DSA – Zadanie 2

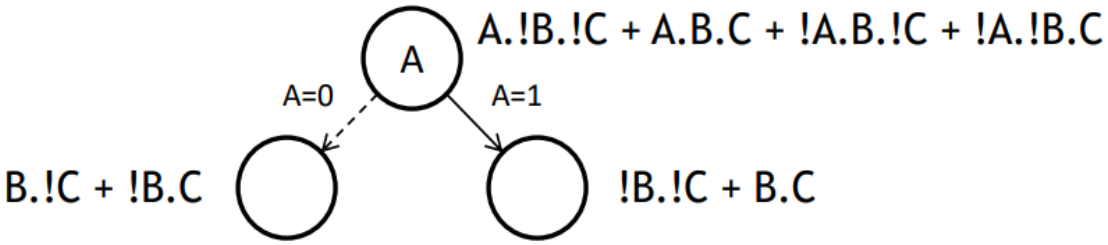
Hredzhev Oleksandr

ID: 123560

Binary Decision Diagram (BDD)

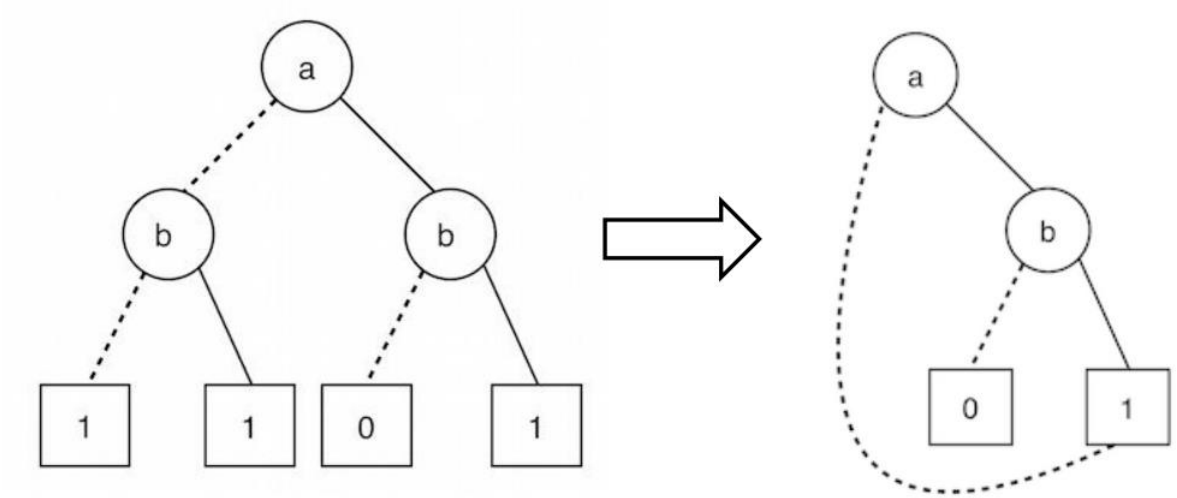
A binary decision diagram (BDD) is a data structure based on a binary tree that is used to represent a Boolean function via using the Shannon expansion.

A Binary Decision Diagram (BDD) is a graph in which each node represents a variable in a Boolean function, and the edges represent the possible values of that variable (true or false). Thus, each path from the root to a leaf node defines a set of variable values in the Boolean function that determines the function's value.

