Oleksandr Oksanich, AIS ID: 122480

Data Structures and Algorithms

Assignment #2 — Binary Decision Diagram

Contents

Introduction....1
Binary Decision Diagram....1
Node reduction....2
Implementation....3
Node structure....3
create....4
createWithBestOrder....7
use....7
Testing....8
Procedure....8
Results....9
Implementation correctness....9
Average reduction percentage....9
Average creation time....10
Average memory usage....11
Total average reduction....12
Conclusion....12

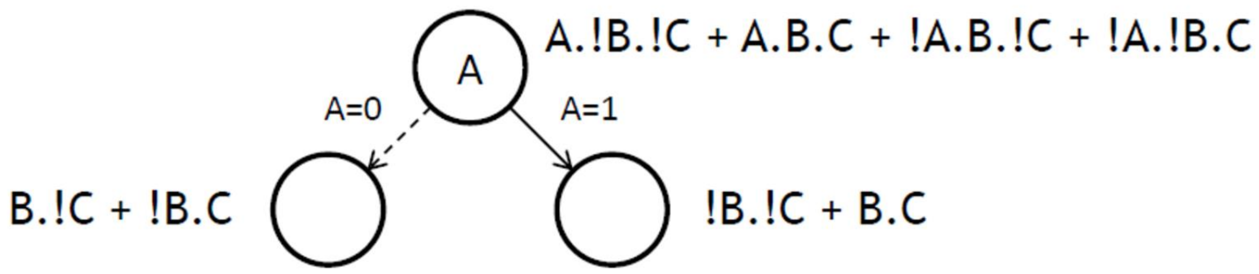
Introduction

The task of the assignment is to develop and thoroughly test an implementation of a reduced ordered binary decision diagram. In order to solve the task, I have chosen Java as the programming language and IntelliJ IDEA as the IDE. Other technical details will be explained in the respective sections of the documentation below.

Binary Decision Diagram

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* (or *decomposition*).

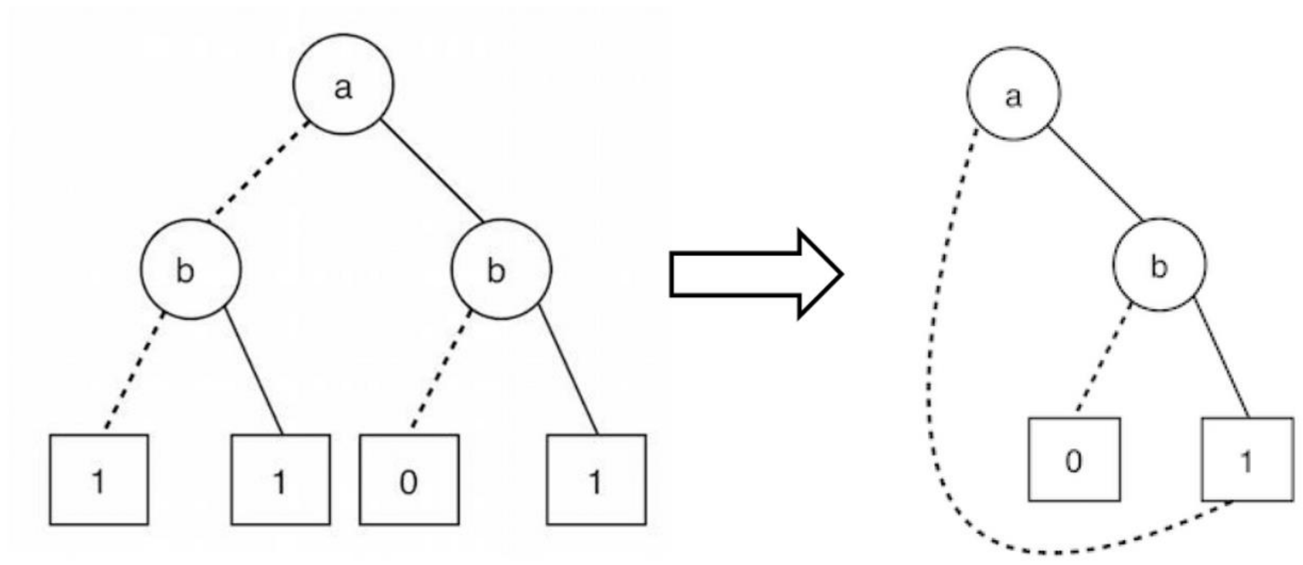
Its basic idea is that each BDD level corresponds to a variable within a Boolean function where left children are given the negated equivalent of the current level's variable whereas right children are given the variable itself:



