Slovak University of Technology in Bratislava

Faculty of Informatics and Information Technologies

BINARY DECISION DIAGRAMS

Data Structures and Algorithms - Assignment 2

Abstract

Implementation of Binary Decision Diagram (BDD) to evaluate DNF Boolean function

David Truhlar

xtruhlar@stuba.sk

ID: 120897

Contents

1. Assignment

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

- BDD *BDD_create(string *bfunction,* string *order*);
- BDD *BDD_create_with_best_order (string *bfunkcia*);
- char BDD_use(BDD **bdd*, string *input_values*);

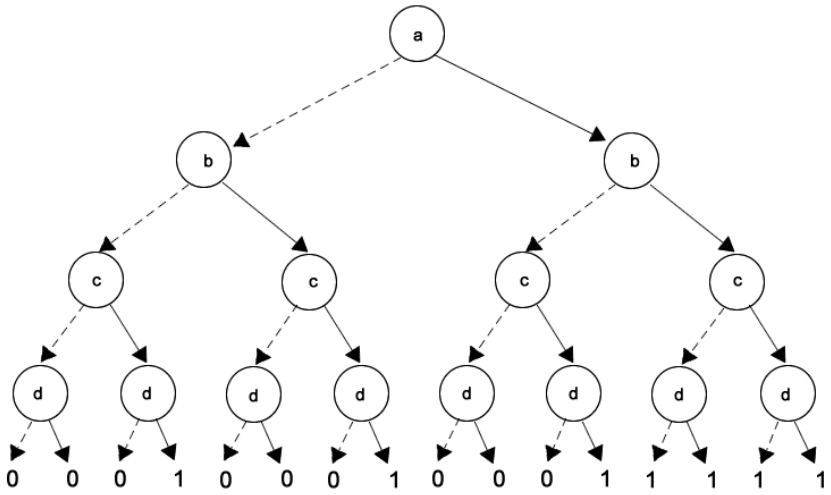
Function **BDD_create** serves to create a reduced binary decision diagram that is supposed to represent/describe any given Boolean function. The Boolean function is provided as function argument called *bfunction* (data type and format is up to you). The Boolean function is described in a form of an expression in a string. The second argument of **BDD_create** is the *order* of variables used for the Boolean function (data type and format is up to you). This order specifies the order of usage of these variables for creation of BDD (i.e. the BDD is created in the order of variables according to the *order* argument). Function **BDD_create** returns a pointer to the created BDD structure. This structure has to contain these parts at least: number of variables, size of BDD (i.e. number of BDD nodes) and a pointer to the root of BDD tree. Of course, you will need your own structure for representation of one node too. This **BDD_create** function already includes the reduction of BDD – you can choose whether you reduce the BDD during its creation or you reduce BDD after it is fully created. If you reduce BDD after it is fully created, then your solution is evaluated with fewer points (4 points penalization).
Function **BDD_create_with_best_order** serves to find the best order (within the explored orders) of variables for the given Boolean function. Searching is based on calling **BDD_create** function, with exploring (trying) various orders of variables (argument *order*). For example, for Boolean function with 5 variables, we can call **BDD_create** 5-times, with orders of variables 01234, 12340, 23401, 34012 a 40123. Alternatively, we can try even all possible permutations of orders, which are N! in total, where N is the number of variables (i.e. in this case 5! = 120). Or, we can use X randomly selected orders of variables. The important thing is that you try at least N unique orders of variables, where N is the number of variables of Boolean function. This function returns a pointer to the smallest found BDD, i.e. which has the lowest number of nodes out of all tried orders of variables.
Function **BDD_use** serves for using created BDD for a specific given combination of values for input variables of Boolean function and for obtaining the result of Boolean function for the given input. Within this function, you „walk through“ BDD tree from its root down to a leaf. The path from root to leaf is determined by the given combination of values of input variables. The arguments of this function are a pointer to a specific BDD that is used and a string called *input_values*. This string / array of chars is representing the given combination of values for input variables of Boolean function. For example, index of the array/string represents one variable and value at this index represents the value of this variable (i.e. for variables A, B, C and D, where A and C are 1s and B and D are 0s, the string can be “1010“), but this is just an example – you can choose to represent the *input_values* in a different way. The return value of function **BDD_use** is a char, which represents the result of Boolean function – it is either ‘1’ or ‘0’. In case of an error (e.g. incorrect input), this result should be negative (e.g. -1).
Apart from implementation of the BDD functionality, it is required to test your solution appropriately. Your solution must be 100% correct. Within the testing it is needed to use some randomly generated Boolean functions, which will be used for creation of BDDs using the **BDD_create** and **BDD_create_with_best_order**. The correctness of BDD can be proved with iterative calling of function **BDD_use** in such a way that you will use all possible combinations of values for the input variables of Boolean function and compare the output/result of **BDD_use** with expected result. The expected results can be obtained from application of the values to variables within the Boolean function expression (evaluating the expression). Test and evaluate your solution for various numbers of variables of Boolean function – the more variables your program can handle, the better (it should be able to handle at least 13 variables). The number of different Boolean functions within the same amount of variables should be at least 100. Within your testing, you should evaluate the relative rate of BDD reduction (i.e the number of deleted nodes / number of all nodes for a full diagram) – comparing reduced BDD with full BDD and comparing the size of BDD using the best found order with basic without looking for the best order of variables.

