Slovak Technical University in Bratislava, Faculty of Informatics and Information Technologies

Data Structures and Algorithms
Assignment 2 – Binary Decision Diagrams

STU FIIT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Filip Zubaj 120914 Academic year 2022/23

Table of contents

Table of Contents

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*);

  Of course, you can implement any other functions, which can be used inside your program, helping with implementation of the abovementioned functions. You cannot, however, use existing code/functions for BDD implementation.

  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.

My understanding of assignment

The goal of this project was to work on a program which will create binary decision diagram. Our binary decision diagram should be used to solve Boolean functions. In my particular project, the input will be Boolean function in the disjunctive normal form, and it should look like this *!a*b*c+c*d+e.* There must be multiplying sign between the literals and the plus sign between the units themselves. In our implementation, 3 methods must be implemented, *BDD_create*, *BDD_create_with_best_order* and *BDD_use*. I will explain these methods further in my documentation.

IDE and programming language

For my assignment I am using the IDE IntelliJ from JetBrains and the language I have chosen for this assignment is Java. My assignment has been done in my MacBook Air 2020, M1 with 8 GB RAM and 256 GB SSD.

What is a binary decision diagram?

Binary decision diagram is a data structure, often used in the computer science field to represent Boolean functions. BDD is a acyclic graph and it is directed. My BDD works with Shannon's decomposition. Every node in the BDD represents a variable and there are always two outputs from a node. Left which represents zero and right which represents one. If it is zero, we substitute the variable with 0 and if it is 1 we substitute it for 1. This way we can see the output of the Boolean functions (O or 1) based on whether the variables are one or zero. This is often used for representing the large and complex Boolean functions in. a compact way.

My binary decision diagram supports the DNF form of a Boolean function. Which means it can solve conjunctions and disjunctions, and also negations. Thanks to the BDD we can solve all of the possible outcomes of the function which is an input. BDDs have applications in different areas, such as model checking, formal verification, logic synthesis and satisfiability testing. BDDs provide us with the efficient representation of the Boolean functions, which is a great tool for analysing logical expressions.

My code

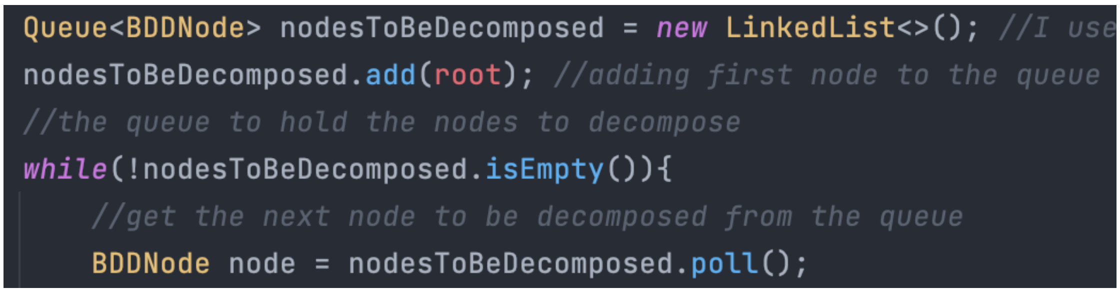
In my code I have one class of *BDDNode*, which stores variable for decomposition, current formula, left and right child, parent of the node and it counts the created nodes. I also have a class of BDD which contains all the methods such as *BDD_create, BDD_create_with_best_order* and *BDD_use*.

BDD

Binary decision diagram itself is a class, which contains the 3 main methods + helper methods, used for better structure of my code. BDD stores the root node, amount of variables and order of the decomposition. Root as a BDDNode, stores the amount of nodes after the reductions. The amount of nodes without reductions is calculated with the formula 2^{n+1}-1 and I also calculate the rate of reduction, by using these two integers. BDD consists of BDDNodes, and with BDD_use, we can traverse throughout our tree, to find the resulting value, which is either true or false.

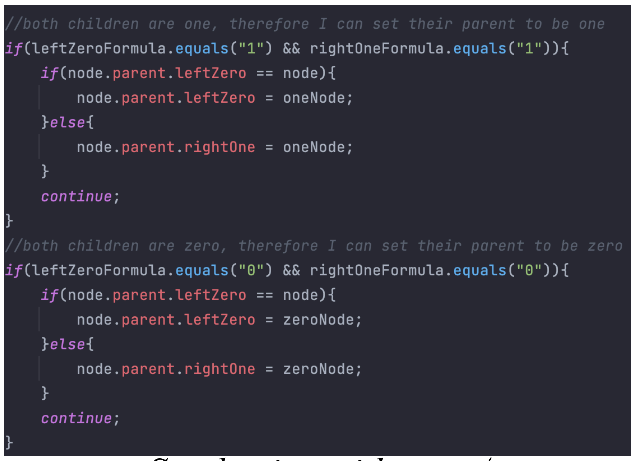
BDD_create

This method takes as arguments Boolean function and the order of variables. At first, I get the number of variables and calculate the expected number of nodes before the reduction with the formula. After this I initialize the HashMap, and two nodes. These nodes – zeroNode and oneNode are nodes which will exist in every diagram, so I have created universal nodes for 0 and 1 formulas. Then I create the root node and queue. I use this queue to save into it the new nodes for decomposition and I use iterative form to do this. The while loop performs until the queue is empty.



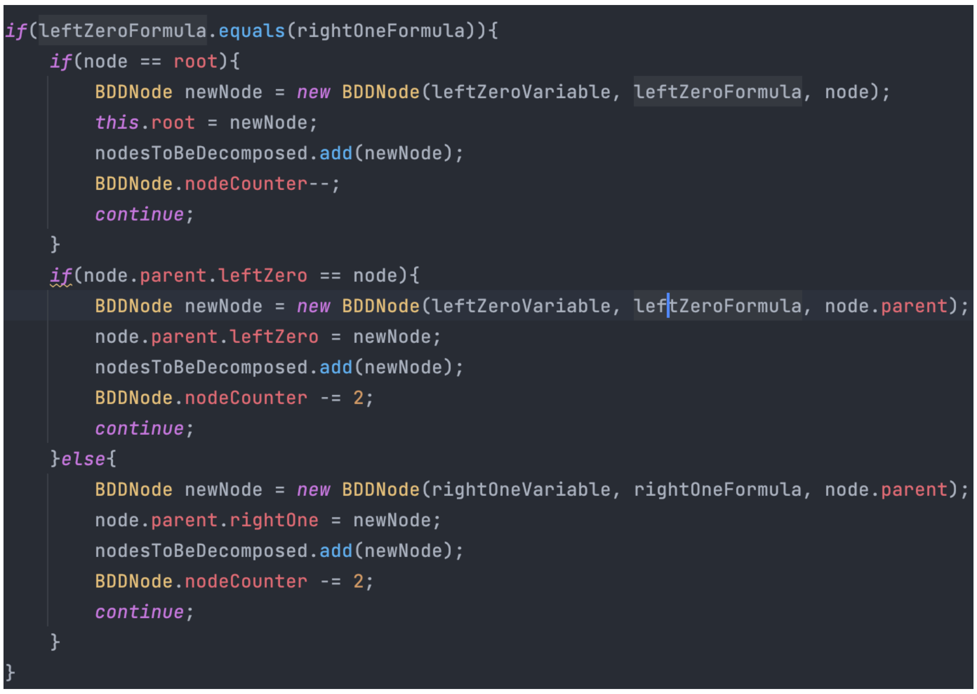
```
Queue<BDDNode> nodesToBeDecomposed = new LinkedList<>(); //I use
nodesToBeDecomposed.add(root); //adding first node to the queue
//the queue to hold the nodes to decompose
while(!nodesToBeDecomposed.isEmpty()){
    //get the next node to be decomposed from the queue
    BDDNode node = nodesToBeDecomposed.poll();
```

First thing in the loop is that I take out and delete element from the queue. I save the formula of this node to cache and create Booleans to keep track of duplicated nodes. I use method which I will explain down below to find the first variable in order. After this I create the decomposition and get the formula for child nodes. If both children are zero or one, I use **S** reduction and I set parent node of the node to one or zero, because the node with equal children is redundant. This also happens if both children are equal formula, then I set the child of parent of the node with equal children to be this formula and I skip that node. This is the type **S** reduction. In this reduction, if the child nodes are not 0 or 1 I also add them to queue because we need to decompose them.



```
//both children are one, therefore I can set their parent to be one
if(leftZeroFormula.equals("1") && rightOneFormula.equals("1")){
    if(node.parent.leftZero == node){
        node.parent.leftZero = oneNode;
    }else{
        node.parent.rightOne = oneNode;
    }
    continue;
}
//both children are zero, therefore I can set their parent to be zero
if(leftZeroFormula.equals("0") && rightOneFormula.equals("0")){
    if(node.parent.leftZero == node){
        node.parent.leftZero = zeroNode;
    }else{
        node.parent.rightOne = zeroNode;
    }
    continue;
}
```

