Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Data structures and algorithms

Assignment 2 – Binary Decision Diagrams

Petra Miková

Petra Miková
ID: 120852

Contents

Petra Miková
ID: 120852

Aims and objectives

The aim of this assignment was to create a program, where a data structure called BDD (Binary Decision Diagram) with a focus for representation of Boolean functions can be created. There were three functions that were compulsory to create:

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

A Binary Decision Diagram (BDD) is a graph-based data structure used to represent and manipulate Boolean functions efficiently. The nodes of the graph correspond to variable assignments, and edges correspond to variable values (true or false). BDDs can represent functions of a large number of variables compactly and can perform operations such as conjunction, disjunction, and negation quickly.

In my case, I decided to implement a BDD that accepts a Boolean function in form of sum of products (e.g. AB+CD+A!E). For conjunction, a + is used, for conjunction there is no sign used, just the variables put next to each other, and for negation ! is used. The correctness of created BDD is then tested with the outputs for the function that are evaluated by inserting the logical operators Java offers and getting a boolean result, which is then compared to result BDD has computed. The code also includes many helper methods, which will be described in this documentation. The whole assignment is coded in Java language and I have used the IntelliJ IDE.

Regarding the testing, I implemented 4 test scenarios – first one takes in certain input from user and tests the function, another one takes an amount of variables and tests one random function, another again takes in the number of variables but tests 100 random functions, and then one time complexity test.

Petra Miková
ID: 120852

The BDD and BDDNode classes

The BDD class in my implementation represents a BDD and provides methods for building, manipulating, and evaluating the BDD. The BDDNode class is used to represent a node in the BDD graph and provides a method for hashing nodes.

The classes for BDD and BDD tree node with their attributes:

```
public class BDD {
    public BDD() {}
    private BDDNode root;
    private String bfunction;
    private int numOfVariables;
    private String orderOfVariables;
    private int size;

    public BDDNode getRoot() {
        return root;
    }
    public void setRoot(BDDNode root) {
        this.root = root;
    }
    public int getSize() {
        return size;
    }

    static class BDDNode {
        public BDDNode parent;
        List<String> terms;
        private int varIndex;
        private BDDNode lowChild;
        private BDDNode highChild;
        private boolean isTerminal;
        private Boolean value;

        public boolean isTerminal() {
            return isTerminal;
        }
        public int getVarIndex() {
            return varIndex;
        }
        public BDDNode getLowChild() {
            return lowChild;
        }
        public BDDNode getHighChild() {
            return highChild;
        }
        public Boolean getValue() {
            return value;
        }

        public BDDNode() {
        }

        public BDDNode(boolean isTerminal, Boolean value) {
            this.isTerminal = isTerminal;
            this.value = value;
        }
    }
```

Petra Miková
ID: 120852

The class BDD has 5 attributes – pointer to the root, its Boolean function, the number and order of variables of the BDD, and its size. The class BDDNode has 7 attributes – pointer to its parent, list of terms at that node, variable index which is the ASCII representation of the variable character the node represents, pointers to the lowchild (when the value of the variable is false) and highchild (when the value of the variable is true), an attribute whether the node is terminal, and the value of the variable the node represents (true, false for terminal nodes and null for other in the process of creation). For both classes there are essential constructors, getters, and setters for my implementation as well.

Implementation of BDD_create

Description

To implement a BDD_create function which returns a created BDD in a binary tree form, I first break down the function to obtain all of the variables from function, and then parse the order argument to get a specific order of variables for the creation of that BDD. Then I break down the whole Boolean function at + sign and obtain a list of terms, which I then modify that the negated variables are substituted with the lowercase version of that variable just for simpler manipulation with the terms later (e.g for !AB!C the output term is aBc). After that, true and false leaves are created, as one reduction of the BDD is to use just two 0 and 1 terminal nodes and set pointers to them instead of creating multiple identical terminal nodes for each pre-terminal node. Then I set the attributes of BDD and create pointers and structures essential for BDD creation and call a helper method which creates the BDD and returns a pointer to the root of it. Then this pointer is set to the current BDD root pointer, we obtain the size of BDD and return the BDD.

BDD_create

How this function works has been briefly explained already, and this is how it looks:

```java
public static BDD BDD_create(String bfunction, String order) {
    Set<Character> variables;
    variables = retrieveVariablesFromBfunction(bfunction);

    List<Character> orderVariableList = new ArrayList<>();
    for (char c : order.toCharArray()) {
        if (variables.contains(Character.toUpperCase(c))) {
            orderVariableList.add(Character.toUpperCase(c));
        }
    }

    String[] termsarray = bfunction.split("\\+");
    List<String> terms = Arrays.asList(termsarray);

    List<String> modifiedTerms = new ArrayList<>();
    for (String term : terms) {
        StringBuilder sb = new StringBuilder(term);
        for (int i = 0; i < sb.length(); i++) {
            if (sb.charAt(i) == '!') {
                sb.deleteCharAt(i);
                sb.insert(i, Character.toLowerCase(sb.charAt(i)));
                sb.deleteCharAt(i + 1);
            }
        }
        modifiedTerms.add(sb.toString());
    }

    BDDNode trueLeaf = new BDDNode(true, true);
    BDDNode falseLeaf = new BDDNode(true, false);
    BDD bdd = new BDD();
    BDDNode root = null;
    BDDNode currentNode = null;
    bdd.orderOfVariables = order;
    bdd.numOfVariables = variables.size();
    bdd.bfunction = bfunction;
    HashMap<String, BDDNode> uniqueTable = new HashMap<String, BDDNode>();
    root = createBDD_helper(modifiedTerms, orderVariableList, root,
currentNode, trueLeaf, falseLeaf, uniqueTable);
    bdd.setRoot(root);
    Set<String> visited = new HashSet<>();
    int uniqueNodes = countUniqueNodes(root, visited);
    bdd.size = uniqueNodes + 2;
    return bdd;
}
```

