT -> e >>"` for the provided boolean function and order. And then the program gives the option for automatic or manual use of created BDD `"Automatic check -> a | Manual check -> m | EXIT -> e >> "` for input vector value `"Input vector value for use created BDD (EXIT = e) >> "`.

3.2. Testing implementation

Testing of data structures was implemented in the Main class using the main method. Automatic testing which was mentioned previously, uses **GenerateFunction** class, which generates using the generateFunctions method the same amount of unique boolean functions for each number of variables.

```
public class GenerateFunctions {
    private final int maxVar;
    private final Map<Integer, List<String>> functionMap = new
HashMap<>();

    public GenerateFunctions(int maxVar, int
numberOfBooleanFunction) {
```

```
        this.maxVar = maxVar;
        generateFunctions(numberOfBooleanFunction);
    }

    private void generateFunctions(int numberOfBooleanFunction) {
        Random random = new Random();

        for (int i = 2; i < maxVar + 1; i++) {
            Set<String> functionSet = new HashSet<>();

            while (functionSet.size() != numberOfBooleanFunction) {
                String booleanFunction = "";
                int functionsCount = random.nextInt(2, 16);

                for (int j = 0; j < functionsCount; j++) {
                    String function = "";
                    int functionLength = random.nextInt(1, i + 1);

                    for (int k = 65; k < 65 + functionLength; k++)
{
                        int percent = random.nextInt(0, 100);
                        if (percent < 80) {
                            if (random.nextBoolean()) {
                                function = function.concat("!" +
Character.toString(k));
                            } else {
                                function =
function.concat(Character.toString(k));
                            }
                        }
                    }

                    if (function.length() == 0) {
                        if (random.nextBoolean()) {
                            function = "!" +
Character.toString(random.nextInt(65, 65 + functionLength + 1));
                        } else {
                            function =
Character.toString(random.nextInt(65, 65 + functionLength + 1));
                        }
                    }

                    booleanFunction =
booleanFunction.concat(function + "+");
                }

                booleanFunction =
booleanFunction.substring(0, booleanFunction.length() - 1);
```

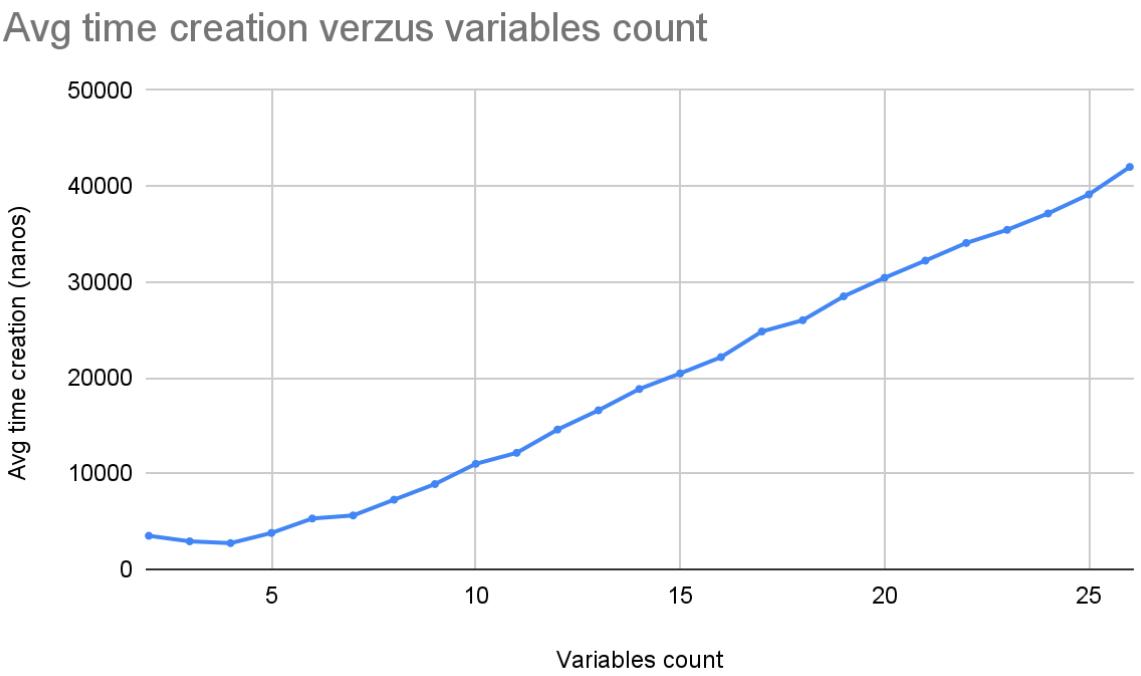
```
                functionSet.add(booleanFunction);
            }
            functionMap.put(i, functionSet.stream().toList());
        }
    }
```

In the main function in automatic testing the program loops through all generated boolean functions for each number of variables and for each boolean function data are shown. Each BDD creation is measured by time complexity and also percentage reduction by. From this the average values are calculated for all functions and they are stored into avg_data.csv file.