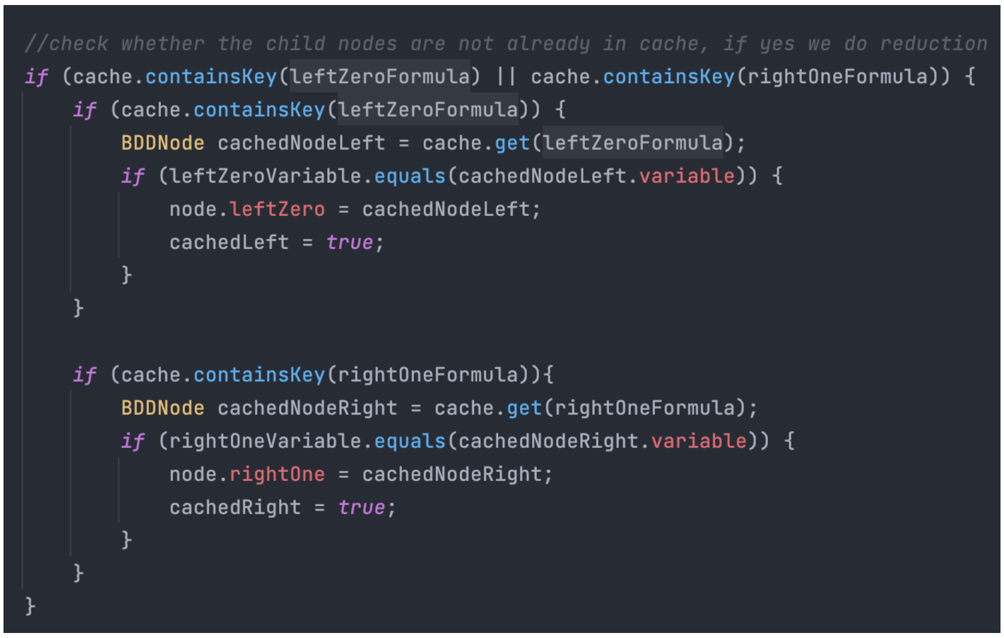
type S reduction with zeros/ones



```java
if(leftZeroFormula.equals(rightOneFormula)){
    if(node == root){
        BDDNode newNode = new BDDNode(leftZeroVariable, leftZeroFormula, node);
        this.root = newNode;
        nodesToBeDecomposed.add(newNode);
        BDDNode.nodeCounter--;
        continue;
    }
    if(node.parent.leftZero == node){
        BDDNode newNode = new BDDNode(leftZeroVariable, leftZeroFormula, node.parent);
        node.parent.leftZero = newNode;
        nodesToBeDecomposed.add(newNode);
        BDDNode.nodeCounter -= 2;
        continue;
    }else{
        BDDNode newNode = new BDDNode(rightOneVariable, rightOneFormula, node.parent);
        node.parent.rightOne = newNode;
        nodesToBeDecomposed.add(newNode);
        BDDNode.nodeCounter -= 2;
        continue;
    }
}
```

type S reduction

After this I perform the reduction **I**, where I check whether the formula with same decompose variable isn't already located in HashMap. If yes, I set these nodes equal and I don't have to create any new nodes. I set Booleans cached right/left to true if this happen, so I don't have to create new nodes.
After this I check whether the formula for some of the children isn't zero or one. If yes, I set it to be the zeroNode or oneNode and again I don't have to create new nodes. After this if the child node is null, and it hasn't been cached or something, I create a new node and I put it in the cache and add it to the queue as well.
I also count the nodes, so in the end, I have stored the root, number of nodes, amount of variables and nodes before reduction.



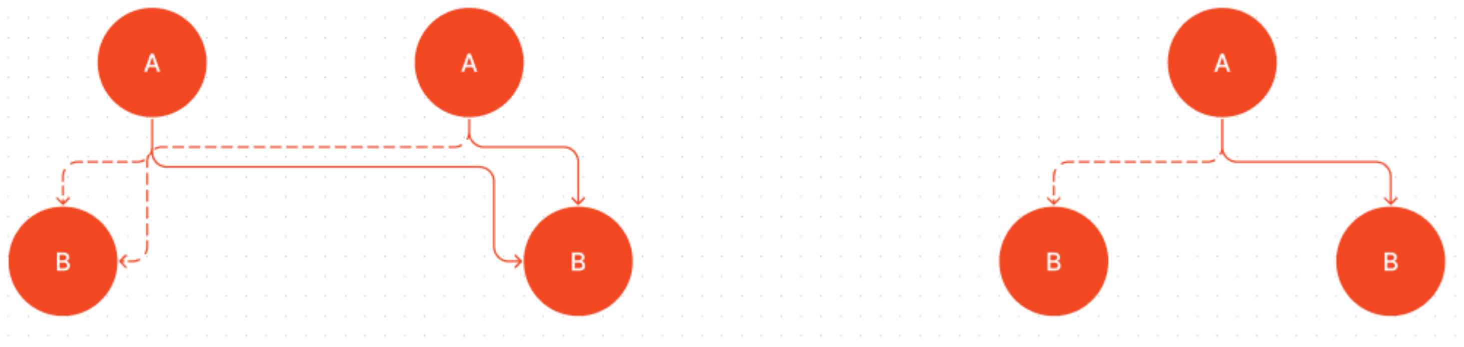
```java
//check whether the child nodes are not already in cache, if yes we do reduction
if (cache.containsKey(leftZeroFormula) || cache.containsKey(rightOneFormula)) {
    if (cache.containsKey(leftZeroFormula)) {
        BDDNode cachedNodeLeft = cache.get(leftZeroFormula);
        if (leftZeroVariable.equals(cachedNodeLeft.variable)) {
            node.leftZero = cachedNodeLeft;
            cachedLeft = true;
        }
    }

    if (cache.containsKey(rightOneFormula)){
        BDDNode cachedNodeRight = cache.get(rightOneFormula);
        if (rightOneVariable.equals(cachedNodeRight.variable)) {
            node.rightOne = cachedNodeRight;
            cachedRight = true;
        }
    }
}
```