Create a documentation, where you solution, individual functions, your structures, way of testing and test results are described. The test results should include average percentage rate of BDD reduction and average execution time of your BDD functions, of course, based on the size of Boolean function (i.e. number of variables). Documentation must contain a header (who, which assignment), brief description of the used algorithms with some pictures showing how the solution works or selected parts of your code. Try to explain why you think that your solution is correct – reasons why it is good/correct, and the way how you tested/evaluated it. At the end, your documentation should contain your estimation of time and memory complexity of your solution (for both, **BDD_create** and **BDD_create_with_best_order**). Overall, it must be clear what you did and how can you prove that it is correct and how efficient it is.
The solution of the assignment must be uploaded in AIS by the set deadline (delays are allowed only in very serious cases, such as sickness, the decision whether the solution is accepted after deadline is decided by the instructor). One zip archive should be uploaded, which contains individual source code files with implementations, a test file and one documentation file in **pdf** format.

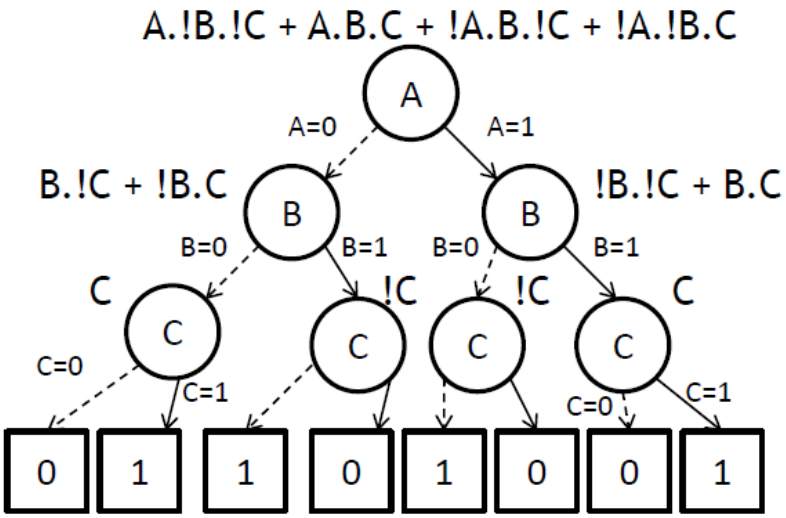
2. Binary Decision Diagrams

The Binary Decision Diagram (BDD) is a prevalent data structure in the field of computer science, utilized for the representation of Boolean functions. Unlike a conventional binary tree, BDD's primary aim is to facilitate decision-making by identifying the optimal branch to traverse. The complete decision-making process is executed from the root to the terminal leaves that are labeled with either "true" or "false" ("0" or "1"). Each node of the BDD represents a singular, partial decision.

bfunction = !A.B.C + A.!B.C + A.B.!C + A.B.C



In order to generate BDDs, our team relies on the application of Shannon's decomposition. This method involves utilizing the distributive law [A.B.C = A.(B.C)] for the control variable [A] at each level of the tree. By consistently implementing this process, we can effectively create BDDs that accurately represent the corresponding Boolean functions.