The boolean function is first parsed to obtain the set of variables involved in the function. The variable order is then parsed to obtain a list of variables in the order specified by the user. The function string is split into a list of terms and each term is modified to remove negations and ensure uniform capitalization. Then the BDD data structure is then initialized with true and false leaf nodes, an empty root node, and a set of unique nodes. The function createBDD_helper recursively constructs the BDD using the list of modified terms, variable order list, and unique node set. The BDD size is calculated by counting the number of unique nodes in the BDD and also counting in the two terminal true and false leaves. Finally, the BDD is returned.

createBDD_helper

This function takes care of whole BDD creation and returns the pointer to the root of it. The function is pretty long, so I will break it down into sections in this documentation and explain all the parts of it.

The part where the last variable od BDD is handled and terminal nodes are assigned:

```
if (orderedV.size() == 1) {
    char variable = orderedV.get(0);
    Character lowerV = Character.toLowerCase(variable);
    CharSequence charSeq = new String(new char[]{variable});
    CharSequence charSeqL = new String(new char[]{lowerV});

    String key = "terms:" + concatTerms(Terms) + ":" + trueLeaf + ":" +
falseLeaf;

    BDDNode existingNode = uniqueTable.get(key);
    if (existingNode != null) {
        return existingNode;
    }

    BDDNode variableNode = new BDDNode();
    variableNode.varIndex = variable;
    variableNode.terms = Terms;


    if (Terms.isEmpty()) {
        variableNode.lowChild = falseLeaf;
        variableNode.highChild = falseLeaf;
    } else if (Terms.size() == 1) {
        for (String term : Terms) {
            if (term.contains(charSeq)) {
                variableNode.lowChild = falseLeaf;
                variableNode.highChild = trueLeaf;
            } else if (term.contains(charSeqL)) {
                variableNode.lowChild = trueLeaf;
                variableNode.highChild = falseLeaf;
            } else if (term.contains("1")) {
                variableNode.lowChild = trueLeaf;
                variableNode.highChild = trueLeaf;
            } else if (term.contains("0")) {
                variableNode.lowChild = falseLeaf;
                variableNode.highChild = falseLeaf;
            }
        }
    } else {
        for (int i = 0; i < Terms.size(); i++) {
            if (i == (Terms.size() - 1)) {
                break;
            }
            if (Terms.get(i).contains(charSeq)) {
                if (Terms.get(i + 1).contains(charSeqL)) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = trueLeaf;
                } else if (Terms.get(i + 1).contains("1")) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = trueLeaf;
                } else if (Terms.get(i + 1).contains("0")) {
                    variableNode.lowChild = falseLeaf;
                    variableNode.highChild = trueLeaf;
                }
            } else if (Terms.get(i).contains(charSeqL)) {
                if (Terms.get(i + 1).contains(charSeq)) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = trueLeaf;
                } else if (Terms.get(i + 1).contains("1")) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = trueLeaf;
                } else if (Terms.get(i + 1).contains("0")) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = falseLeaf;
                }
            } else if (Terms.get(i).contains("1")) {
                if (Terms.get(i + 1).contains(charSeq)) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = trueLeaf;
                } else if (Terms.get(i + 1).contains(charSeqL)) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = trueLeaf;
                }
            } else if (Terms.get(i).contains("0")) {
                if (Terms.get(i + 1).contains(charSeq)) {
                    variableNode.lowChild = falseLeaf;
                    variableNode.highChild = trueLeaf;
                } else if (Terms.get(i + 1).contains(charSeqL)) {
                    variableNode.lowChild = trueLeaf;
                    variableNode.highChild = falseLeaf;
                }
            }
        }
    }

    if (parentNode == null) {
        rootNode = variableNode;
    } else if (parentNode.highChild == null) {
        parentNode.highChild = variableNode;
    } else {
        parentNode.lowChild = variableNode;
    }

    uniqueTable.put(key, variableNode);
    return rootNode;
}
```

This section of the code which is at the beginning of function specifically handles the case where there is only one variable left to process, meaning that we have reached the end of the variable order. First, the variable and its lowercase equivalent are extracted from the list of ordered variables. Then, a key is constructed using the ordered list of terms, the true and false leaves (which represent the terminal nodes of the tree), and the variable. This key is used to check if a BDD node with the same key already exists in the unique table. If it does, that existing node is returned (reduction type I). If there is no existing node with the same key, a new BDD node is created and initialized with the appropriate variable index and list of terms.