type I reduction

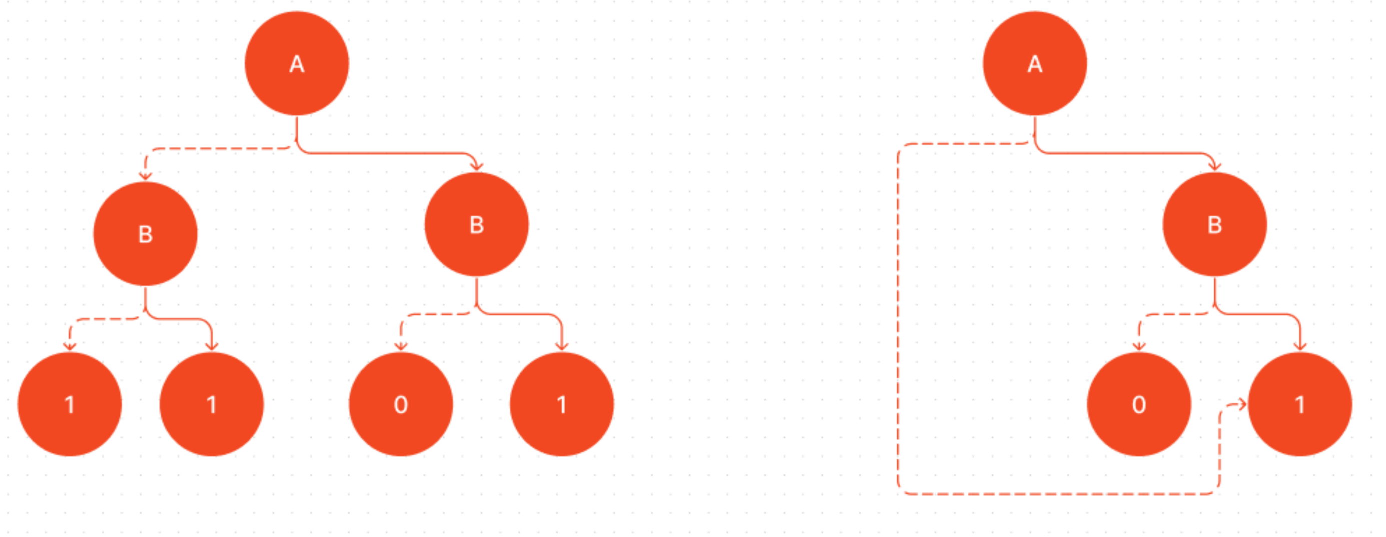
Reductions

Type S reduction



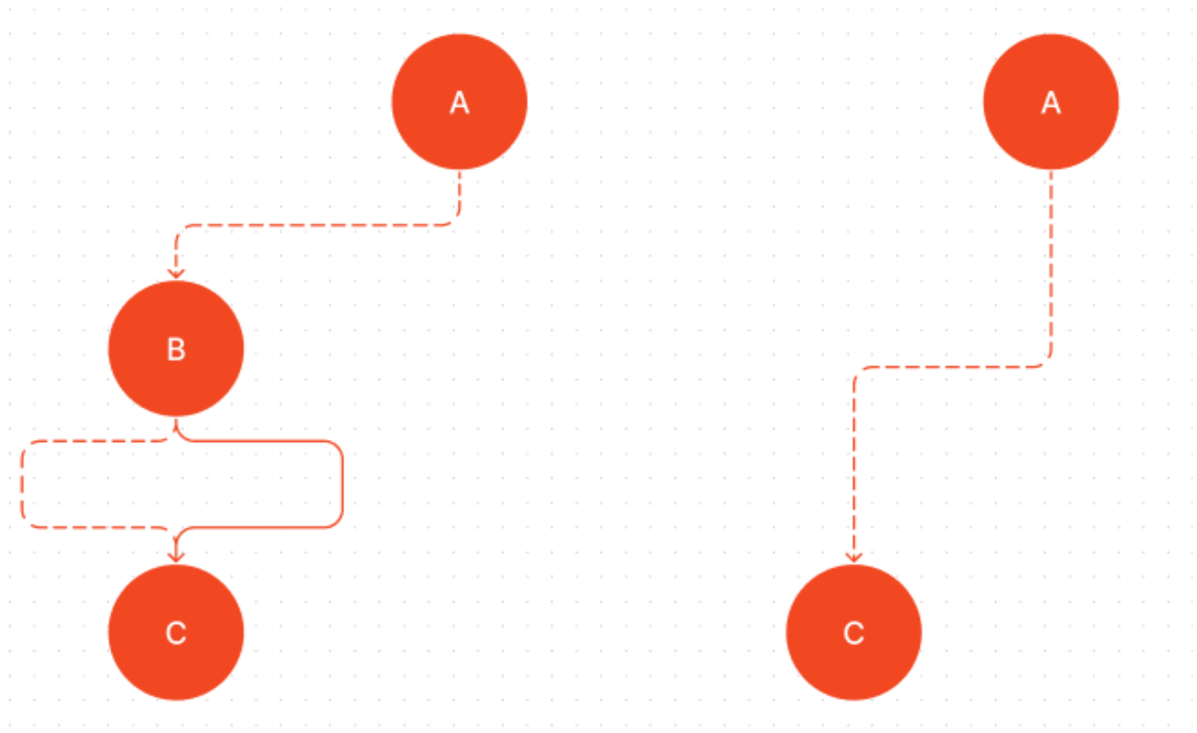
nodes with same formula and decompose variable

Type I reduction



the left B node is redundant

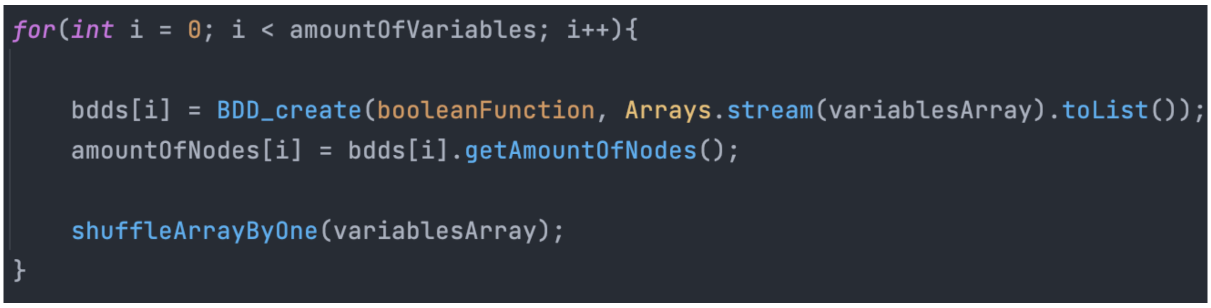
Type I reduction with equal formula



node B is redundant

BDD_create_with_best_order

In this method, firstly I get the number of variables, because I will create that many trees and return the one with the best order. Then I create two arrays with this size, one for the BDDs and one for the amounts of nodes in each one. After this I create the hashset for the variables, because in the set there is variable only once, so I get all of them in here. I go through the string and add it to the hashset. Right after I turn it into array, including all the variables from Boolean function. Then I use for loop which will be performed amount of variables times. I use there the *bdd_create* method and I save the tree into the array. In each iteration I shuffle the array (I am creating permutations in method *shuffleArrayByOne*).



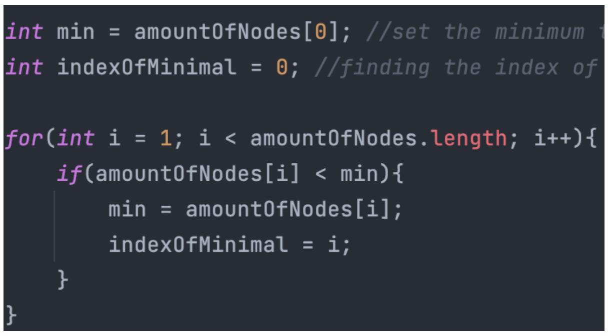
```
for(int i = 0; i < amountOfVariables; i++){

    bdds[i] = BDD_create(booleanFunction, Arrays.stream(variablesArray).toList());
    amountOfNodes[i] = bdds[i].getAmountOfNodes();

    shuffleArrayByOne(variablesArray);
}
```

for loop to create multiple trees

After the for loop is performed, I find the index of the smallest integer in the amount of nodes array. On the same index in the BDDs array, there is the tree with smallest number of nodes. In the end I return this tree and print the best order and number of nodes.



```
int min = amountOfNodes[0]; //set the minimum
int indexOfMinimal = 0; //finding the index of

for(int i = 1; i < amountOfNodes.length; i++){
    if(amountOfNodes[i] < min){
        min = amountOfNodes[i];
        indexOfMinimal = i;
    }
}
```

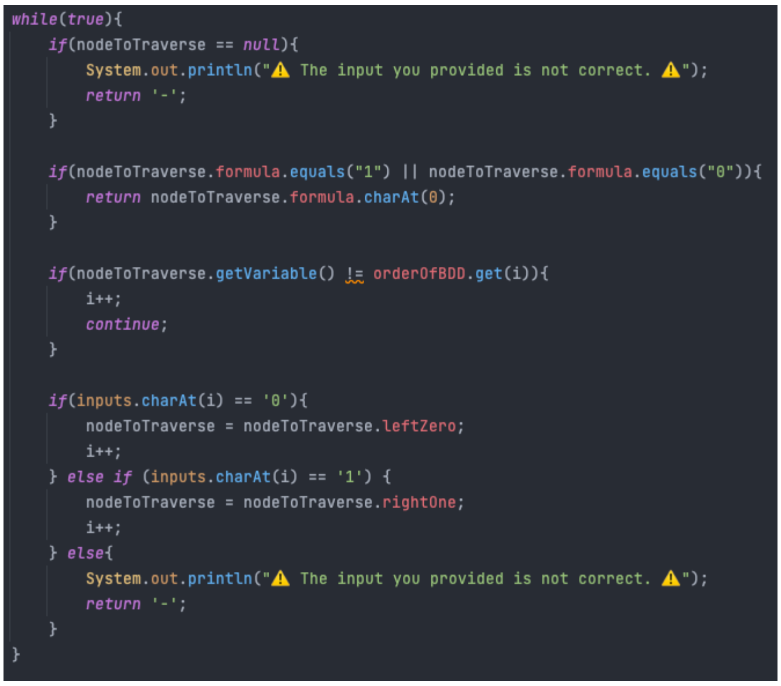
for loop to find best ordered tree

In the end there is `return bdds[indexOfMinimal]`

BDD_use

This method has the type of character, it returns only 1 or 0 based on what the output of the Boolean function is (it also may return error if the input is not correct). The argument for this method is binary decision diagram we are working with and the string which consists of zeros and one, representing *false* and *true*.