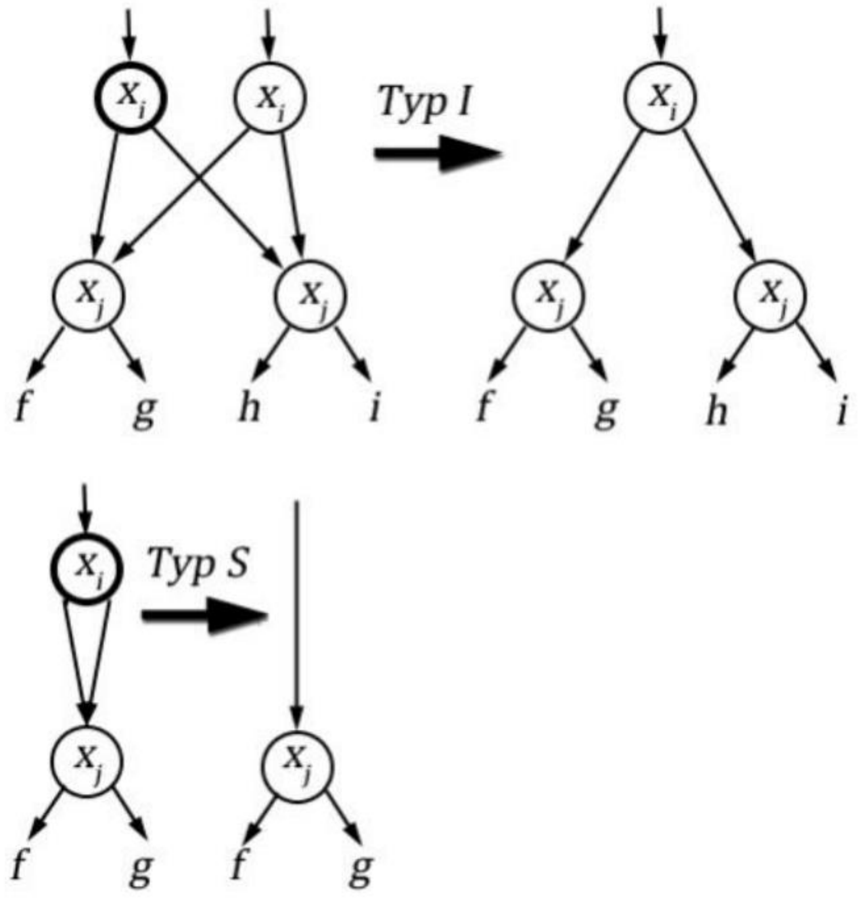
1. Illustration from the lecture on BDDs: the Shannon expansion

Node reduction

One issue with an unreduced BDD is that it would generate a perfect binary tree requiring too many nodes to easily process a Boolean function. In order to improve the efficiency of the data structures, there are a few possibles ways to reduce the node count:



2. Illustration from the lecture on BDDs: linking the ending nodes (as they are always either "0" or "1")



3. Illustration from the lecture on BDDs: I-reduction and S-reduction

Implementation

Node structure

My node implementation is quite straightforward since it only uses references to its function, parents, and children. Although within the `Node` class I have included a number of different static function-parsing methods that are used across the code.