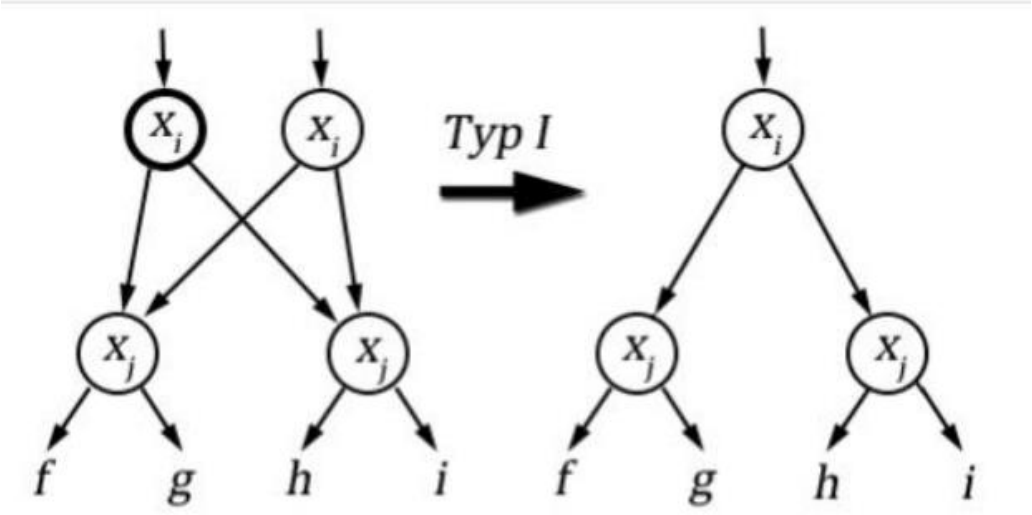


Reduction

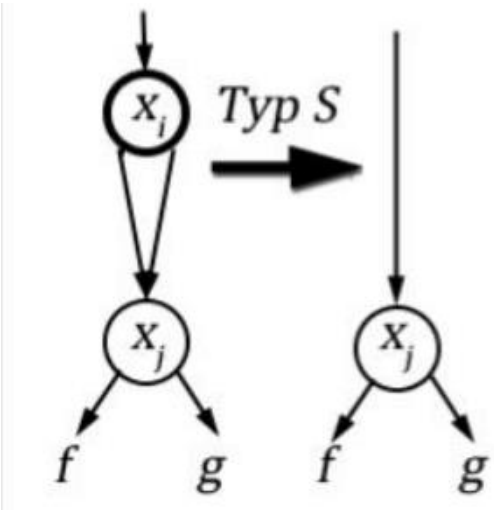
Visualization of connection to end nodes, since there are only two of them - 1 and 0:



Type I - removal redundant nodes by comparing pairs:



Type S - removal unnecessary nodes by comparing his descendants:



Realization

The implementation of the node is implemented exactly the same as in my previous project, but with some changes. A node has a link to the left and right nodes (which are children), as well as a list of parent nodes, since during reductions a node can have multiple parents.

The Format function changes the appearance of the function by changing !A -> a for further operations:

```
1 usage
protected static String format(String function) {
    function = function.replaceAll( regex: "\\s+",  replacement: "");

    Pattern pattern = Pattern.compile( regex: "!([A-Z])");
    java.util.regex.Matcher matcher = pattern.matcher(function);

    return matcher.replaceAll(match -> match.group().toLowerCase().substring( beginIndex: 1));
}
```

Method prepareData modifies the function string according to the rules of Boolean algebra:

```
2 usages
protected static String prepareData(String input) {
    List<String> functionVariables = input.chars() IntStream
            .mapToObj(i -> (char) i) Stream<Character>
            .filter(c -> c >= 'A' && c <= 'Z')
            .distinct()
            .sorted()
            .map(String::valueOf) Stream<String>
            .toList();

    List<String> filtered = Arrays.stream(input.split( regex: "\\+")).filter(c -> !c.contains("0")).toList();

    if (filtered.stream().anyMatch(c -> c.matches( regex: "1+"))) {
        return "1";
    }

    String cleaned = filtered.isEmpty() ? "0" : filtered.stream().distinct() Stream<String>
            .collect(Collectors.joining( delimiter: "+")) String
            .replace( target: "1",  replacement: "");

    filtered = Arrays.stream(cleaned.split( regex: "\\+")).toList();

    for ( String variable : functionVariables) {
        if (filtered.contains(variable) && filtered.contains(variable.toLowerCase())) {
            return "1";
        }
    }

    return cleaned;
}
```

The buildTree iReduction method builds a tree and includes I-reduction.