Firstly, I check whether input is correct. Then I set the *nodeToTraverse* to be equal to the root. Right after I create for each loop, to iterate throughout the tree based on the number in the input (I change the string to array). If the number on index i is zero I go to the left, if one I go to right subtree.



```java
while(true){
    if(nodeToTraverse == null){
        System.out.println("⚠ The input you provided is not correct. ⚠");
        return '-';
    }

    if(nodeToTraverse.formula.equals("1") || nodeToTraverse.formula.equals("0")){
        return nodeToTraverse.formula.charAt(0);
    }

    if(nodeToTraverse.getVariable() != orderOfBDD.get(i)){
        i++;
        continue;
    }

    if(inputs.charAt(i) == '0'){
        nodeToTraverse = nodeToTraverse.leftZero;
        i++;
    } else if (inputs.charAt(i) == '1') {
        nodeToTraverse = nodeToTraverse.rightOne;
        i++;
    } else{
        System.out.println("⚠ The input you provided is not correct. ⚠");
        return '-';
    }
}
```

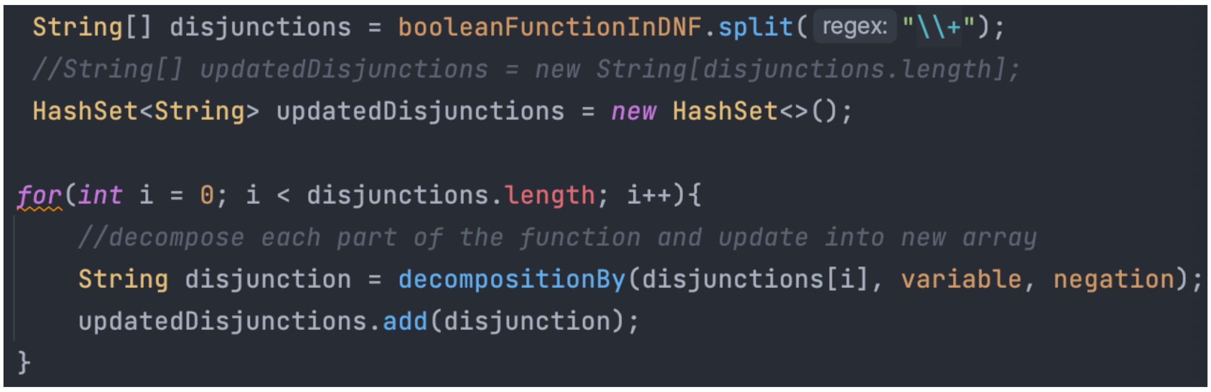
for loop to traverse through the tree

BDD use looks like this because, sometimes the reductions may cause complications, when some node is redundant and it is skipped, and in that case, the order and variable in the node are not equal. It his happens we need to increment the index, because of the skipped node. If the formula is zero or one, we are in the final node. If not, we continue to traverse the diagram, based on the input 0 – left, 1 – right. If the node is null or we have short input, or there are not legal symbols in input, the return value is '-' and error statement is printed.

Helper methods

fullDecomposition

This method consists of another two methods, but it is really simple process how to effectively decompose a Boolean function. The arguments for this method are Boolean function in DNF, variable to decompose by and Boolean value whether we are decomposing by negated or not-negated variable.
Firstly, I split the DNF by the plus signs and I decompose each part separately in method ***decomposition By***. After each decomposition I add the edited part of DNF into the HashSet, so there won't be any duplicates. In the end, I create an array from the HashSet, and I call ***createDisjunctions***.



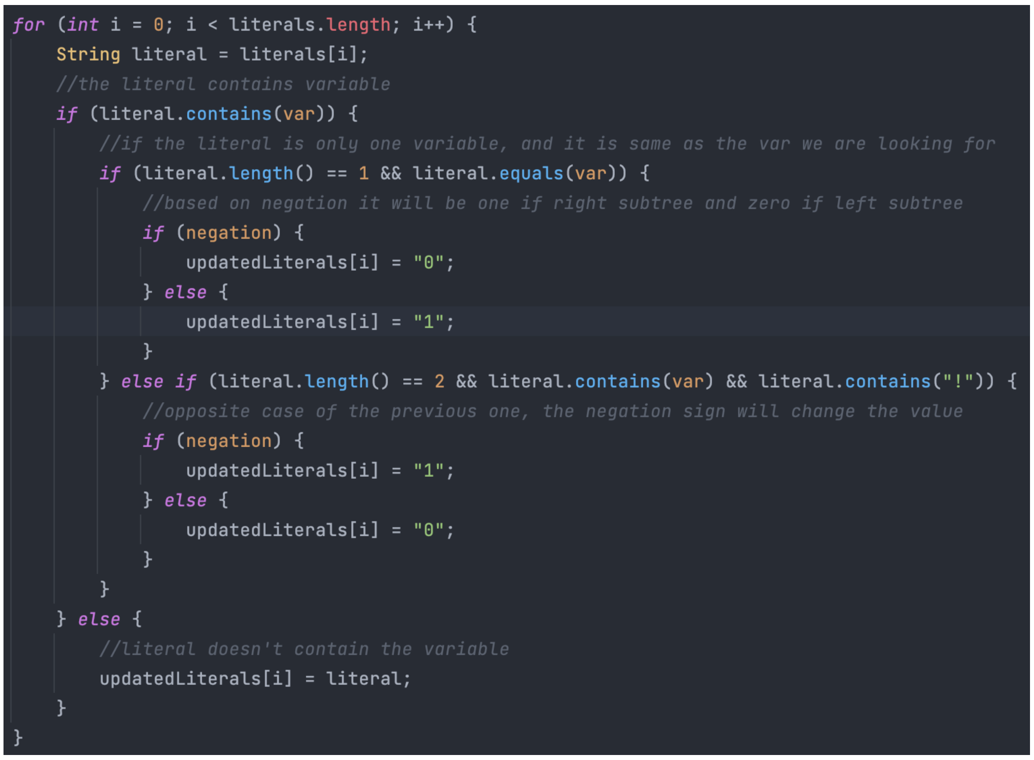
```
 String[] disjunctions = booleanFunctionInDNF.split( regex: "\\+");
 //String[] updatedDisjunctions = new String[disjunctions.length];
 HashSet<String> updatedDisjunctions = new HashSet<>();

for(int i = 0; i < disjunctions.length; i++){
    //decompose each part of the function and update into new array
    String disjunction = decompositionBy(disjunctions[i], variable, negation);
    updatedDisjunctions.add(disjunction);
}
```

decomposing each conjunction

decompositionBy

This method takes as an argument conjunction, variable to decompose by and negation of the variable. If the conjunction doesn't contain the variable for decomposition, we return it back. If it does, we split it by multiplying signs into literals. After this I create the updated literals array, and I add there the decomposed literals. If the literal contains the variable, I check whether it is only the variable itself or whether it is with negation sign and based on whether we are going left or right I return value.
After editing every unit, I go through updated literals, and it there are only 1s, I return 1 as a result. If not, I check whether there is at least one zero. If yes, the whole conjunction is zero and I return it. It there is variable I add it with multiplying sign and I also add exclamations marks for negations. In the end, if the formula ends with multiplying sign, I delete it because it is the additional sign in the end. I return the result.

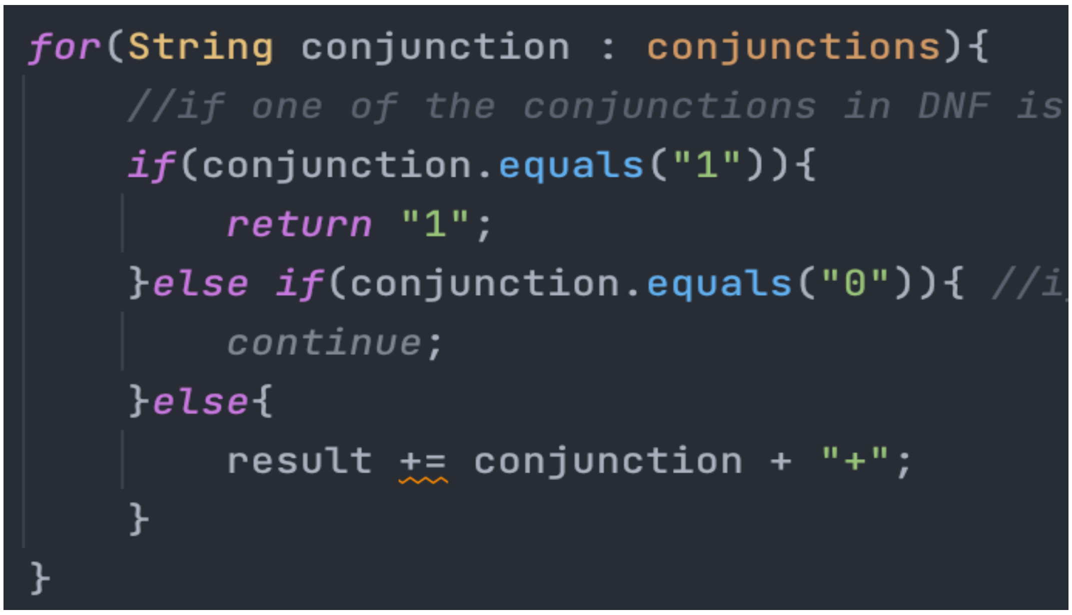


```java
for (int i = 0; i < literals.length; i++) {
    String literal = literals[i];
    //the literal contains variable
    if (literal.contains(var)) {
        //if the literal is only one variable, and it is same as the var we are looking for
        if (literal.length() == 1 && literal.equals(var)) {
            //based on negation it will be one if right subtree and zero if left subtree
            if (negation) {
                updatedLiterals[i] = "0";
            } else {
                updatedLiterals[i] = "1";
            }
        } else if (literal.length() == 2 && literal.contains(var) && literal.contains("!")) {
            //opposite case of the previous one, the negation sign will change the value
            if (negation) {
                updatedLiterals[i] = "1";
            } else {
                updatedLiterals[i] = "0";
            }
        }
    } else {
        //literal doesn't contain the variable
        updatedLiterals[i] = literal;
    }
}
```