The `format` function is used to remove all the unnecessary spaces from the input and to replace the negated variables provided as “!A” with its lowercase equivalent without the negation operator.

```java
public static String format(String function) {
    function = function.replaceAll("\\s+", "");

    var pattern = Pattern.compile("!([A-Z])");
    var matcher = pattern.matcher(function);

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

    return Arrays.stream(negated.split("\\+"))
            .map(c -> c.chars().mapToObj(i -> (char)
i).sorted().map(String::valueOf).collect(Collectors.joining("")))
            .distinct()
            .sorted()
            .collect(Collectors.joining("+"));
}
```

The following method is used to filter all the formula clauses that contain binary values based on what values are within each clause:

```
public static String parseDigits(String input) {
    var functionVariables = getDnfVariablesOrdered(input, true, true);

    var filtered = Arrays.stream(input.split("\\+")).filter(c ->
!c.contains("0")).toList();

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

    var parsed = filtered.isEmpty() ? "0" : filtered.stream().distinct()
            .collect(Collectors.joining("+"))
            .replace("1", "");

    filtered = Arrays.stream(parsed.split("\\+")).distinct().toList();

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

    return String.join("+", filtered);
}
```

create

My `create` function also appears to be straightforward as its only function is to check correctness of the arguments, fill the I-reduction map with the BDD's "levels", and call the `parse` function that actually creates the BDD:

```
public static BinaryDecisionDiagram create(String function, String order) {
    if (function.isEmpty() || order.isEmpty()) {
        throw new IllegalArgumentException("Neither the function nor the order can
be empty!");
    }

    var functionVariables = String.join("", Node.getDnfVariablesOrdered(function,
true, true));
    var orderVariables = String.join("", Node.getDnfVariablesOrdered(order, false,
false));

    if (!functionVariables.equals(orderVariables)) {
        throw new IllegalArgumentException("The function and order provided do not
correspond to each other!");
    }

    if (!function.matches("[!A-Z+\\s]+")) {
        throw new IllegalArgumentException("The provided format is not correct! DNF
(e.g., ABC + A!B!C) should be used instead!");
    }

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

    for (var variable : functionVariables.toCharArray()) {
        // creating an individual map for each variable
        map.put(String.valueOf(variable), new HashMap<>());
    }

    var bdd = new BinaryDecisionDiagram(function, order, map);

    bdd.parse(bdd.getRoot(), bdd.getOrder());
    bdd.size();

    return bdd;
}
```

The `parse` function is based on recursive node creation where a variable is replaced with either 0 or 1 for each child to be created at each level:

```
var function = root.getFunction();

var left = function.replace(order.charAt(0),
'0').replace(Character.toLowerCase(order.charAt(0)), '1');
var right = function.replace(order.charAt(0),
'1').replace(Character.toLowerCase(order.charAt(0)), '0');

left = Node.parseDigits(left);
right = Node.parseDigits(right);
```

After parsing the functions of the children of the node that's currently processed, those functions are inserted into their nodes depending on whether the functions are already present at this BDD level (I-reduction):

```
var mapLevel = map.get(String.valueOf(order.charAt(0)));

var containsLeft = mapLevel.containsKey(left);
var leftNode = mapLevel.getOrDefault(left, left.equals("0") ? falseLeaf :
left.equals("1") ? trueLeaf : new Node(left));

leftNode.addParent(root);
root.setLeft(leftNode);

mapLevel.put(left, leftNode);
```

```
var containsRight = mapLevel.containsKey(right);
var rightNode = mapLevel.getOrDefault(right, right.equals("0") ? falseLeaf :
right.equals("1") ? trueLeaf : new Node(right));

rightNode.addParent(root);
root.setRight(rightNode);

mapLevel.put(right, rightNode);
```

Then the created nodes are to be parsed in case their function is neither “0” nor “1” (as they are ending nodes which are not to be processed deeper) and it is not already contained in the reduction map (as they would be already processed), but if it contained, then S-reduction is applied to the node to ensure it will be reduced after it has been attached to a new parent. After all, S-reduction is also applied to the node that is currently processed.

```
if (!left.equals("0") && !left.equals("1")) {
    if (!containsLeft) {
        parse(leftNode, order.substring(1));
    } else {
        sReduction(leftNode);
    }
}

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

sReduction(root);
```