Then, based on the number of terms, one of three scenarios occurs:

1. If there are no terms, the node's low and high child pointers are both set to the false leaf node.
2. If there is only one term, the node's low and high child pointers are set based on the presence of the variable in that term. If the variable is present in the term, the high child pointer is set to the true leaf node and the low child pointer is set to the false leaf node. If the variable's lowercase equivalent is present in the term, the high and low child pointers are reversed. If neither the variable nor its lowercase equivalent is present in the term, the low and high child pointers are both set to the appropriate terminal node based on the presence of 0s or 1s in the term.
3. If there are multiple terms, the node's low and high child pointers are set based on the ordering of the terms in the list. The first term is compared to the variable, and the second term is then compared to the next variable in the ordered list. Depending on the presence or absence of variables and 0s/1s in the two terms, the low and high child pointers are set accordingly. This process continues until all terms have been processed.

Finally, the newly created node is added to the tree as either the root node (if there is no parent node), the high child node of the parent node (if the parent node's high child pointer is null), or the low child node of the parent node (if the parent node's low child pointer is null). The new node is also added to the unique table using the previously constructed key. The root node of the tree is then returned.

If the variable is not the last one in the order, then we hop on to the second part of the function:

```
Character variable = orderedV.get(0);
List<String> trueTerms = new ArrayList<>();
List<String> falseTerms = new ArrayList<>();
for (String term : Terms) {
    if (term.indexOf(variable) == -1) {
        if (term.indexOf(Character.toLowerCase(variable)) != -1) {
            String newTerm;
            newTerm = term.replace(Character.toLowerCase(variable), ' ');
            String cleanedTerm = "";
```

```java
            for (int i = 0; i < newTerm.length(); i++) {
                char c = newTerm.charAt(i);
                if (Character.isLetter(c)) {
                    cleanedTerm += c;
                }
            }
            if (cleanedTerm.isEmpty()) {
                cleanedTerm = "1";
            }
            falseTerms.add(Associativity(Indempotent(cleanedTerm)));

        } else {
            String newTerm;
            newTerm = term.replace(variable, ' ');
            String cleanedTerm = "";
            for (int i = 0; i < newTerm.length(); i++) {
                char c = newTerm.charAt(i);
                if (Character.isLetter(c)) {
                    cleanedTerm += c;
                }
            }
            if (cleanedTerm.isEmpty()) {
                cleanedTerm = "1";
            }
            trueTerms.add(Associativity(Indempotent(cleanedTerm)));
            falseTerms.add(Associativity(Indempotent(cleanedTerm)));
        }
    } else {
        String newTerm;
        newTerm = term.replace(variable, ' ');
        String cleanedTerm = "";
        for (int i = 0; i < newTerm.length(); i++) {
            char c = newTerm.charAt(i);
            if (Character.isLetter(c)) {
                cleanedTerm += c;
            }
        }
        if (cleanedTerm.isEmpty()) {
            cleanedTerm = "1";
        }
        trueTerms.add(Associativity(Indempotent(cleanedTerm)));

    }
}

trueTerms =
Absorption(InverseLaw(Annulment(Identity(removeDuplicates(trueTerms)))));

falseTerms =
Absorption(InverseLaw(Annulment(Identity(removeDuplicates(falseTerms)))));
```

First, we retrieve the current variable and initialize two lists for true and false terms regarding this variable. We then traverse all the current terms and for each one we check where to put it.

1. Case where variable is not in the term

If the variable is not in the term, but its lower case version is present, it means this variable is negated in the term. In that case we create a "cleaned term" without this variable as per the Shannon decomposition. This term cleaning process is performed in every case.

If there is not even the negation of the variable present in the term, it means the value of the variable does not matter for that term and we put the term to both false and true terms.

2. Case where variable is in the term
   If the variable is present in the term, it is simply put into the true terms list.

In each case, if the term after cleaning it off of the current variable is empty, the term is set to 1. Moreover, before adding each term into a list, it is first checked whether the Asociativity and Indempotent laws from Boolean algebra could be applied on it.

Eventually for this part, for both the new true and false terms list, again some of the Boolean algebra laws are applied to the full list, which will be explained in the section Helper methods for BDD_create.

The last part of this function is an actual creation of the BDD:

```
    BDDNode trueNode;
    BDDNode falseNode;

    trueNode = createBDD_helper(trueTerms, orderedV.subList(1,
orderedV.size()),orderedVstatic, rootNode, parentNode, trueLeaf, falseLeaf,
uniqueTable);
    falseNode = createBDD_helper(falseTerms, orderedV.subList(1,
orderedV.size()), orderedVstatic, rootNode, parentNode, trueLeaf,
falseLeaf, uniqueTable);

    String key = "terms:" + concatTerms(Terms) + ":" + trueNode.toString()
+ ":" + falseNode.toString();

    BDDNode existingNode = uniqueTable.get(key);

    if (existingNode != null) {
        return existingNode;
    } else {
        BDDNode variableNode = new BDDNode();
        variableNode.varIndex = variable;
        variableNode.highChild = trueNode;
        trueNode.parent = variableNode;
        variableNode.lowChild = falseNode;
        falseNode.parent = variableNode;
        variableNode.terms = Terms;
        uniqueTable.put(key, variableNode);
        reduction_typeS(variableNode, uniqueTable);


        if (parentNode == null) {
            rootNode = variableNode;
        } else if (parentNode.highChild == null) {
            parentNode.highChild = variableNode;
        } else {
            parentNode.lowChild = variableNode;

        }
        if (rootNode.varIndex == orderedVstatic.get(0) &&
rootNode.lowChild.lowChild != null && rootNode.lowChild.highChild != null
&&  rootNode.highChild.lowChild != null && rootNode.highChild.highChild !=
null){
        String rootLowchildkey = "terms:" +
concatTerms(rootNode.lowChild.terms) + ":" +
rootNode.lowChild.lowChild.toString() + ":" +
rootNode.lowChild.highChild.toString();
        String rootHighchildkey = "terms:" +
concatTerms(rootNode.highChild.terms) + ":" +
rootNode.highChild.lowChild.toString() + ":" +
rootNode.highChild.highChild.toString();
            if (rootLowchildkey.equals(rootHighchildkey)) {
                if (rootNode.lowChild.parent != null ) {
                    rootNode.lowChild.parent = null;
                    rootNode.lowChild.terms.addAll(rootNode.terms);
                    rootNode = rootNode.lowChild;
                    return rootNode;
                }
            }
        }

        return rootNode;
    }
}
```