the for loop for editing each literal as I mentioned previously

createDisjunctions

This method takes an array of conjunctions as an input and it the length is only one, it returns the only one conjunction. If it is longer, I go through all of the elements. If the element is 1, I return one because if one conjunction is true, the result is true. If it is zero, I ignore it, it won't affect the method. If it is a variable, I add it with the plus sign. In the end I erase the last additional plus sign and return the resulting DNF.

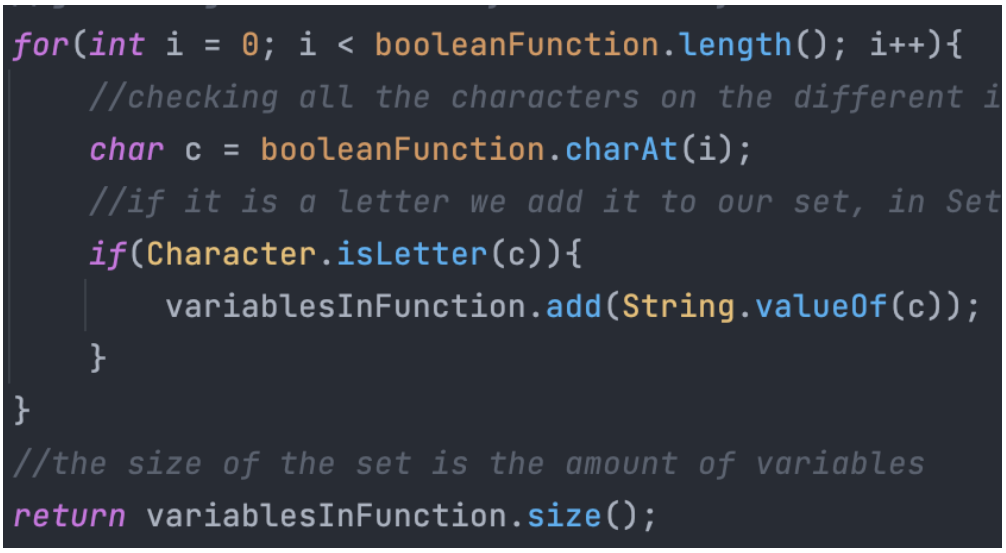


```java
for(String conjunction : conjunctions){
    //if one of the conjunctions in DNF is
    if(conjunction.equals("1")){
        return "1";
    }else if(conjunction.equals("0")){ //i
        continue;
    }else{
        result += conjunction + "+";
    }
}
```

for loop creating the DNF

getAmountOfVariables

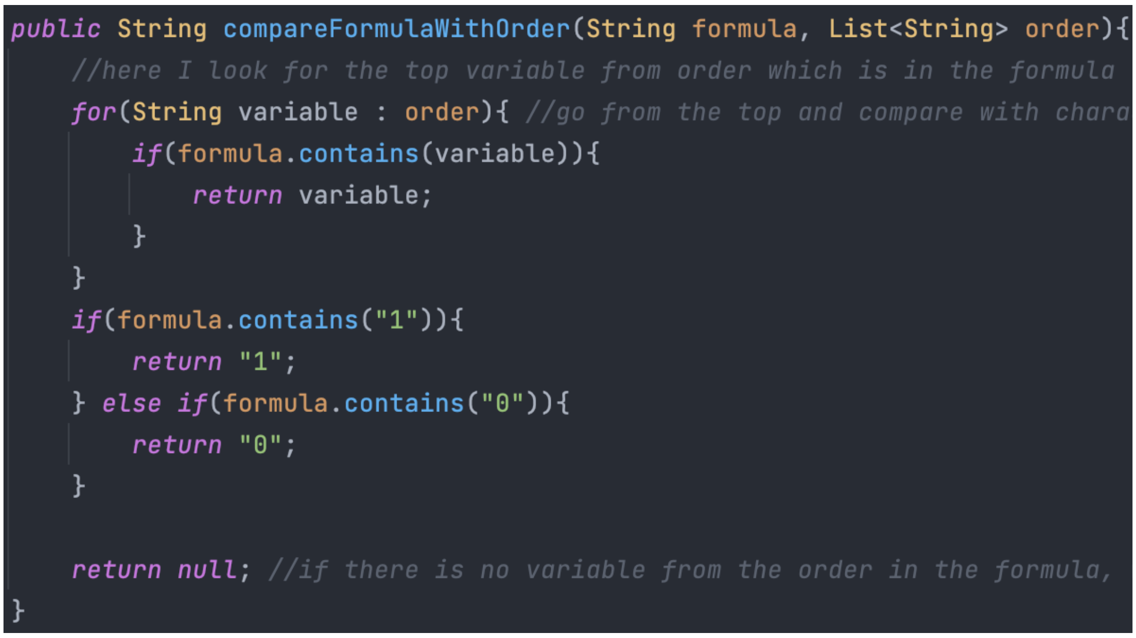
This method takes Boolean function as an input, and it returns amount of variables in here. It is a simple method, which uses HashSet, which store each value only once, that's why it returns the correct number of different signs.



```
for(int i = 0; i < booleanFunction.length(); i++){
    //checking all the characters on the different i
    char c = booleanFunction.charAt(i);
    //if it is a letter we add it to our set, in Set
    if(Character.isLetter(c)){
        variablesInFunction.add(String.valueOf(c));
    }
}
//the size of the set is the amount of variables
return variablesInFunction.size();
```

compareFormulaWithOrder

This method gets formula and order as an input, and it returns the highest variable in the order which is in the formula. This method uses for each loop, and if the formula is 0 or 1 it returns these values. If it doesn't contain any of the variables, it returns null.

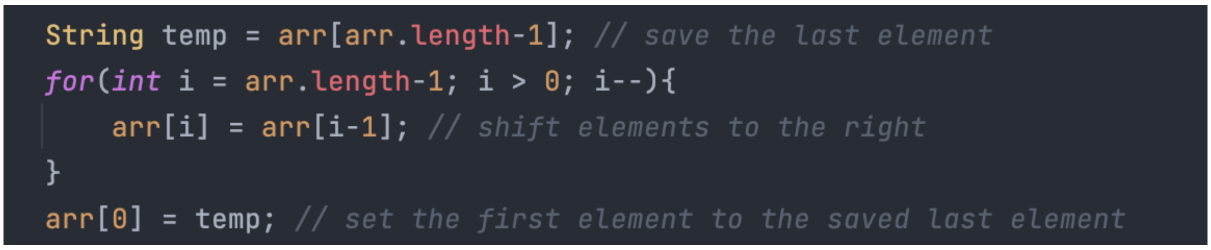


```
public String compareFormulaWithOrder(String formula, List<String> order){
    //here I look for the top variable from order which is in the formula
    for(String variable : order){ //go from the top and compare with chara
        if(formula.contains(variable)){
            return variable;
        }
    }
    if(formula.contains("1")){
        return "1";
    } else if(formula.contains("0")){
        return "0";
    }

    return null; //if there is no variable from the order in the formula,
}
```

shuffleArrayByOne

This method is used for creating permutations from the array of order. We save the last index to temporary and in the end assign it as first (index 0). Then we shift every element to the right,



```
String temp = arr[arr.length-1]; // save the last element
for(int i = arr.length-1; i > 0; i--){
    arr[i] = arr[i-1]; // shift elements to the right
}
arr[0] = temp; // set the first element to the saved last element
```

BinaryTreePrinter

This is the method I found on the internet, and it is not my method. It is not used in the process of creating or using binary decision diagram. It is only used for better visualisation, so I can see whether my binary decision diagram works correctly. As I said, this method is not obligatory, and it is there only for better testing.