As it has already been illustrated, S-reduction is based on checking whether the node’s children refer to the same function, and if this is the case, then one child of the node becomes the child of its parent, replacing the node itself. In case the child does not have a grandparent (i.e., if its parent is the root of the BDD), then it becomes the root itself:

```
private void sReduction(Node root) {
    if (root.getLeft() != null && root.getRight() != null &&
root.getLeft().getFunction().equals(root.getRight().getFunction())) {
        var child = root.getLeft();

        child.removeParent(root);

        if (!root.equals(this.root)) {
            for (var 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;
        }
    }
}
```

createWithBestOrder

My `createWithBestOrder` function uses the linear method of searching the best possible variable order (for example, the orders that will be sequentially checked for a function “ABC + !ABD” are “ABCD”, “BCDA”, “CDAB”, and “DABC”). For each order a BDD is created and the one with the least nodes is returned from the function:

```
var variables = String.join("", Node.getDnfVariablesOrdered(function, true, true));
var temp = variables;

var orders = new HashSet<String>();

// linear method (e.g., ABC, BCA, CAB)
do {
    orders.add(temp);

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

// checking which order generates the least nodes
var bdds = orders.stream()
        .map(o -> create(function, o))
        .sorted(Comparator.comparing(BinaryDecisionDiagram::size))
        .toList();

return bdds.get(0);
```

use

The implementation of the function `use` is based on recursive traversal of the BDD until an ending node (i.e., the one that has no children) is found to return. Before that, which child (left or right) will be processed next depends on whether the value of the currently processed variable (following the order from the input) is set to 0 or 1. Nevertheless, there is a special case when the level of the variable is reduced within the BDD so the node gets processed again with the next variable instead of traversing its child:

```
private boolean use(String input, Node root) {
    if (root.getLeft() == null && root.getRight() == null) {
        // just in case a leaf node contains something else than "0" or "1"
        // should never occur
        if (!root.getFunction().equals(falseLeaf.getFunction()) &&
!root.getFunction().equals(trueLeaf.getFunction())) {
            throw new IllegalStateException("Unexpected value: " +
root.getFunction());
        }

        return root.getFunction().equals(trueLeaf.getFunction());
    } else {
        var data = root.getFunctionDnf();

        // checking whether the current variable has been reduced
        if (!data.contains(String.valueOf(order.charAt(order.length() -
input.length())))) {
            return use(input.substring(1), root);
        }

        return use(input.substring(1), input.charAt(0) == '0' ? root.getLeft() :
root.getRight());
    }
}
```

Testing

Procedure

In order to test my assignment solution, I implemented a testing program which is able to generate a random DNF function, find out its Boolean value for each input combination, and compare it to the corresponding values from the BDD.

The `generateDnfExpression` method generates a function with the specified count of variables that may contain from 15 to 60 DNF clauses:

```
public static String generateDnfExpression(int variableCount) {
    var random = new Random();

    var clausesCount = random.ints(1, MIN_CLAUSES, MAX_CLAUSES + 1)
            .findFirst()
            .orElseThrow();

    var clauses = new ArrayList<String>();

    var letters = new ArrayList<>(CAPITAL_LETTERS.chars().mapToObj(i -> (char)
i).toList());

    Collections.shuffle(letters);

    var variables = letters.stream().limit(variableCount).toList();

    for (int i = 0; i < clausesCount; i++) {
        var variablesToTake = random.ints(1, 1, variableCount + 1)
```

```
                .findFirst()
                .orElseThrow();

        var temp = new ArrayList<>(variables);

        Collections.shuffle(temp);

        // 15% is the probability of whether a variable within a clause will be
negated
        // but only if a clause doesn't already contain a non-inverted variable to
prevent a contradiction
        clauses.add(letters.stream()
                .limit(variablesToTake)
                .map(v -> (Math.random() <= 0.15 &&
!clauses.contains(String.valueOf(v)) ? "!" : "") + v)
                .collect(Collectors.joining("")));
    }

    var result = String.join(" + ", clauses.stream().distinct().toList());
    var resultVariableCount = result.chars()
            .mapToObj(i -> (char) i)
            .distinct()
            .filter(c -> c >= 'A' && c <= 'Z')
            .count();

    // in case the result contains fewer variables than expected
    return resultVariableCount < variableCount ?
generateDnfExpression(variableCount) : result;
}
```