Here, in the formula, the letters of the current queue are replaced by 0 or 1, depending on whether the child formula is left or right:

```
String left = function.replace(order.charAt(0),  newChar: '0').replace(Character.toLowerCase(order.charAt(0)),  newChar: '1');
String right = function.replace(order.charAt(0),  newChar: '1').replace(Character.toLowerCase(order.charAt(0)),  newChar: '0');
```

This method also includes a filter that clears the current formula from identical parts of the expression:

```
if (!left.equals("1") && !left.equals("0")) {

    left = Arrays.stream(left.split( regex: "\\+"))
            .filter(clause -> clause != null && !clause.isEmpty())
            .map(clause -> clause.chars() IntStream
                    .mapToObj(c -> (char) c) Stream<Character>
                    .filter(c -> (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
                    .distinct()
                    .sorted()
                    .map(String::valueOf) Stream<String>
                    .collect(Collectors.joining( delimiter: ""))
            )
            .sorted()
            .distinct()
            .collect(Collectors.joining( delimiter: "+"));
}
```

The next node is compared with those previously added in the tree, if the node is already in the tree, then it is carried out with S-reduction, if this is not the case, the given node is not final, then the operation is repeated recursively:

```
if (!right.equals("0") && !right.equals("1")) {
    if (!containsRight) {
        buildTree_iReduction(rightNode, order.substring( beginIndex: 1));
    } else {
        sReduction(rightNode);
    }
}
```

Reduction is carried out only if the root has a left and a right child and they are identical.

With this line we delete the current root:

```
child.removeParent(root);
```

The following code parent links a child node of a node that no longer exists and its parent node. If there is no parent node, the child node becomes the root:

```
if (!root.equals(this.root)) {
    for (DSA_BDD.Node grandparent : root.getParents()) {
        child.addParent(grandparent);

        if (root.equals(grandparent.getLeft())) {
            grandparent.setLeft(child);
        }

        if (root.equals(grandparent.getRight())) {
            grandparent.setRight(child);
        }
    }
} else {
    this.root = child;
}
```

The create function first checks the correctness of the entered data and their correspondence to each other, and then refers to the buildTree_iReduction method, after the completion of the construction it returns bdd:

```
public static DSA_BDD create(String function, String order) {
    if (function.isEmpty() || order.isEmpty()) {
        throw new IllegalArgumentException("Empty format");
    }

    String functionVariables = function.chars()
            .mapToObj(i -> (char) i)
            .filter(c -> c >= 'A' && c <= 'Z')
            .distinct()
            .sorted()
            .map(String::valueOf)
            .collect(Collectors.joining(""));
    String orderVariables = order.chars()
            .mapToObj(i -> (char) i)
            .sorted()
            .map(String::valueOf)
            .collect(Collectors.joining(""));

    if (!functionVariables.equals(orderVariables)) {
        throw new IllegalArgumentException("Mismatch of order
and function");
    }
```

```
    if (!function.matches("[!A-Z+\\s]+")) {
        throw new IllegalArgumentException("The provided format
is incorrect!");
    }

    HashMap<String, Map<String, Node>> map = new HashMap<String,
Map<String, Node>>();

    for (char variable : functionVariables.toCharArray()) {
        map.put(String.valueOf(variable), new HashMap<>());
    }

    DSA_BDD bdd = new DSA_BDD(Node.format(function), order,
map);

    bdd.buildTree_iReduction(bdd.getRoot(), bdd.getOrder());

    return bdd;
}
```