Source:
https://github.com/eugenp/tutorials/blob/master/data-structures/src/main/java/com/baeldung/printbinarytree/BinaryTreePrinter.java

Creating Boolean functions and measuring time

***BooleanFunctionGenerator* is class used for testing.** It works with random library. There are two main methods – one to create random Boolean function and one to create the order. To generate order, I just add to the order the first N variables alphabetically and shuffle the List, because when creating functions, I always go alphabetically. In create function method, I create random amount (1 to 5) conjunctions, consisting of N different variables and possible negations.
There are also two **generateBDD** methods, that work with the methods mentioned above. Their purpose is to create BDD with random function and order 100 times and calculate the average of the time to create one diagram and also average amount of nodes.

The memory and time complexity of my BDD

Memory complexity

The space/memory complexity of my **BDD_create** method is O(2^n). This is because the memory requirements of my BDD depend on the amount of nodes in the graph. Amount of nodes in my diagram are dependent on the Boolean function – by this I mean amount of variables inside and it depends on the order as well, which may cause really huge difference.
In **BDD_create_with_best_order** the diagram is also dependent on the amount of nodes, however in this method, the tree with the smallest amount of nodes is returned, so it will return the tree, that will cost us the smallest amount of memory. However, throughout the process we create N (number of variables) trees, which may take up a lot of memory.

Time complexity

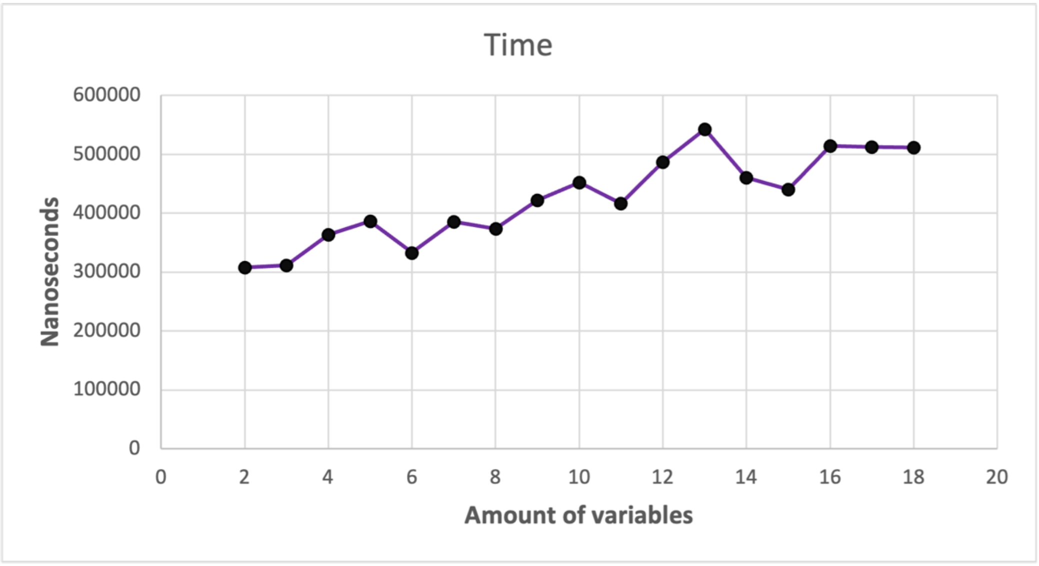
As for the memory complexity, the time complexity in **BDD_create** depends on the amount of nodes in the diagram. Based on order and on the amount of variables, the time complexity is in worst case O(2^n), where **n** is amount of variables.
Time complexity in **BDD_create_with_best_order** for the final tree will be the one of the better scenarios for the Boolean function we use, because we won't test every single order, but we test a few orders and the tree with the most effective one will be the one returned. However, this method will take a longer time to be performed, due to a fact that there will be N different trees made.

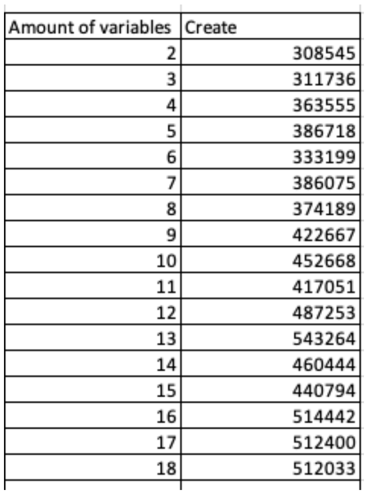
Testing correctness of my BDD

The initial testing for my BDD tree was printing it and drawing it myself, that's how I tested the correctness of my binary decision diagram. I also used Karnaugh maps to check whether my output was correct. Another type of testing correctness was comparing my BDD with my classmates. We used the same formula and then we have checked whether the output is similar. This way I found out many mistakes and errors I made and corrected it. Right now, my code seems to be working correctly based on the many tests I have performed. For BDD use I used Karnaugh maps as well, to find out, whether it works correctly.

Graphs and tables for *BDD_create*

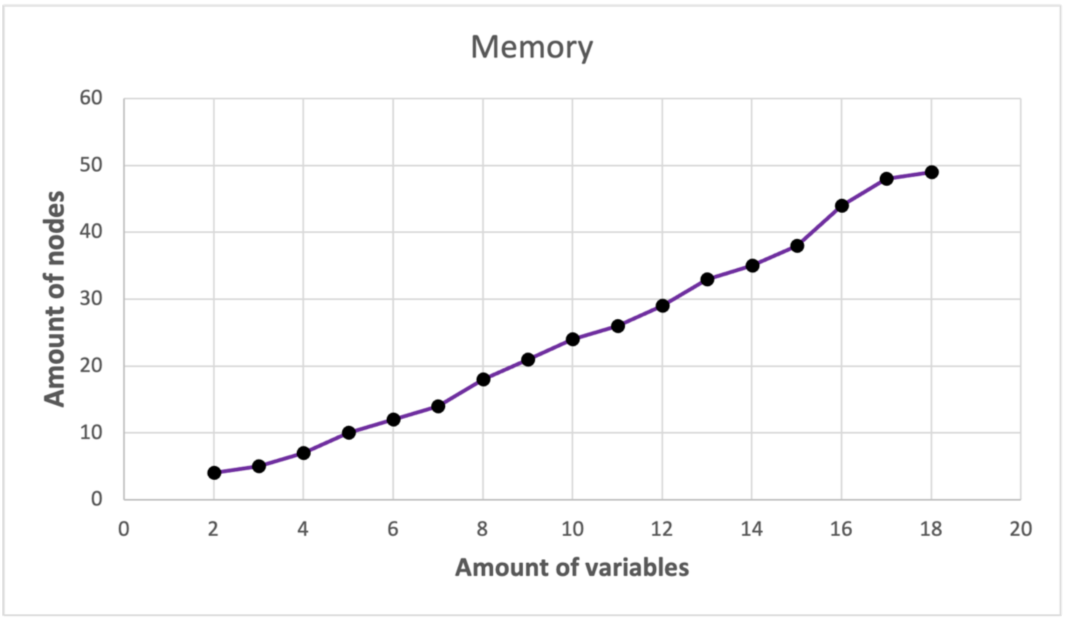
To create these graphs, I used method that I mentioned earlier. I have created 100 Boolean functions, and random orders and used them to create BDDs. Then I counted the average time to create diagram, average amount of nodes for BDDs and reduction rate. I have created 100 BDDs for every number of variables from 2 to 18. So, the graph represents time/space/reduction rate when the amount of variables is rising. The **maximal** amount of variables I used was 26 (whole alphabet of uppercase variables).