3. My implementation of BDD

3.1 Initialization

In order to commence the BDD construction process, our team initiated the creation of a new class. This class serves as a representation of a single node within the BDD structure. By implementing this class, we were able to efficiently create and manipulate nodes within the BDD.

```
public class BDDNode {
    /* Štruktúra BDDNode */
    private final String variable;
    private BDDNode low;
    private BDDNode high;
    private BDDNode parent;
    private String controlVariable;
}
```

These two nodes represent the final leaves – terminals **true** and **false**.

```
/* Koncové uzly pre "0" a "1" */
static BDDNode trueNode = new BDDNode("1", null, null);
static BDDNode falseNode = new BDDNode("0", null, null);
```

Then I created another class that represents BDD tree. It consists of root node, variable order and number of nodes.

```
public class BDD {
    /* Štruktúra BDD */
    static HashMap<String, BDDNode> hashTable = new HashMap<>();
    public static BDDNode root;
    public String[] variablesOrder;
    public static int nOfNodes;
}
```

I have selected the Disjunctive Normal Form (DNF) format for the bfunction. It is represented without the use of braces and the logical operators are depicted as follows: '!' for NOT, '+' for OR, and '' for AND.

Example:

```
String formula = "A!B!C+ABC+!AB!C+!A!BC";
```

3.2 Shannon's decomposition

After I created necessary structures, I created methods to decompose bfunction. Parameters of this methods are CurrentNode, control variable and Boolean value to represent if it is high or low branch.

```
public BDDNode decompose(BDDNode currentNode, String variable, boolean logHodnota)
```

First, I create array by splitting the formula into clauses – '+' is separator. I am using StringBuilder in java to create formula on lower level of the BDD tree. Then there is if condition to add new node to the right side of BDD. When the argument logHodnota is false, I know that I must negate decisions – meaning that usually when you have variable, you decide as follows:

variable[true] -> High variable[false] -> Low,

but with logHodnota set to false I have to change it to:

variable[true] -> Low variable[false] -> High

After I check what logHodnota is there are more conditions I must take in account for ich clause from clauses array.

```
if (!logHodnota) {
    for (String clause : clauses) {
       if (clause.equals(negatedVariable) && clauses.length > 1) {
            return trueNode;
        }
        if (clause.contains(negatedVariable)) {
            clause = clause.replace(negatedVariable, "");
            if (!nextLevelVariable.toString().equals("")) {
                nextLevelVariable.append("+").append(clause);
            } else {
                nextLevelVariable.append(clause);
            }
        } else if (!clause.contains(variable)) {
            if (!nextLevelVariable.toString().equals("")) {
                nextLevelVariable.append("+").append(clause);
            } else {
                nextLevelVariable.append(clause);
            }
        }
    }
```

In first case we just need to check, ich clause is not the variable alone. In this case we are on logHodnota = false so we check if it is negatedVariable.

Second conditions check if the clause contains the negatedVariable and if yes it just removes the variable from clause and add it to nextLevelVariable which represent bformula on lower level.

In the last case there, clause does not contain control variable, so the whole clause is added to nextLevelVariable.

The same logic is applied on logHodnota = true branch but instead of negatedVariable I check variable.