In this part I recursively construct the tree by calling the function again for the true and false child of the current node, with updating the parameters. Then the key for the current node is computed and it is checked whether the node like that already exists, so the reduction type I is performed within the creation of the BDD. If the node is unique, then a new node is created and all its attributes are set. It is put into the unique table so that reductions can keep performing and also the node is checked whether a type S reduction can be performed in a function that will be described later. Lastly it is put into the BDD tree structure and there is a check for reduction type S of the root node, so the node with the variable that is the first in the order, because as I am creating the structure recursively I am checking the children of the current node in the reduction type S function and thus the root node would not get reduced there, so that's why I perform it here.

Helper methods for BDD_create

1. retrieveVariablesFromBfunction

This method is simply taking the Boolean function as a parameter and while traversing it retrieves the variables from it by checking whether the current character is a letter.

```
public static Set<Character> retrieveVariablesFromBfunction(String
bfunction){
    Set<Character> variables = new HashSet<>();
    for (int i = 0; i < bfunction.length(); i++) {
        char c = bfunction.charAt(i);
        if (Character.isLetter(c)) {
            variables.add(c);
        }
    }
    return variables;
}
```

2. countUniqueNodes

   Aim of this helper method is to count the unique nodes in the fully created BDD also in a way by putting the hashed nodes into a hash set structure. It traverses the tree and then returns 0 for terminal nodes as they are being counted in in the BDD_create function itself.

```
public static int countUniqueNodes(BDDNode root, Set<String> visited) {
    if (root == null) {
        return 0;
    }

    if (root.isTerminal) {
        return 0;
    }

    String hash = "terms:" + concatTerms(root.terms) + ":" + root.highChild
+ ":" + root.lowChild;

    if (visited.contains(hash)) {
        return 0;
    }
    visited.add(hash);
    int count = 1;
    count += countUniqueNodes(root.getLowChild(), visited);
    count += countUniqueNodes(root.getHighChild(), visited);
    return count;
}
```

3. concatTerms

   Aim of this function is to simply put together all the terms of the current node into one string so that this string can be used for hashing a node. It also puts a | between them to avoid incorrect behavior like hashing the node with terms D+e and node with term De as a same node.

```
private static String concatTerms(List<String> terms) {
    if (terms != null){
    Collections.sort(terms);
    return String.join("|", terms);}
    return null;
}
```

Reduction of the BDD

The BDD is reduced throughout the creation process. The reductions performed are:

1. **type I reduction**
   This reduction ensures that each node exists only once and if it occurs multiple times in the BDD, simply just pointer is set to this one node instead of creating a new one every time it is needed. There is no separate function for it, but it happens directly in the createBDD_helper method when a new node should be inserted – a key for the node is created based on its terms and children, and we try to retrieve this node from the unique nodes table. If we get null, so the does not exist yet, we create it and put into this table as well. However, if it does exist, we just return this equal existing node.

2. **type S reduction**
   This reduction ensures that if there is a node whose low and high child point to the exact same node, we omit this node and set the parent of the node to which those children

pointed to the parent of ommited node as this ommited node is irrelevant in the evaluation later. In my implementation I perform it on every node that was added as a new unique node in a way that i access its high and low child and then access their children and perform reduction if it should happen. Because of that, this reduction on the root node happend right in the createBDD_helper method as this function would never get to reduce the root itself.

```
private static void reduction_typeS(BDDNode node, HashMap<String, BDDNode>
uniqueTable) {
    if (node.lowChild.lowChild == node.lowChild.highChild &&
node.lowChild.lowChild != null && node.lowChild.highChild != null) {
        if (node.lowChild.parent != null) {
            String key = "terms:" + concatTerms(node.lowChild.terms) + ":"
+ node.lowChild.highChild.toString() + ":" +
node.lowChild.lowChild.toString();
            node.lowChild.lowChild.parent = node.lowChild.parent;
            node.lowChild = node.lowChild.lowChild;
            uniqueTable.remove(key);
        }
    }
    if (node.highChild.lowChild == node.highChild.highChild &&
node.highChild.lowChild != null && node.highChild.highChild != null) {
        if (node.highChild.parent != null || node.highChild.isTerminal) {
            String key = "terms:" + concatTerms(node.highChild.terms) + ":"
+ node.highChild.highChild.toString() + ":" +
node.highChild.lowChild.toString();
            node.highChild.highChild.parent = node.highChild.parent;
            node.highChild = node.highChild.highChild;
            uniqueTable.remove(key);
        }
    }
}
```

The function checks if the low child and high child of the input node have the same children. If so, it removes the input node and updates the parent of the child node to point to the grandparent of the removed node. It also removes the removed node from the unique nodes hashmap. The same process is repeated for the high child of the input node.