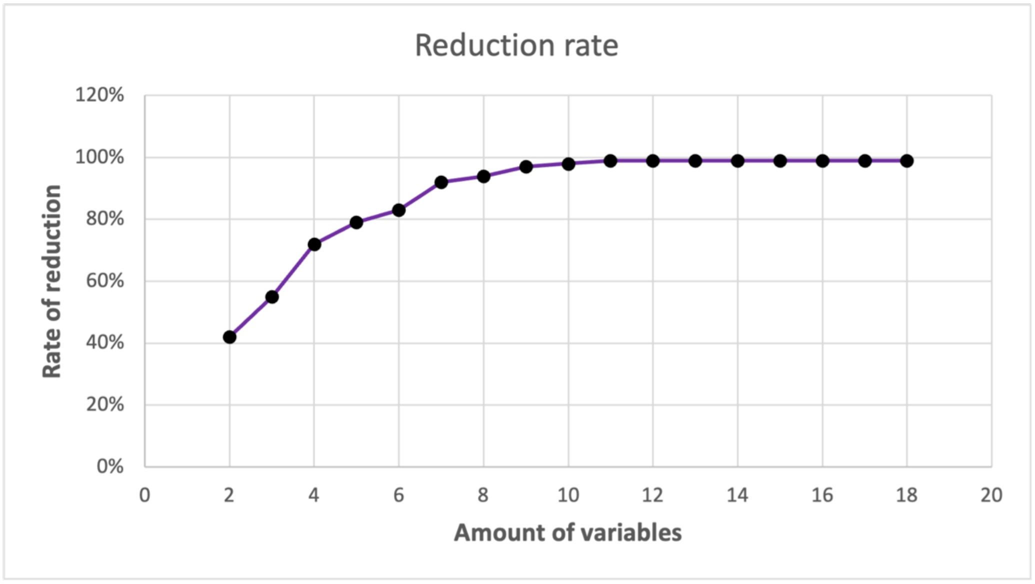
Amount of variables	Create
2	308545
3	311736
4	363555
5	386718
6	333199
7	386075
8	374189
9	422667
10	452668
11	417051
12	487253
13	543264
14	460444
15	440794
16	514442
17	512400
18	512033

This is the graph of the time complexity in **nanoseconds** for my BDD_create method. The graph looks like this, because we are generating random functions with random order, that’s why the graph looks like this.



Amount of variables	Create
2	4
3	5
4	7
5	10
6	12
7	14
8	18
9	21
10	24
11	26
12	29
13	33
14	35
15	38
16	44
17	48
18	49

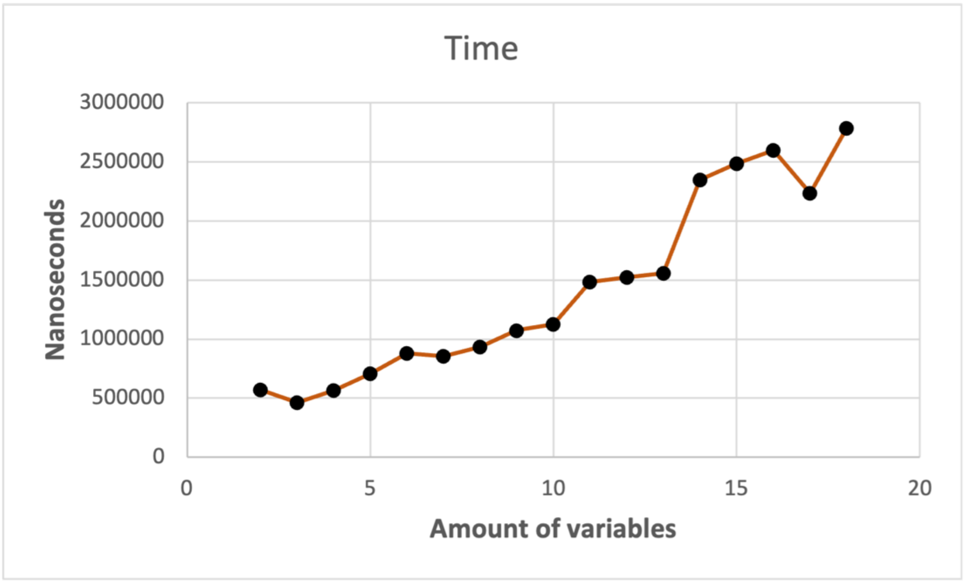
This is the graph of the average memory used for one diagram. I have counted all the nodes (after reduction) and divided it by 100, for every amount of variables. This is the final graph. With rising amount of variables, the amount of nodes is rising. The graph is almost linear.



Amount of variables	Create
2	42%
3	55%
4	72%
5	79%
6	83%
7	92%
8	94%
9	97%
10	98%
11	99%
12	99%
13	99%
14	99%
15	99%
16	99%
17	99%
18	99%

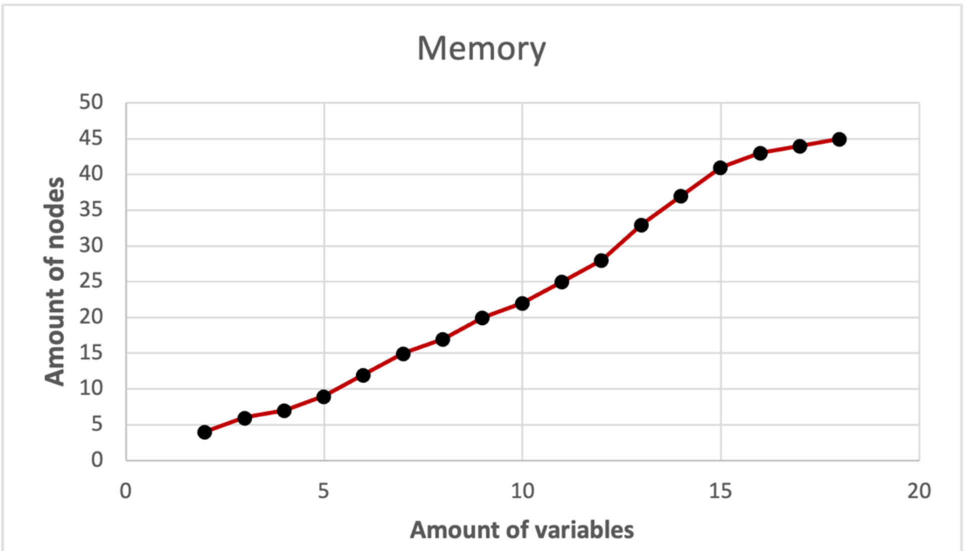
This is the graph of the reduction rate. The more variables there is, the more nodes there is and that's why there are many duplicates, which are supposed to be reduced.

Graphs for BDD_create_with_best_order



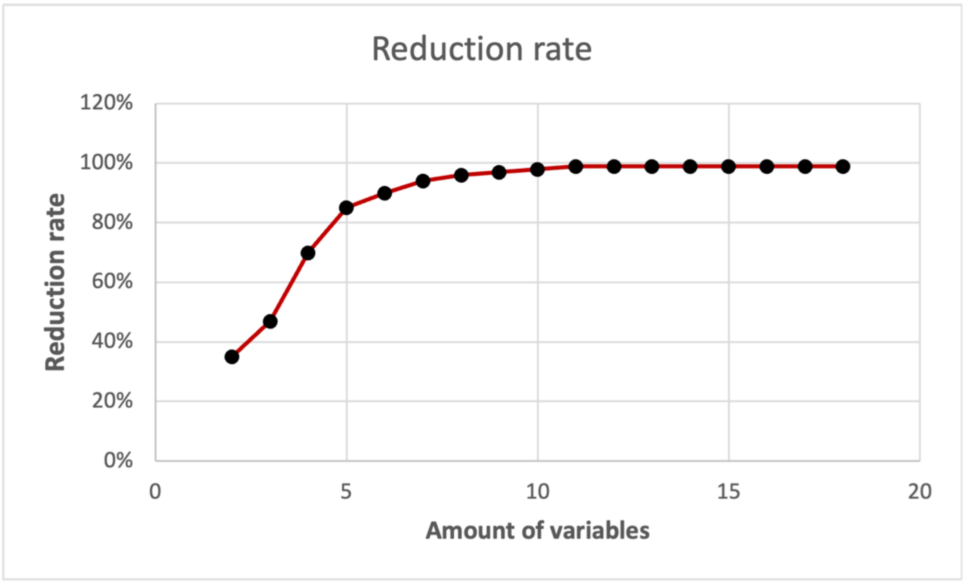
Amount of variables	Best order
2	571876
3	462062
4	563650
5	707767
6	878824
7	855947
8	932105
9	1074045
10	1123338
11	1483357
12	1521960
13	1555590
14	2348819
15	2487791
16	2598482
17	2233792
18	2785342

Time is in **nanoseconds**. The average time to create N trees, where N is amount of variables is significantly higher than in the previous graph (BDD_create). This is because we are creating more diagrams with increasing amount of variables. We always return the diagram with best order from the BDD orders we tried.



Amount of variables	Best order
2	4
3	6
4	7
5	9
6	12
7	15
8	17
9	20
10	22
11	25
12	28
13	33
14	37
15	41
16	43
17	44
18	45

Memory, used to create the BDD (amount of nodes after reduction) is rising linearly with the amount of variables, because with more variables, there are more nodes. We always return the tree with smallest amount of nodes from the orders we tried. That's why for most cases, the memory is lower in BDD with best order than in regular BDD_create.



Amount of variables	Best order
2	35%
3	47%
4	70%
5	85%
6	90%
7	94%
8	96%
9	97%
10	98%
11	99%
12	99%
13	99%
14	99%
15	99%
16	99%
17	99%
18	99%

Similarly, to the BDD_create, even here we can see, that the more nodes we have, the bigger reduction rate there is. That's because there is a lot of redundant nodes, because formulas are repeating, or the child nodes are equal.