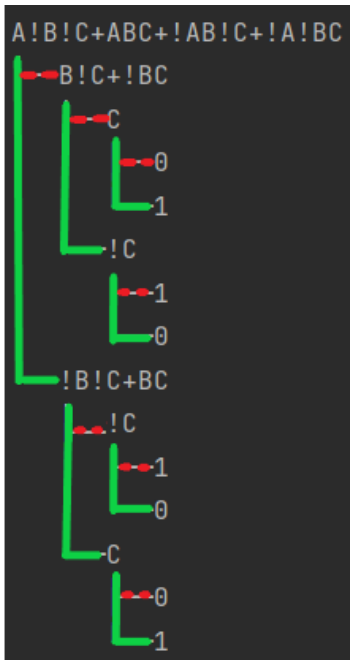
Lastly, I need to take care of the end of the tree – terminals. So, I check if the nextLevelVariable is “” – empty string and if yes, based on if we use variable or negatedVariable I decide to return “0” or “1” terminal.

```
if (nextLevelVariable.toString().equals("")) {
    if (currentNode.getVariable().contains(negatedVariable)) {
        if (logHodnota) {
            return falseNode;
        } else {
            return trueNode;
        }
    } else {
        if (logHodnota) {
            return trueNode;
        } else {
            return falseNode;
        }
    }
}
```

After this implementation of Shannon's decomposition, the BDD for bformula A!B!C+ABC+!AB!C+!A!BC looks like this:

Green is High

Red is Low

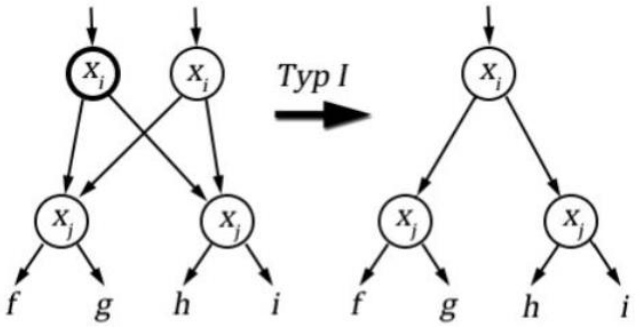


3.3 Reducing BDD

After decomposition I implemented numerous rules to make tree smaller – meaning it has fewer nodes.

Reduction – Type I

- this reduction compares two nodes – their *bformula* and if they are the same, they are merged into one node.



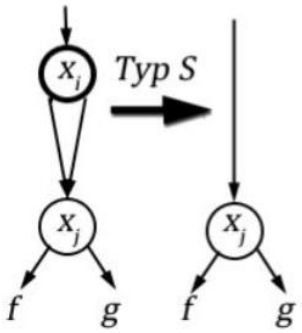
My implementation looks as followed code.

```
/* Redukcia typu I */
for (int i = 0; i < array.size(); i++) {
    for (int j = i + 1; j < array.size(); j++) {
        BDDNode node1 = array.get(i);
        BDDNode node2 = array.get(j);
        /* Ak nájdeme dva nody s rovnakou formulou */
        if (node1.getVariable().equals(node2.getVariable())) {
            /* Ak majú rovnakého low aj high node, môžeme jeden z nich vymazať */
```

```
            if (node1.getLow() != null && node1.getHigh() != null &&
node1.getLow().getVariable().equals(node2.getLow().getVariable()) &&
node1.getHigh().getVariable().equals(node2.getHigh().getVariable())) {
                if
(node2.getParent().getLow().getVariable().equals(node2.getVariable())) {
                    if (node2.getParent().getParent().equals(node2.getParent())) {
                        node2.getParent().getParent().setLow(node1);
                    } else {
                        node2.getParent().getParent().setHigh(node1);
                    }
                } else if
(node2.getParent().getHigh().getVariable().equals(node2.getVariable())) {
                    if (node2.getParent().getParent().equals(node2.getParent())) {
                        node2.getParent().getParent().setLow(node1);
                    } else {
                        node2.getParent().getParent().setHigh(node1);
                    }
                }
                array.remove(node2);
            }
        }
    }
}
```

Reduction – Type S

This reduction is used when the node's low and high are the same.




```
/* Redukcia typu S */
for (int i = 0; i < array.size(); i++) {
    BDDNode node1 = array.get(i);
    /* Ak sa formuly v low a high rovnajú */
    if (node1.getHigh() != null && node1.getLow() != null &&
node1.getLow().getVariable().equals(node1.getHigh().getVariable())) {
        /* nastavíme pointer parenta na child aby sme mohli node1 vynechať a
vymazať */
        if (node1.getParent() != null && node1.getParent().getLow().equals(node1))
{
            node1.getParent().setLow(node1.getLow());
        }
        if (node1.getParent() != null && node1.getParent().getHigh().equals(node1))
{
            node1.getParent().setHigh(node1.getLow());
        }
        array.remove(node1.getHigh());
        array.remove(node1);
    }
}
```

4. BDD_create

In my implementation I chose this cycle to create the BDD tree with given bfunction together with order of variables. First, I create 'array' which represents the current level of the tree.