Via the `testBdd` function, 100 tests were conducted for each count of variables given (from 13 through 20) to find out average memory usage, average creation time, and, of course, whether every single BDD value is correct. Moreover, I have conducted all the mentioned tests twice in order to test both alphabetical order creation and the `createWithBestOrder` function.

Results

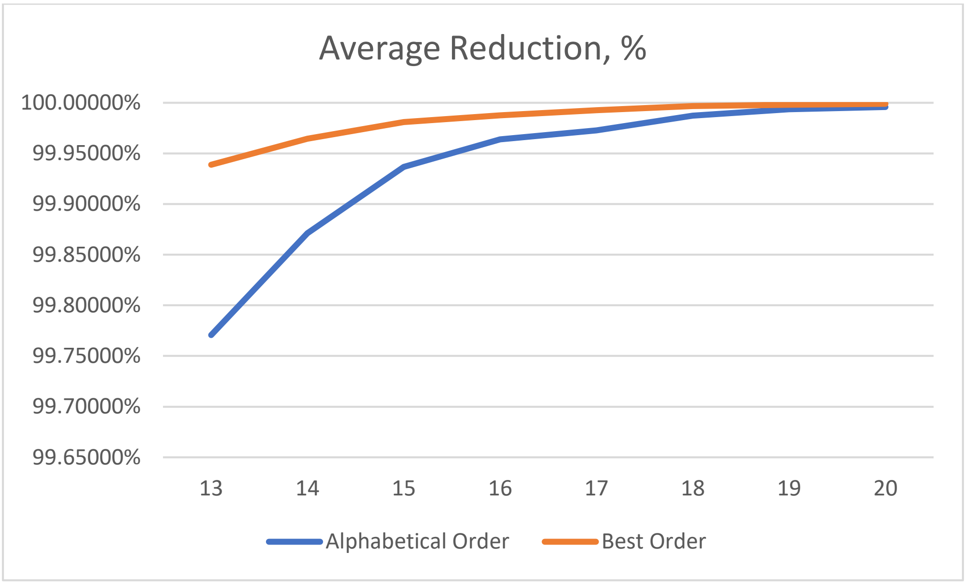
Implementation correctness

Based on the raw testing results described in the files named “alphabetical_order_testing_results.txt” and “best_order_testing_results.txt” in the project directory, all the tests have been successfully passed as well as a number of special cases that are listed in the main source code file.

Average reduction percentage

Variable Count	**Alphabetical Order**	**Best Order**
13	99.77068%	99.93884%
14	99.87124%	99.96466%
15	99.93681%	99.98084%
16	99.96399%	99.98764%
17	99.97276%	99.99269%