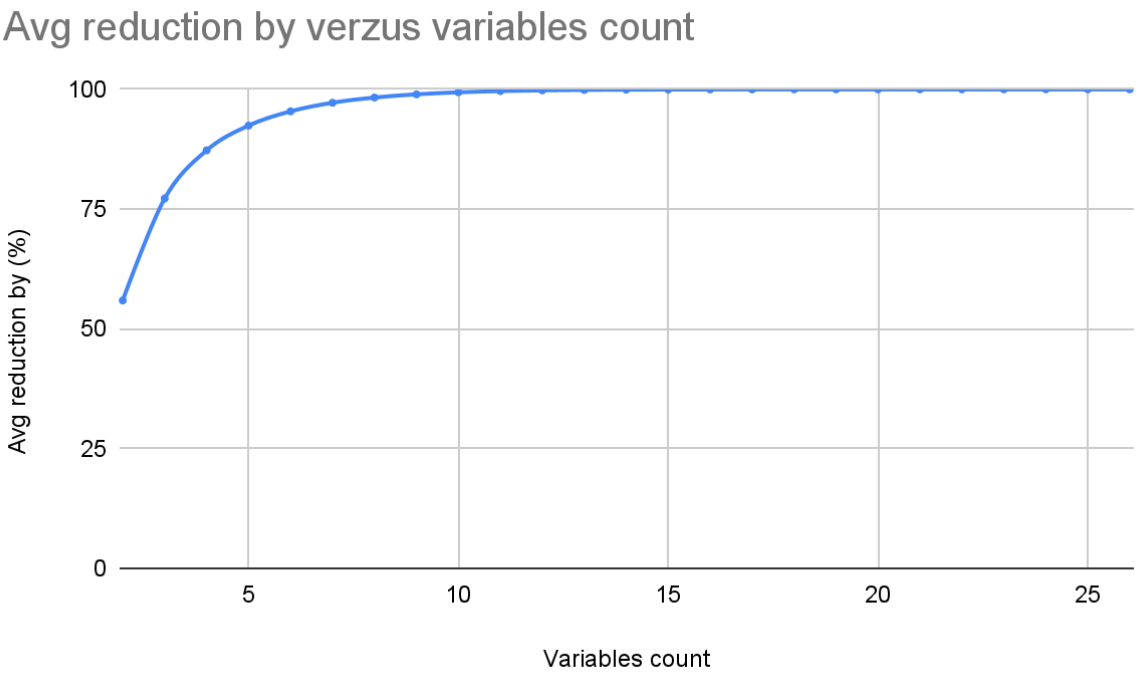
In the manual testing section, the similar data are shown but only for one provided boolean function. There are three options: creation with best order, creation with reduction or create full BDD. After BDD creation, the whole BDD is written into file bdd_print.txt file using **BinaryTreePrinter** class (3).

3.3. Time complexity

From the graph shown below we can see that reduction implementation using a hash table is very efficient because we were able to reduce time complexity from O(N^2) to O(N).



3.4. Space complexity



From this percentage reduction graph we can see that if the number of variables is increased the number of nodes in the binary decision diagram is approaching quite a steep reduction by almost to 100%. Theoretically space usage of one BDD is: nodes count * sizeof(node) + sizeof(BDD attributes).

4. Conclusion

In conclusion, we can say that our implementation of the Shannon decomposition and implementation of the reduction using hashtable and reduction of redundant nodes is correct because the result shows from graphs that percentage reduction has a logarithmic potential O(log n) and the time complexity of the BDD creation has linear complexity O(n).

5. References

1. dsa_07.pdf
2. dsa_08.pdf
3. https://github.com/eugenp/tutorials/blob/master/data-structures/src/main/java/com/baeldung/printbinarytree/BinaryTreePrinter.java