Then for each variable I create newArray array list, and for each node that is already created I create low and high branch using Shannon's decomposition. I also set 'controlVariable' by which current node was decomposed. Then I reduce this 'newArray' and set the 'array' with 'newArray'.

```
/* Metóda BDD_create */
public static BDD BDD_create(String formula, String[] variableOrder) {
    BDDNode root = new BDDNode(formula, null, null);
    ArrayList<BDDNode> array = new ArrayList<>();
    array.add(root);
    root.setParent(root);

    for (String variable : variableOrder) {
        ArrayList<BDDNode> newArray = new ArrayList<>();
        for (BDDNode node : array) {
           if (!node.getVariable().equals("0") && !node.getVariable().equals("1")){
                node.setControlVariable(variable);

                node.setLow(BDD.decompose(node, variable, false));
                node.getLow().setParent(node);
                newArray.add(node.getLow());

                node.setHigh(BDD.decompose(node, variable, true));
                node.getHigh().setParent(node);
                newArray.add(node.getHigh());

                }
            }
        }
        BDD.BDDreduce(newArray);
        array = newArray;
    }
    return new BDD(root, variableOrder);
}
```

5. BDD_Create_with_best_order

When creating tree with best order, I try to create *n* variations trees, count how many nodes in each of these tree is, and the order which created smallest tree is returned as bestOrder. Then this method creates the tree with formula and bestOrder of variables.

```
/* Metóda bestOrder */
public static String[] bestOrder(String formula, String[] chooseFrom) {
    String[] bestOrder = {};
    HashMap<Integer, String[]> table = new HashMap<Integer, String[]>();
    int min = (int) Math.pow(2, chooseFrom.length) + 1;

    // First try with the given order
    BDD temp = BDD_create(formula, chooseFrom);
    int tempInt = countUniqueNodes(temp.getRoot());
    if (tempInt < min) {
        min = tempInt;
    }
    table.put(tempInt, chooseFrom.clone());

    // Iterate over all other orders
    for (int i = 1; i < chooseFrom.length; i++) {
        // Randomly shuffle the array
```

```
        Collections.shuffle(Arrays.asList(chooseFrom));
        temp = BDD_create(formula, chooseFrom);
        tempInt = countUniqueNodes(temp.getRoot());
        if (tempInt < min) {
            min = tempInt;
            table.clear();
        }
        if (!table.containsKey(tempInt)) {
            table.put(tempInt, chooseFrom.clone());
        }
    }

    bestOrder = table.get(min);
    return bestOrder;
}
```

6. BDD_use

The BDD_use method first rewrite formula in the root to '0' and '1s' like this:

formula: AB+!CD variable order: ABCD input: 0001

The program takes index of input String (index 0: "0", index 1: "0", index 2: "0", index 2: "1") and index of each variable (A: 0, B:1, C:2, D:3). Then every instance of variable in formula is rewritten.

variable : A, input : "0" -> 0B+!CD, … ,D, input: "1" -> 00+11. This rewritten formula is slit to clauses, and if there is at least one clause which doesn't contain "0" the result should be **true**.