3. **Boolean algebra laws**
Boolean algebra provides a set of rules for manipulating with boolean expressions. By applying these rules, we can simplify the expression and potentially reduce the size of the BDD. In my implementation, I have managed to implement some of the most powerful Boolean algebra laws.

   i. removeDuplicates
   This function ensures that within the terms, all the duplicates of each term are removed due to the rule that A+A+...+A is still just A.

```
public static List<String> removeDuplicates(List<String> terms) {
    Set<String> uniqueTerms = new HashSet<>();
    List<String> nonDuplicateTerms = new ArrayList<>();
```

```java
    for (String term : terms) {
        if (!uniqueTerms.contains(term)) {
            uniqueTerms.add(term);
            nonDuplicateTerms.add(term);
        }
    }
    return nonDuplicateTerms;
}
```

ii. Indempotent
This method ensures that in each term, there are no duplicates of the variables present, due to the law that A.A.A....A = A.

```java
public static String Indempotent(String term){
    String nonDuplicateTerm = "";
    Set<Character> uniqueChars = new HashSet<>();
    for (int i = 0; i < term.length(); i++) {
        char c = term.charAt(i);
        if (!uniqueChars.contains(c)) {
            uniqueChars.add(c);
            nonDuplicateTerm += c;
        }
    }
    return nonDuplicateTerm;
}
```

iii. Associativity
This method takes a term as a string input and returns the same term but with its variables sorted alphabetically. This is because of the associative law and is very efficient for the reduction type I, as the AB and BA terms are the same but if we did not sort the BA in the beginning before creating the node, these would be assigned different key and be evaluated as two unique nodes.

```java
public static String Associativity(String term){
    char[] charArray = term.toCharArray();
    Arrays.sort(charArray);
    String sortedStr = String.valueOf(charArray);
    return sortedStr;
}
```

iv. Absorption
This method takes in a list of terms as input and creates a copy of that list called result. It then iterates through the terms in the list and compares each term to every other term that comes after it in the list. If one term is a proper subset of the other (i.e. the shorter term is entirely contained within the longer term), then the longer term is removed from the result list as the absorption law declares that A+AB = A. The new terms list is then returned.

```java
public static List<String> Absorption(List<String> terms) {
    List<String> result = new ArrayList<>(terms);
    for (int i = 0; i < terms.size(); i++) {
        String term1 = terms.get(i);
        for (int j = i + 1; j < terms.size(); j++) {
            String term2 = terms.get(j);
            if (term1.length() < term2.length() && term2.startsWith(term1))
{
                result.remove(term2);
            } else if (term2.length() < term1.length() &&
term1.startsWith(term2)) {
                result.remove(term1);
                break;
            }
        }
    }
    return result;
}
```

v. Annulment
This method performs the boolean law of annulment. If the list of terms contains the string 1, then the function returns a new list that only contains 1 since any expression OR-ed with 1 evaluates to 1. Otherwise, it returns a new list with the same terms as the original input list.

```
public static List<String> Annulment(List<String> terms) {
    List<String> simplifiedTerms = new ArrayList<>();
    if (terms.contains("1")) {
        simplifiedTerms.add("1");
    } else {
        simplifiedTerms.addAll(terms);
    }
    return simplifiedTerms;
}
```

vi. Identity
This method looks for a 0 term in the terms list and if it is there, it removes it and returns a list of terms without is as according to the Identity law A+0 = A.

```
public static List<String> Identity(List<String> terms) {
    if (terms.contains("0")) {
        terms.remove("0");
    }
    return terms;
}
```

vii. Inverse law
The method performs the law which says that a variable OR-ed with its negation is equal to 1, so A+!A = 1. It traverses the whole terms list and compares all variables to find all such cases. It then returns a new result list with these variables being replaced with single 1.

```
public static List<String> InverseLaw(List<String> terms) {
    List<String> result = new ArrayList<>();
    for (int i = 0; i < terms.size(); i++) {
        String term = terms.get(i);
        boolean inverseFound = false;
        for (int j = i + 1; j < terms.size(); j++) {
            String otherTerm = terms.get(j);
            if (otherTerm.length() == term.length() &&
                    areInverseTerms(term, otherTerm)) {
                inverseFound = true;
                terms.remove(term);
                terms.remove(otherTerm);
                result.add("1");
                break;
            }
        }
        if (!inverseFound) {
            result.add(term);
        }
    }
    return result;
}

private static boolean areInverseTerms(String term1, String term2) {
    if (term1.length() != 1 || term2.length() != 1) {
        return false;
    }
    char c1 = term1.charAt(0);
    char c2 = term2.charAt(0);
    if (Character.isLowerCase(c1)) {
        if (Character.toUpperCase(c1) == c2) {
            return true;
        }
    } else if (Character.isUpperCase(c1)) {
        if (Character.toLowerCase(c1) == c2) {
            return true;
        }
    }
    return false;
}
```

Implementation of BDD_create_with_best_order

Description

The aim of this function is to try out multiple variables of the Boolean function order and return the BDD which is the most efficient – so has the least nodes from all the orders that were tried out. The best way to do this would be to generate all possible permutations for the number of variables, however as we are testing a number of variables like 13 or even higher, this would take a lot of time, so for the sake of this assignment the number of permutations tried out is reduced to a much smaller number than (number of variables)!.