CreateWithTheBestOrder is just iterating over several orders by substituting the first letter of the order at its end and building a tree for each of the orders, as well as comparing the total number of nodes in all the trees built and choosing the smallest of them:

```
do {
    orders.add(temp);

    temp = temp + temp.charAt(0);
    temp = temp.substring( beginIndex: 1);
} while (!temp.equals(variables));

List<DSA_BDD> bdds = orders.stream() Stream<String>
        .map(o -> create(function, o)) Stream<DSA_BDD>
        .sorted(Comparator.comparing(DSA_BDD::computeSize))
        .toList();

return bdds.get(0);
```

The use function traverses the created tree using a combination of 0 and 1. In my implementation, each character from the current order is assigned 0 or 1

from the input. If there is an order symbol in this node, the program moves us to the left or right node, if the symbol is absent, the symbol and digit are skipped:

```
private boolean use(StringBuilder sb, Node root, StringBuilder
order, String input) {

    if (root.equals(trueNode)) {
        return true;
    } else if (root.equals(falseNode)) {
        return false;
    }


    if (order.length() == 0 || order.length() != input.length())
{
        throw new IllegalStateException("The order is empty");
    }

    char currentOrder = order.charAt(0);
    order.deleteCharAt(0);

    if (!input.matches("[01]+")) {
        throw new IllegalArgumentException("Incorrect format");
    }

    char a = input.charAt(0);
    input = input.substring(1);

    if (root.getTitleData().chars().anyMatch(c -> c ==
currentOrder)) {


        if (root.getTitleData().equals(trueNode.getTitleData()))
{
            return true;
        } else if
(root.getTitleData().equals(falseNode.getTitleData())) {
            return false;
        } else if (root.getLeft() == null && root.getRight() ==
null) {
            throw new IllegalStateException("Unexpected value: "
+ root.getData());
        } else {
            boolean res = use(sb, a == '0' ? root.getLeft() :
root.getRight(), order, input);
            sb.append(a);
            return res;
        }
    } else {
        return use(sb, root, order, input);
    }
}
```

We call the Evaluate function in parallel with the use function to check the correctness of the constructed bdd. The Evaluate function does the same as the use function, but instead of traversing the compiled tree, it creates a part of it on its own, also replacing the current order letter with the corresponding input digit. Then it applies the rules of boolean algebra and does the same with the original formula until it reaches 1 or 0. *If the final values of Evaluate and use match, the bdd is built correctly*:

```
protected static boolean Evaluate(Node root, String input,
DSA_BDD bdd) {

    String tInput = input;


    String tData = root.getData();
    String tOrder = bdd.getOrder();

    tData = tData.replaceAll("\\s+", "");

    Pattern pattern = Pattern.compile("![A-Z]");
    java.util.regex.Matcher matcher = pattern.matcher(tData);

    tData = matcher.replaceAll(match ->
match.group().toLowerCase().substring(1));

    while (tOrder.length() > 0 && tInput.length() > 0) {
        char c = tOrder.charAt(0);

        char digit = tInput.charAt(0);

        if (tData.contains(Character.toString(c)) ||
tData.contains(Character.toString(c).toLowerCase())) {

            if (digit == '1') {
                tData = tData.replace(tOrder.charAt(0),
'1').replace(Character.toLowerCase(tOrder.charAt(0)), '0');
            } else {
                tData = tData.replace(tOrder.charAt(0),
'0').replace(Character.toLowerCase(tOrder.charAt(0)), '1');
            }

            List<String> functionVariables = tData.chars()
                    .mapToObj(i -> (char) i)
                    .filter(a -> a >= 'A' && a <= 'Z')
                    .distinct()
                    .sorted()
                    .map(String::valueOf)
                    .toList();

            List<String> DataList =
Arrays.stream(tData.split("\\+")).filter(a ->
```