The second result is found in the tree. Based on controlVariable stored in the node the next level variable is selected.

```
public static void BDD_use(BDD tree, String input) {
    String[] variablesOrder = tree.variablesOrder;
    String formulaToRewrite = tree.getRoot().getVariable();
    /* Pre každú premennú ... */
    for (String variable : variablesOrder) {
        String negatedVariable = "!" + variable;
        int index = Arrays.asList(variablesOrder).indexOf(variable);
        if (index >= input.length()) {
            throw new IndexOutOfBoundsException("Index out of bounds: " + index);
        }
        boolean inputBool = input.charAt(index) == '1';
        /* ... prepisujeme na základe 'true' / 'false' premenné na '1' alebo '0' */
        if (inputBool) {
            formulaToRewrite = formulaToRewrite.replaceAll(negatedVariable, "0");
            formulaToRewrite = formulaToRewrite.replaceAll(variable, "1");
        } else {
            formulaToRewrite = formulaToRewrite.replaceAll(negatedVariable, "1");
            formulaToRewrite = formulaToRewrite.replaceAll(variable, "0");
        }
    }
    boolean result = false;
    /* Rozdelíme na klauzuly */
    String[] clauses2 = formulaToRewrite.split("\\+");
    /* Ak je aspoň jedna klauzula, ktorá neobsahuje '0' teda obsahuje samé '1'-tky
formula je pravdivá pri danom vstupe */
    for (String clause : clauses2) {
        if (!clause.contains("0")) {
            result = true;
            break;
        }
    }
    String[] inputs = input.split("");
```

```
    int k = inputs.length;
    boolean[] arr = new boolean[k];
    for (int i = 0; i < inputs.length; i++) {
        if (inputs[i].equals("0")) {
            arr[i] = false;
        }
        if (inputs[i].equals("1")) {
            arr[i] = true;
        }
    }
    /* Result 2 je traverzovanie vo vytvorenom strome, k získaniu 'true' / 'false'
*/
    boolean result2 = evaluate(tree.getRoot(), arr, tree.variablesOrder);
    System.out.println("Očakávaný výsledok: " + result + " <-> Výsledok: " +
result2);
    /* vyhodnotenie result vs result2 */
    if (result == result2) {
        System.out.println("☑ Výsledky sa zhodujú");
        System.out.println();
    } else {
        System.out.println("❌ Výsledky sa nezhodujú");
        System.out.println();
    }
}
```

7. Other functions

7.1 getVariables()

```
public static String[] getVariables(String formula) {...}
```

This method extracts unique variables from formula.

7.2 percentReduction()

```
public static double percentReduction(int numOfVariables, BDD tree) {...}
```

This method calculates how much the tree was reduced in [%].

7.3 countUniqueNodes()

```
public static int countUniqueNodes() {...}
```

This method counts nodes in BDD tree.

7.4 testPrint()

```
public static void testPrint(BDD root, String formula, String[] variableOrder,
double k) {...}
```

This method prints information about tree in console.

7.5 generateRandomFormula()

```
public static String generateRandomFormula(int numOfVariables) {...}
```

This method generates random formula in DNF form.

7.6 generateInputs(), results(), evaluate() and BDD_test()

```
public static List<boolean[]> generateInputs(int n) {...}
public static List<Boolean> results(BDD tree, List<boolean[]> inputs){...}
public static boolean evaluate(BDDNode node, boolean[] inputs, String[]
variablesOrder){...}
public static void BDD_test(BDD tree) {...}
```

These methods are used, to check if the tree is constructed correctly, generates all possible inputs, then expected results, results with created tree and compare them.

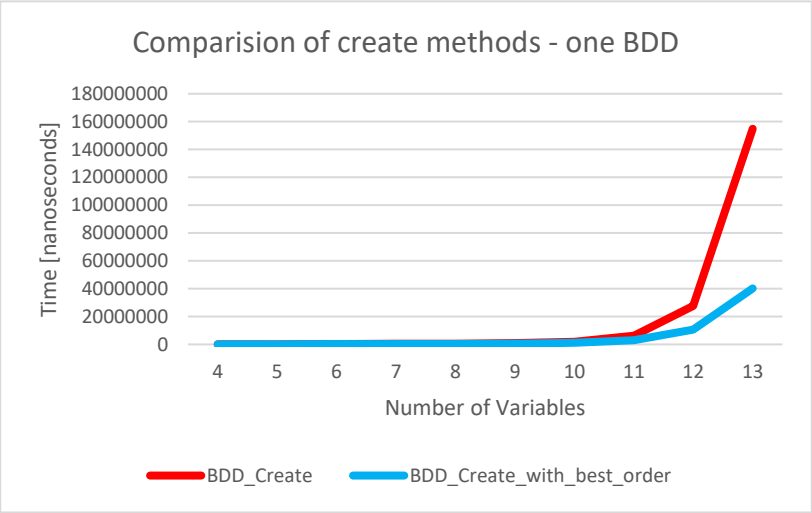
8. Testing

To test if my implementation is effective, I took different tests. First to find if my reduction is effective. How many nodes each of created BDD has. I also measured how much time it takes to create one tree with BDD_Create method and with BDD_Create_with_best_order.