BDD_create_with_best_order

```
public static BDD BDD_create_with_best_order(String bfunction) {
    Set<Character> variables;
    variables = retrieveVariablesFromBfunction(bfunction);
    Set<String> permutations = getPermutations(variables);
    List<String> permutationsList = new ArrayList<>(permutations);
    List<Integer> nodeCounts = new ArrayList<>();

    for (int i = 0; i < permutationsList.size(); i++){
        BDD bdd = BDD_create(bfunction, permutationsList.get(i));
        nodeCounts.add(bdd.size);
    }

    int minimum = Collections.min(nodeCounts);

    for (int j = 0; j < nodeCounts.size(); j++){
        if (nodeCounts.get(j) == minimum){
            BDD bestBDD = BDD_create(bfunction, permutationsList.get(j));
            return bestBDD;
        }
    }
    return null;
}
```

Firstly, the variables are retrieved from the input Boolean function and random permutations are generated using the getPermutations method. Then a list of node counts is initialized where the size of each bdd created for certain order is stored. We simply create the BDDs in a for loop and then retrieve the minimum size from the nodeCounts list, where the index of that minimum is also index of that certain permutation of variables in a permutations list. Finally, the "best order BDD" is returned.

Helper methods for BDD_create_with_best_order

1. retrieveVariablesFromBfunction

This method has already been described in the helper methods of BDD_create.

2. getPermutations

This method generates unique permutations of a set of characters by iterating through each pair of characters and swapping their positions. It first converts the input set of characters into a list, then creates a nested loop to iterate over each pair of characters, and swaps their positions to create a new permutation. Finally, it adds each permutation to a set to ensure that only unique permutations are returned.

```
public static Set<String> getPermutations(Set<Character> variables) {
    Set<String> permutations = new HashSet<>();
    List<Character> list = new ArrayList<>(variables);
    int n = list.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            List<Character> permutation = new ArrayList<>(list);
            Collections.swap(permutation, i, j);

permutations.add(permutation.stream().map(String::valueOf).collect(Collecto
rs.joining()));
        }
    }
    return permutations;
}
```

Implementation of BDD_use

Description

This method serves for obtaining a specific result of Boolean function base on what kind of input is passed into the function. It takes in a BDD and the input (in form of string "0100") as arguments and returns either 1, 0 or E as for error. The evaluation is performed in a evaluateBDD helper method that will be descriped in this section.

BDD_use

```
public static char BDD_use(BDD bdd, String input_values){
    if (input_values.length() != bdd.numOfVariables ||
!input_values.matches("[01]+")){
        return 'E';
    }else{
        boolean result =
evaluateBDD(bdd.getRoot(),bdd.orderOfVariables,input_values);
        if (result){
            return '1';
        }else if (!result){
            return '0';
        }
    }
    return 'E';
}
```

The error in this case happens if the input has values for more or less variables than there are in the BDD or if there are any other characters different from 0 or 1.

evaluateBDD

This method takes the BDD node as an argument, the specific order of variables of the BDD, and the input values. The BDD root is passed into the node argument.

```
public static boolean evaluateBDD(BDDNode node, String variableOrder,
String inputValues){
    if (node == null){
        return false;
    }

    if (node.isTerminal()) {
       return node.getValue();
   }

    int varIndex = variableOrder.indexOf(node.getVarIndex());
    boolean varValue = inputValues.charAt(varIndex) == '1';
    BDDNode child = varValue ? node.getHighChild() : node.getLowChild();
    return evaluateBDD(child, variableOrder, inputValues);
}
```

If the node is null, it returns false, other way, it returns the value at the terminal node once it is reached which is the result for that input of values. If the current node is not a terminal node, it gets the index of the variable associated with this node from the variableOrder string. It then gets the value of this variable from the inputValues string, which is a binary string of 0s and 1s representing the values of each variable in variableOrder. It chooses the appropriate child node of the current node based on the value of the variable and recursively calls evaluateBDD on the chosen child node with the same variableOrder and inputValues. The method keeps calling itself recursively until it reaches a terminal node, at which point it returns the value of that node.

Testing

Testing the correctness of outputs from BDD

Compulsory part of the assignment to make sure that the BDD produces 100 percent corrects outputs. To ensure this, I have a function that firstly rewrites the Boolean function into a form so that Java can parse it as a boolean function and then I evaluate it in another function. Then the ouput from this Java computed result is compared to the output from BDD and success rate is evaluated.

1. rewriteBfunction

```
        public static String rewriteBfunction(String function){
        function = function.replace("+", "||");

        String[] terms = function.split("\\|\\|");
        for (int i = 0; i < terms.length; i++) {
            String term = terms[i].trim();
            String modifiedTerm = "";
            for (int j = 0; j < term.length(); j++) {
                char c = term.charAt(j);
                if (Character.isLetter(c)) {
                    if (j != term.length()-1) {
                        modifiedTerm += c + " && ";
                    }else {
                        modifiedTerm += c;
                    }
                } else {
                    modifiedTerm += c;
                }
            }
            terms[i] = modifiedTerm;
        }
        function = String.join(" || ", terms);
        return function;
     }
```

This function modifies a Boolean function into a function that can be then parsed and evaluated by Java. The + signs are replaced by ||, then the function is split into terms and into each term && is added after the variables as AND sign. Then each modified term replaces the original term and the terms are again rejoined with || OR operator. The function returns the new modified Boolean function.