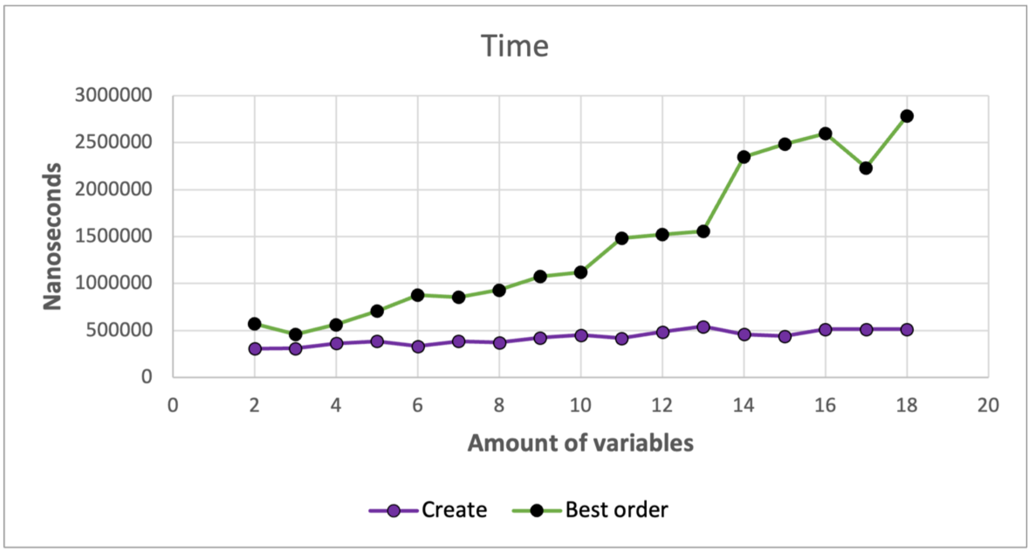
BDD_USE

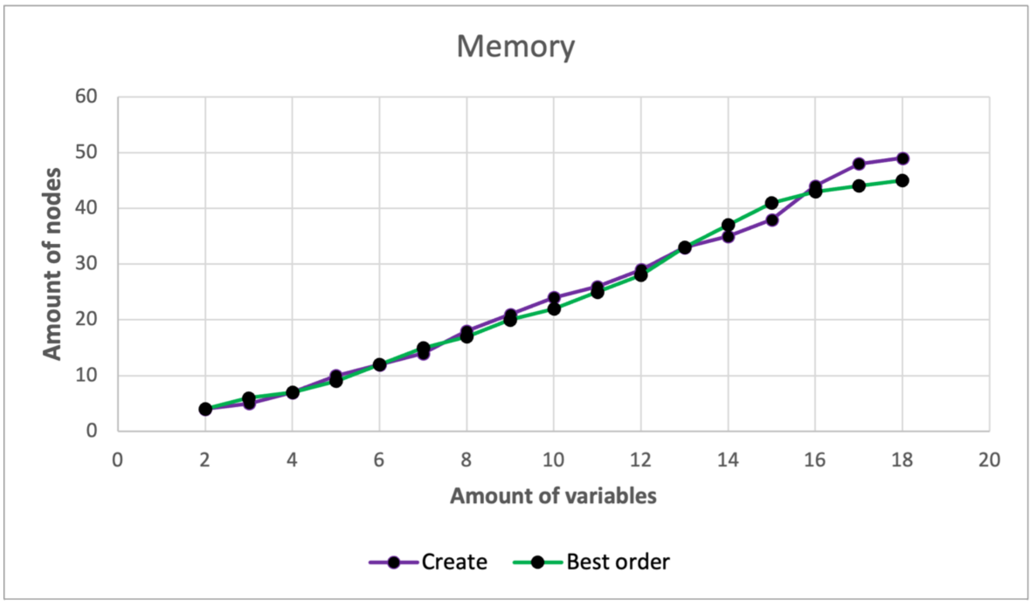
BDD, is a method that depends on the binary decision diagram (its amount of nodes). This method traverses from the root to the result which may be one or zero. The amount of **memory** it takes would be probably only two bytes (Java uses Unicode) because we only return one character. The **time complexity** is based on the size of the tree.

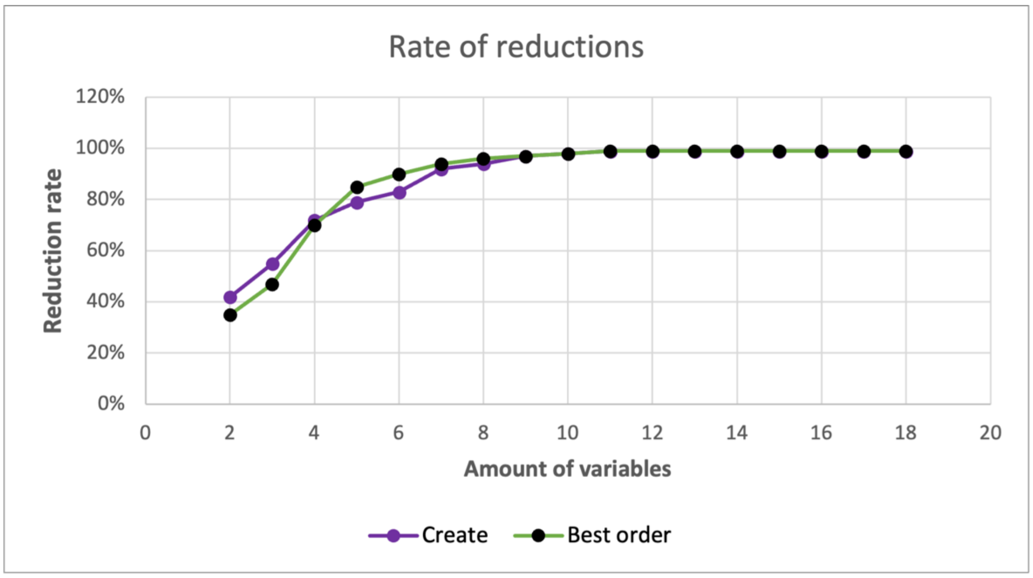
Conclusion and comparison

The binary decision diagram is a great way, how we can solve Boolean function, and find out all the possible outcomes. It is effective data structure, which could find use in many fields, not only in computer science. In my implementation there are two function to construct BDD. The original is BDD_create, BDD_create_with_best_order only calls BDD create N times, where N is amount of variables. There is also BDD_use to find the outcome of given input.
Throughout the testing I found out, that even though the BDD with best order takes **longer (talking about time)** to perform, it is more effective in the **memory** way, because there is a higher chance of getting better order, than in the single calling of BDD_create. The **reduction rate** is slightly better in the BDD with best order, because there must be more reductions, for the BDD to have less nodes.
So when comparing the binary decision diagram from BDD_create and create_with_best_order, I would say, the one with best order is better, even though it takes longer to create N trees, the one which is created has higher chance of being much more effective.

The visual comparison of the functions




Memory
Amount of nodes
Amount of variables
Create
Best order


Rate of reductions
Reduction rate
Amount of variables
Create
Best order

Testing file

For testing how my program works, there is a program called testing, in here you just run it, and choose from 3 options. Number one is to test and measure the time and memory for the method BDD_create. After choosing this option, you are asked to input the amount of Boolean functions (amount of trees) you would like to create and amount of variables you would like to use (maximum 26 because that's the whole alphabet of upper-case letters. Output is the print for each tree – it prints out the Boolean function, order, and then information about each tree. Then it sends the overall statistics like average time for creating one tree, average amount of nodes in the trees. Then there is the option two that works the same way, but it is for create with best order.
Third option is for the **use** method and it is designed in the way, that you input the order and the Boolean function, after this the binary decision diagram is created, and also printed so you can check afterwards the output of the **use** method. After this you are asked to input the 0 and 1 values into a String and the returned value is the corresponding output in the tree. As I said you cas easily check it in the text file "data.txt" in which the diagram is printed by the ***BinaryTreePrinter*** method.

References

The pictures of code were mine, and also the reduction picture has been created by me.
DSA lectures
https://www.geeksforgeeks.org/binary-decision-diagram/
https://people.eecs.berkeley.edu/~sseshia/219c/lectures/BinaryDecisionDiagrams.pdf
https://www.youtube.com/watch?v=_C9yLZ2BGBA&t=11s&ab_channel=LastBencherComrade
https://github.com/eugenp/tutorials/blob/master/data-structures/src/main/java/com/baeldung/printbinarytree/BinaryTreePrinter.java