Final graphs represent relative rate of reduction, absolute number of nodes and time of creation.

8.1 Time complexity

To see if my code is effective enough it must manage to create tree with at least 13 variables in reasonable time. This graph represents this complexity, on x axis there are number of variables in the tree and the y axis represents time in nanoseconds.



Graph 1.

Time complexity

BDD_create method vs BDD_crete_with_best_order

Creation of one BDD tree using random order and best-found order.

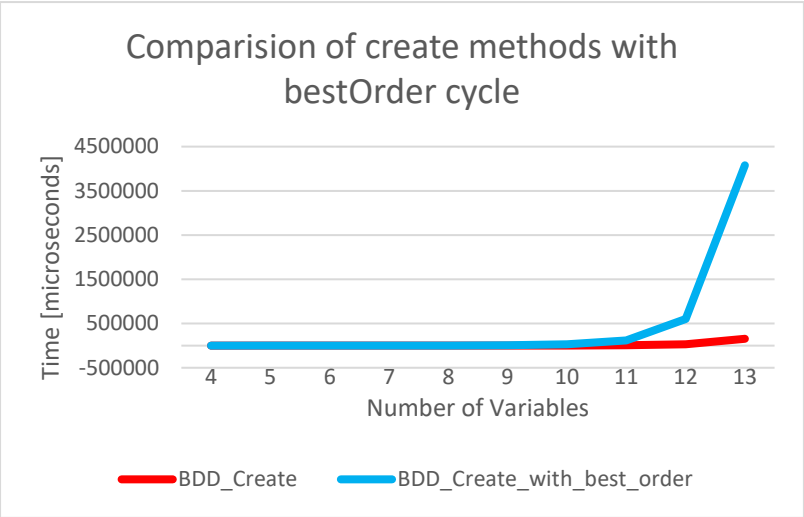
Time complexity is increasing exponentially in both cases.

This measurement shows, that method BDD_create_with_best_order has purpose with bigger trees. For example, when there are 13 variables in the tree, the time of creation is almost 4-times (3.85x) faster compared to BDD_create alone.

8.2 Time complexity – finding best variable order

There are many ways to find best possible order of variables to create BDD tree with given formula. It is important to choose the order of variables wisely, but also we need to keep the creation time together with finding the best order reasonable. In my implementation, I try **n-variation** of 'VariableOrder[]' array and the one with fewest nodes is saved and BDD_create_with_best_order uses this order.

In the 8.1 paragraph we can see that BDD_create_with_best_order is faster than BDD_create alone but the next graph shows the price for it. It takes much longer to find bestOrder than just create tree with random order.



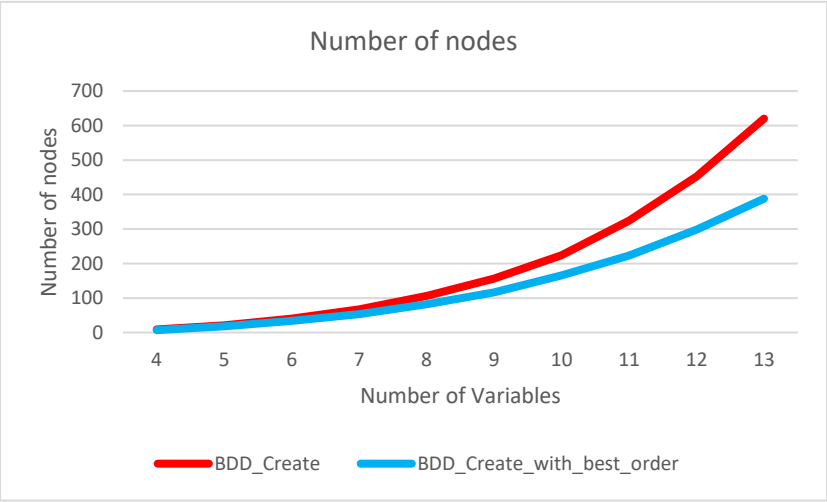
Graph 2.