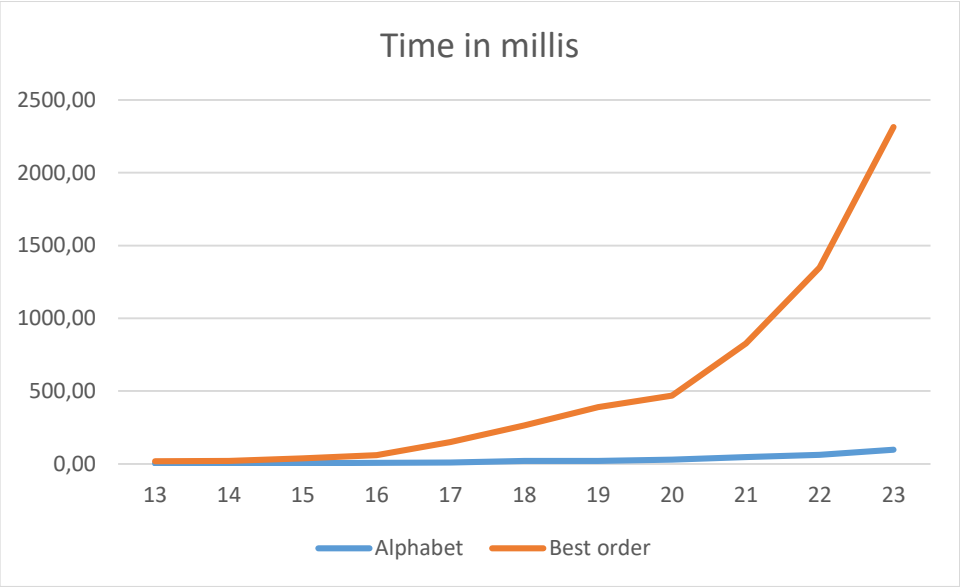
```
!a.contains("0")).toList();

            if (DataList.stream().anyMatch(a ->
a.matches("1+"))) {
                return true;
            }

            String done = DataList.isEmpty() ? "0" :
DataList.stream().distinct()
                    .collect(Collectors.joining("+"))
                    .replace("1", "");

            DataList =
Arrays.stream(done.split("\\+")).toList();

            for (String variable : functionVariables) {
                if (DataList.contains(variable) &&
DataList.contains(variable.toLowerCase())) {
                    return true;
                }
            }

            tData = done;

        }

        tOrder = tOrder.substring(1);
        tInput = tInput.substring(1);

    }

    if(tData == "1")
    {
        return true;
    } else
    {
        return false;
    }

}
```

Test

In the test file, I check the effectiveness of the bdd construct. I measure the amount of memory and the time spent on the execution of operations. Using the DNF function generator, I can choose the number of variables in the expression and the number of clauses (100 clauses were used for all tests).