2. evaluateBfunction

   This method evaluates the Boolean function using the Java boolean logical operators.

```
public static Boolean evaluateBfunction (String function, String input,
String order){
    boolean[] inputs = new boolean[input.length()];
    for (int i = 0; i < input.length(); i++) {
        inputs[i] = (input.charAt(i) == '1');
    }

    function = rewriteBfunction(function);

    for (int i = 0; i < order.length(); i++) {
        char var = order.charAt(i);
        boolean value = inputs[i];
        function = function.replaceAll(Character.toString(var),
Boolean.toString(value));

    }

    function = function.replaceAll("!true", "false");
    function = function.replaceAll("!false","true");

    boolean result = false;
    for (String conj : function.split("\\|\\|")) {
        boolean b = true;
        for (String literal : conj.split("&&"))
            b &= Boolean.parseBoolean(literal.trim());
        result |= b;
    }

    if (result) {
        return true;
    } else if (!result) {
        return false;
    }
    return null;
}
```

This method takes in a Boolean function as a string, along with an input string and a string representing the variable order. It first converts the input string into an array of boolean values, and then applies a series of transformations to the Boolean function string, including replacing variables with their corresponding boolean values and simplifying negations. The resulting function is then evaluated using a nested loop that iterates over each disjunctive clause and each conjunctive literal, evaluating each literal's boolean value and checking if the resulting conjunction evaluates to true. Finally, the method returns either true, false, or null, depending on the evaluation of the function.

3. testCorrectnessOfAllPossibleOutputs

   This is the function where comparison of all results happen and success rate is computed.

```
public static void testCorrectnessOfAllPossibleOutputs(BDD bdd){
    List<String> perms =
generatePermutationsForTruthTable(bdd.numOfVariables);
    int allOutputs = perms.size();
    int correctOutputsFromBDD = 0;
    char result;


    for (int i = 0; i < perms.size(); i++){
        if (evaluateBfunction(bdd.bfunction, perms.get(i),
bdd.orderOfVariables) == true){
            result = '1';
        }else if  (evaluateBfunction(bdd.bfunction, perms.get(i),
bdd.orderOfVariables) == false){
            result = '0';
        }else result = 'E';

        if (result != BDD_use(bdd, perms.get(i))){
            System.out.println("ERROR");
        }else{
            correctOutputsFromBDD++;
        }
    }
    System.out.println("Number of all possible outputs/Number of correct
outputs from BDD: " + allOutputs + "/" + correctOutputsFromBDD);
    double successRate = correctOutputsFromBDD/allOutputs * 100;
    System.out.println("Success rate: " + successRate + "%");
}
```

Firstly, all the possible permutations are created for the “truth table” in a generatePermutationsForTruthTable.

```
public static List<String> generatePermutationsForTruthTable(int n) {
    List<String> permutations = new ArrayList<>();
    int numPermutations = (int) Math.pow(2, n);

    for (int i = 0; i < numPermutations; i++) {
        String binaryString = Integer.toBinaryString(i);
        while (binaryString.length() < n) {
            binaryString = "0" + binaryString;
        }
        permutations.add(binaryString);
    }

    return permutations;
}
```

This function simply returns all the possible permutations of binary string for a specific number of variables.

Then the results from the evaluateBfunction method are set to character type 1 or 0 to match the data type of BDD_use and the for each permutation, these two are compared to find out whether they match and thus whether BDD_use outputs correct result. Finally, the all possible outputs to correct outputs ratio is printed as well as the success rate.

Helper methods for testing scenarios

In order to perform tests, the methods for creating random functions and order for them are necessary.

1. generateRandomFunction

```
    public static String generateRandomFunction(int numVariables) {
        Random RANDOM = new Random();
        StringBuilder sb = new StringBuilder();

        int numTerms = RANDOM.nextInt(5,10);
        for (int i = 0; i < numTerms; i++) {
            int termLength = RANDOM.nextInt(numVariables) + 1;

            Set<Character> variablesUsed = new HashSet<>();

            StringBuilder termBuilder = new StringBuilder();
            for (int j = 0; j < termLength; j++) {
                char variable;
                boolean negate;

                do {
                    negate = RANDOM.nextBoolean();
                    variable = (char) ('A' + RANDOM.nextInt(numVariables));
                } while (variablesUsed.contains(variable) ||
                        (negate &&
variablesUsed.contains(Character.toLowerCase(variable)) ||
                            (!negate &&
variablesUsed.contains(Character.toUpperCase(variable))));
                if (negate) {
                    termBuilder.append("!");
                }
                termBuilder.append(variable);

                variablesUsed.add(variable);
                if (negate) {
                    variablesUsed.add(Character.toLowerCase(variable));
                } else {
                    variablesUsed.add(Character.toUpperCase(variable));
                }
            }
            sb.append(termBuilder.toString()).append("+");
        }

        sb.deleteCharAt(sb.length() - 1);
        return sb.toString();
    }
```

This method generates a random Boolean function with a given number of variables. The function is generated as a disjunction of random terms, where each term is a conjunction of variables or their negations. The method generates a random number of terms between 5 and 10, and for each term, it generates a random number of variables between 1 and the number of variables. It then randomly selects variables or their negations for the term, making sure that each variable is only used once per term and that the negation of a variable is only used if the variable itself is also used in the term. The method returns the generated function as a string.