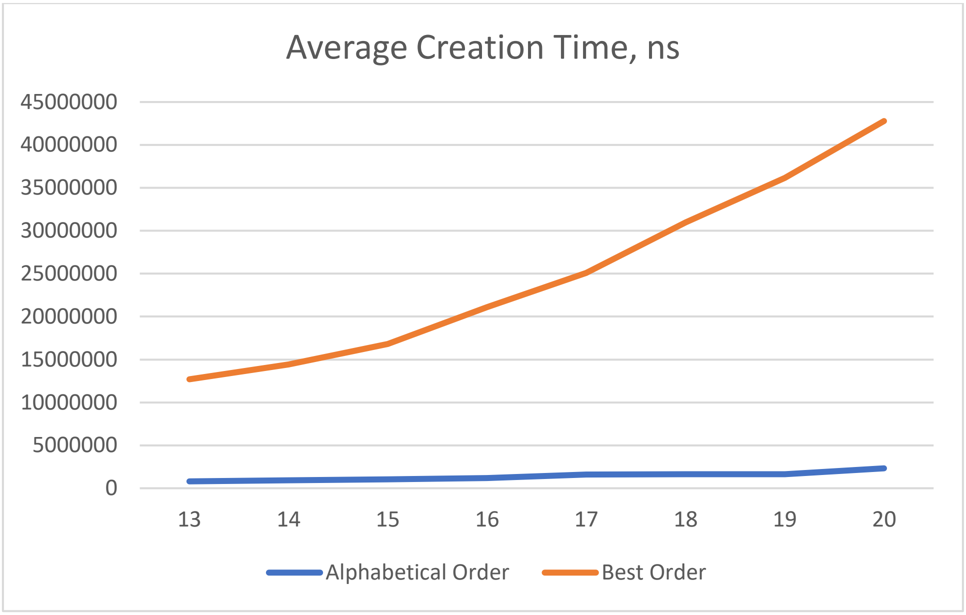
18	99.98745%	99.99675%
19	99.99363%	99.99794%
20	99.99583%	99.99898%



Average creation time

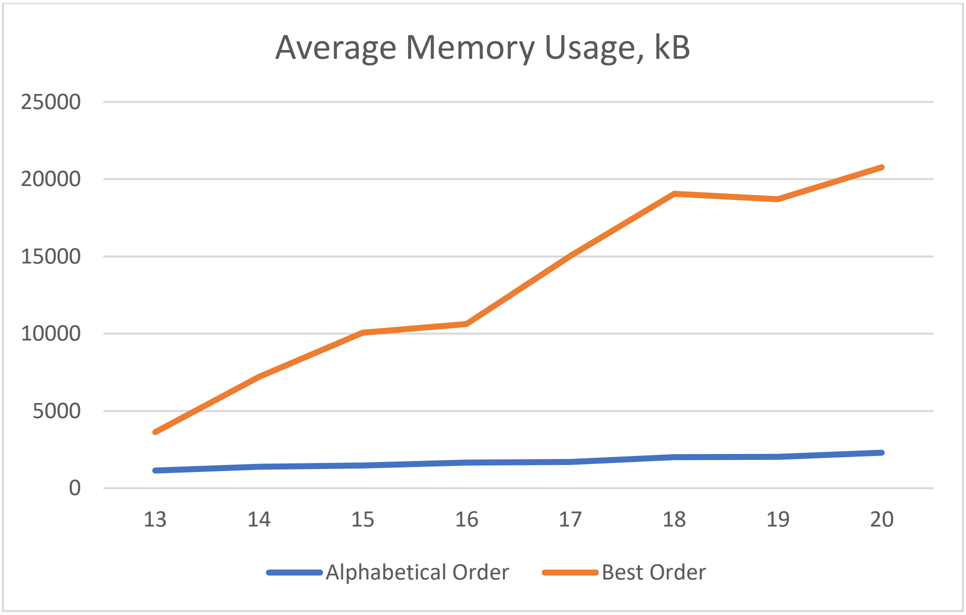
Variable Count	**Alphabetical Order**	**Best Order**
13	801451	12695539
14	910211	14404769
15	1034636	16822909
16	1192936	21109842
17	1592256	25069221
18	1623175	30960316
19	1647853	36143501
20	2320993	42783836

Oleksandr Oksanich, AIS ID: 122480



Average memory usage

Variable Count	**Alphabetical Order**	**Best Order**
13	1143	3627
14	1385	7205
15	1469	10063
16	1662	10621
17	1702	15036
18	2015	19054
19	2021	18697
20	2297	20771



Oleksandr Oksanich, AIS ID: 122480

Total average reduction

Alphabetical Order	Best Order
99.93655%	99.98229%

Conclusion

The testing results demonstrate that searching for the best possible variable order obviously require more time and memory, but one advantage lies in the fact that the average node reduction appears to be better in most cases, although the difference gets less obvious as the count of variables increases.