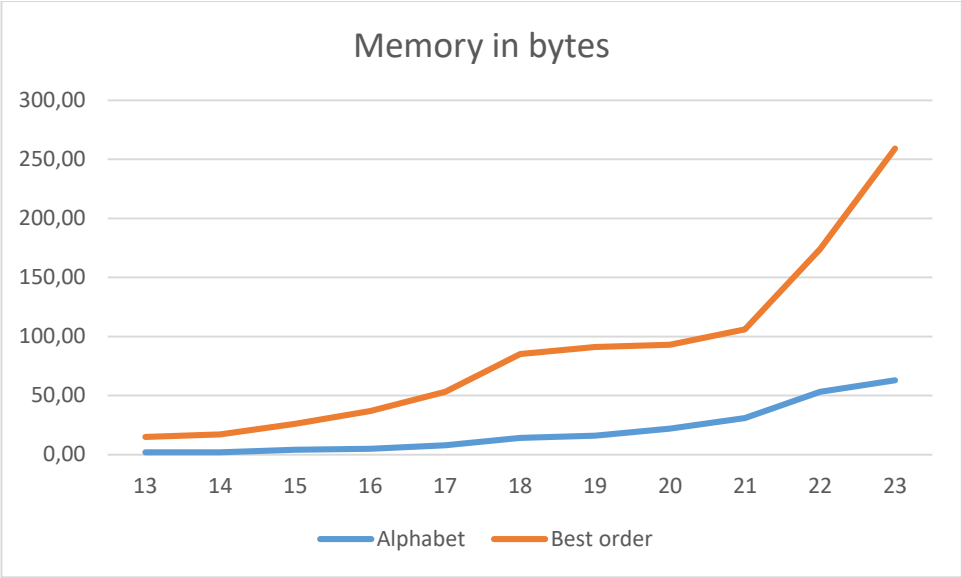
Average Create **Time (in Millis)**

Var	Alphabet	Best order
13	2,12	17,24
14	2,66	20,06
15	4,16	36,96
16	7,96	59,88
17	10,18	149,06
18	19,24	264,64
19	20,66	389,00
20	28,24	468,32
21	45,60	828,02
22	61,14	1348,22
23	97,02	2313,66



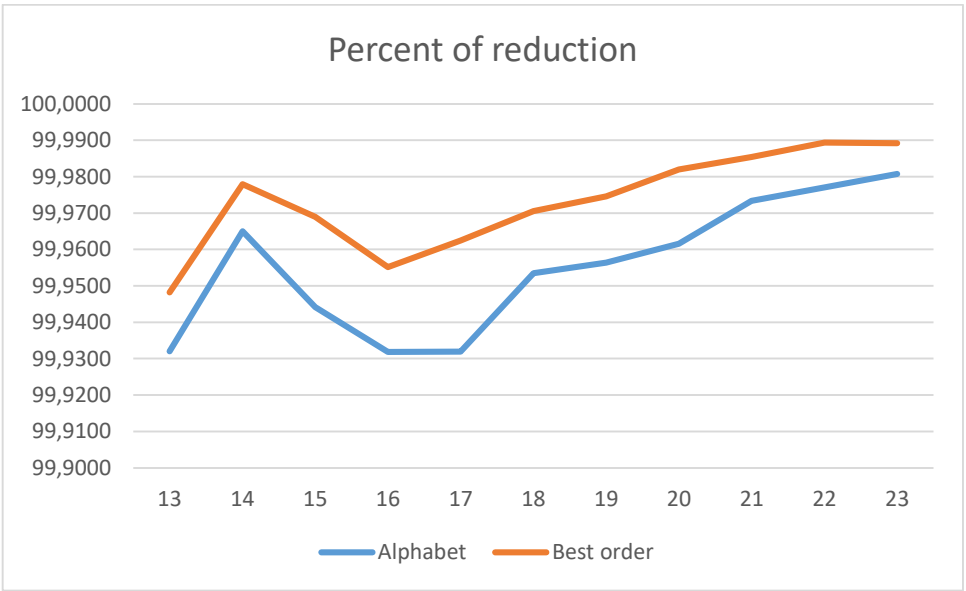
Average Create **Memory (in bytes)**

Var	Alphabet	Best order
13	2,00	15,00
14	2,00	17,00
15	4,00	26,00
16	5,00	37,00
17	8,00	53,00
18	14,00	85,00
19	16,00	91,00
20	22,00	93,00
21	31,00	106,00
22	53,00	174,00
23	63,00	259,00



Average **Reduction Percentage**

Var	Alphabet	Best order
13	99,9320	99,9482
14	99,9650	99,9779
15	99,9441	99,9690
16	99,9318	99,9551
17	99,9319	99,9625
18	99,9535	99,9705
19	99,9564	99,9746
20	99,9616	99,9820
21	99,9733	99,9854
22	99,9770	99,9894
23	99,9807	99,9892



Average **Reduction Percentage Final**

Alphabet	Best order
99,95522107	99,97306715

According to the test results, more memory and time is spent on creating a BDD with the best order, but in most cases the number of nodes in such a BDD will be less.