2. generateRandomOrder

```
public static String generateRandomOrder(String bfunction) {
    Set<Character> variables = retrieveVariablesFromBfunction(bfunction);
    List<Character> varList = new ArrayList<>(variables);
    int n = varList.size();

    Random rand = new Random();
    for (int i = n - 1; i > 0; i--) {
        int j = rand.nextInt(i + 1);
        char temp = varList.get(i);
        varList.set(i, varList.get(j));
        varList.set(j, temp);
    }

    StringBuilder sb = new StringBuilder(n);
    for (char c : varList) {
        sb.append(c);
    }
    return sb.toString();
```

This method takes a boolean function as input and returns a randomly shuffled string representing the order of variables to be used in evaluating the function. First, it calls another method retrieveVariablesFromBfunction to extract all the unique variables from the input boolean function. It then shuffles these variables using the Fisher-Yates algorithm, which randomly swaps each variable in the list with another one. Finally, the shuffled list of variables is converted back to a string and returned as the output.

Test scenarios

In my implementation, I created 4 test scenarios – three for testing the BDD and its correctness as well as other things, and one solely for time complexity purposes.

1. testTheBDD
   This test takes in certain function and variable order and it computes the time for creation of this BDD, how many nodes it would have with no reduction, how many nodes this BDD has after reductions, how many nodes would the BDD with best order have after reductions, reduction rate of both, input order to best order size ratio, and also tests the correctness of outputs from both of these BDDs.

2. testTheBDDRandom
   This test does the exact same thing as the first one despite not using a certain input but generating a random function and order for it. The only input is the preffered number of variables for this test.

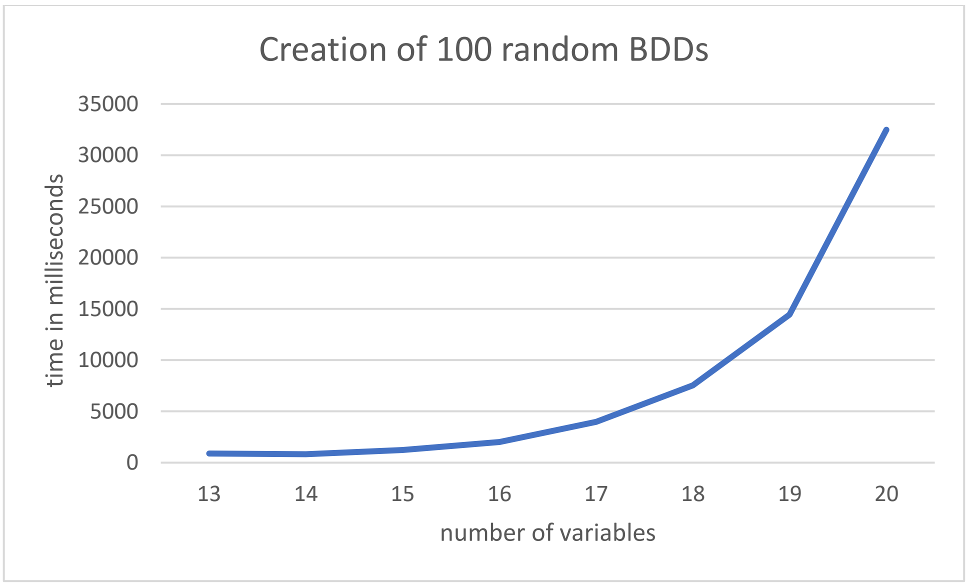
3. testTheBDD100Times
   This test calls the testTheBDDRandom hundred times, so 100 random function of certain amount of variables from input are tested and also time complexity of creating 100 random BDDs is computed.

4. timeComplexityTest
   This test takes no parameters but provides an automatic test for time complexity of creating 100 random functions consisting of amount of variables from 13 to 20.

Time and space complexity

Time complexity

To evaluate the time complexity of BDD creation, I perform the timeComplexityTest().



We can see that the time needed for creation of 100 BDDs for each amount of variables grows each time a variable is added.

Space complexity

Regarding the space complexity, we can compute as the amound of nodes each BDD have. The best case scenario is when the BDD can be reduced in a way that it has much less nodes than the full BDD. Also, creating the BDD with BDD_create_with_best_order should ensure the best case scenario for each BDD, however this would be 100 percent functional if that function used all the possible permutations for the certain amount of variables.

Conclusion

My task was to implement a reduced BDD and mainly the three functions BDD_create, BDD_create_with_best_order, and BDD_use.

My implementation of BDD ensures that it is reduced throughout the proccess of creation by holding unique nodes in a hash table and thus applying reduction of type I and also checking each added node for reduction of type S. In addition, boolean algebra laws are used on the terms.

For testing the correctness of BDD, I evaluate the function which BDD represents using the Java boolean logical operators. Then I compare these ouputs and print out the success rate to make sure the implementation is 100 percent correct.

The time complexity suggests that the bigger the number of variables, the more time it is required for its creation.

Resources

For this implementation I mainly got ideas from the DSA lectures from 7th and 8th week, which are all available in the document server on AIS, I also used some external materials and information from the internet:

1. *Binary decision diagram* (2022) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/binary-decision-diagram/ (Accessed: May 2, 2023).