Finding best order of variables

The finding bestOrder cycle makes the BDD_create_with_best_order less effective with bigger number of variables. The red curve is the same red curve as in graph 1. Also, I have to change time unit to microseconds to see both curves .

8.3 Absolute number of nodes

Another test is related to time complexity, but in a different way. When we are using BDD_use method it is important to find the correct solution of the DNF formula using the BDD tree we just created, it means we need to find 'true' or 'false' with the lowest possible decisions in the nodes.



Graph 3.

Number of Nodes

Number of unique nodes in the tree reflects why the best order is created faster.

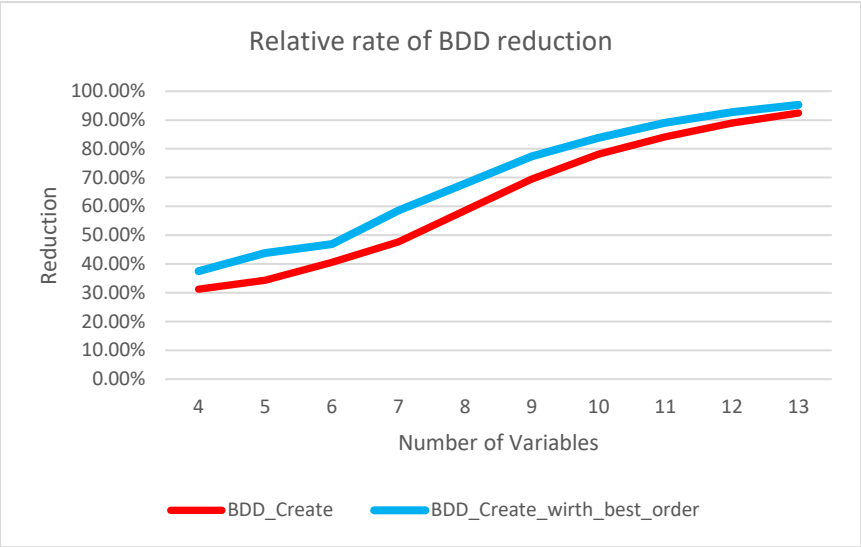
Complexity of BDD_Create is 2^n, but we can see that create with best order is making the rise of the number of nodes slower.

8.4 Relative rate of reduction

Relative rate of reduction is calculated with formula:

$$\frac{Number\ of\ deleted\ nodes}{Number\ of\ nodes\ in\ full\ BDD\ tree} * 100$$

Graph 3. represents how this reduction change when number of variables change. It is important to node that the more variables there is the more tree is reduced because the tree is getting wider – leading to repeating its low and high branches.



Graph 4.

Relative rate of reduction

The reduction differs with smaller number of variables but the more variables there is the more the tree is reduced.

9. Conclusion

To conclude, Binary Decision Diagrams are used primarily to solve *bfuncions.* Our implementation is able to solve any DNF form of bfunction. Testing that can be done on every BDD tree shows that solutions are 100% correct.

```
BDD_test
⏳ Generovanie 'inputov': [########################################] 100%
⏳ Generovanie očakávaných výsledkov: [########################################] 100%
⏳ Generovanie výsledkov pomocou stromu: [########################################] 100%
⏳ Porovnávanie výsledkov: [########################################] 100%

✅ Výsledky sa zhodujú
```

Also in the program there is option BDD_use to check given input one by one.

```
BDD_use [počet premenných = 4]
1101
Očakávaný výsledok: false <-> Výsledok: false
✅ Výsledky sa zhodujú
```

When we were sure that tree is always correct, we applied reductions. In this implementation not all possible reductions are used. However, the reduction rate is high anyway. All points from assignment fulfilled.