Project 2 (Binary decision diagram)

Data structures and algorithms

Márk Bartalos

Seminars: Monday 9:00-11:00

Seminars with: Ing. Lukáš Kohútka, PhD.

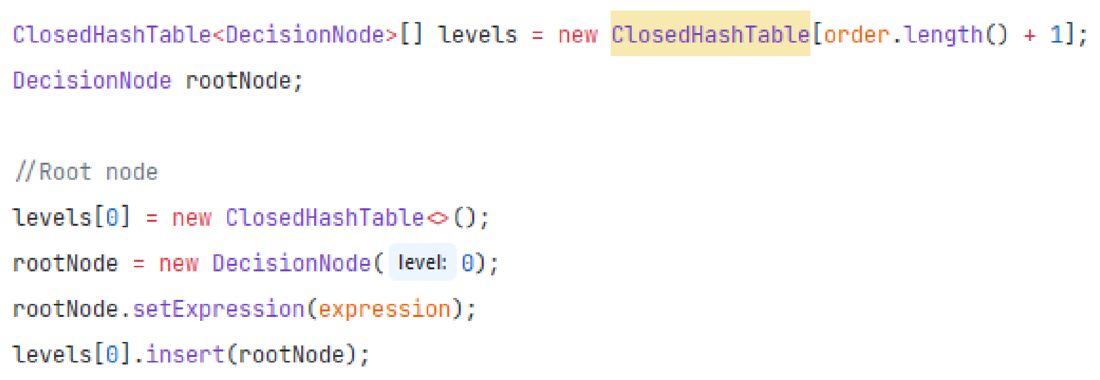
Table of content

Binary decision diagram

Binary decision diagram is a data structure similar to binary trees. It is used to determine the value of a Boolean expression for a given input. A BDD can be generated for any kind of Boolean function, and the emphasis is on the “boolean” keyword, as the BDD uses two children for each node.

Function BDD_create and tree generation

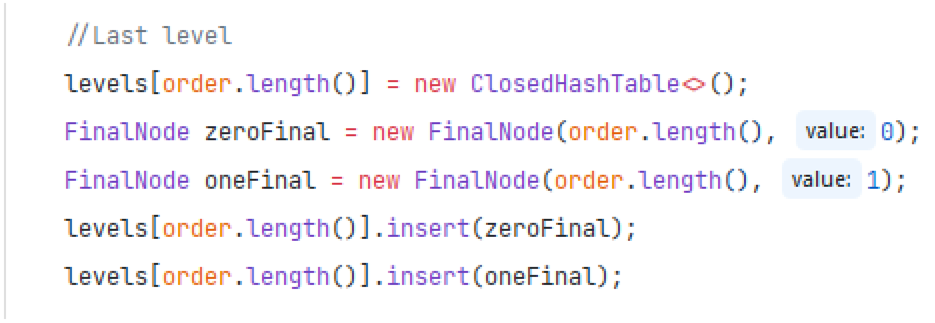
For generating the binary decision diagram from an expression, I choose a linear method, with level-by level generation. That way each level is generated one after the other. Each level is represented by a hash map (linked list type) containing the nodes for the given level.



```
ClosedHashTable<DecisionNode>[] levels = new ClosedHashTable[order.length() + 1];
DecisionNode rootNode;

//Root node
levels[0] = new ClosedHashTable<>();
rootNode = new DecisionNode( level: 0);
rootNode.setExpression(expression);
levels[0].insert(rootNode);
```

Generation of nodes

Nodes are generated level by level. The first node created is always the root nodes and the two of the final nodes. The root node always contains the whole expression and is always on the first level. The final nodes are created after, and they are always on the last levels.

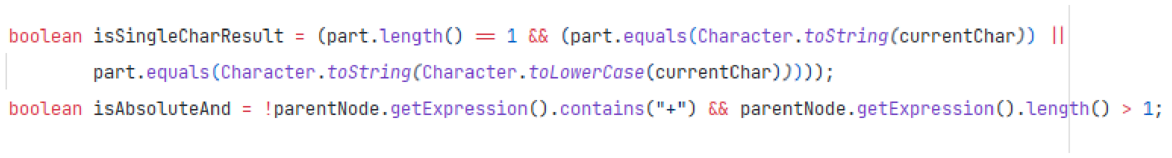


```
//Last level
levels[order.length()] = new ClosedHashTable<>();
FinalNode zeroFinal = new FinalNode(order.length(),  value: 0);
FinalNode oneFinal = new FinalNode(order.length(),  value: 1);
levels[order.length()].insert(zeroFinal);
levels[order.length()].insert(oneFinal);
```

After these nodes are generated, the program continues to generating the nodes from the expression. As we use DNF type expressions, we first split up the string by the ‘+’ character, and going by the “order” variable, which contains the orders of the characters in the expression in all uppercase format, we check the current character in the expression pare-by part. One character in “order” represents one level in the diagram. We loop through the expression parts using a for loop and we control the existence of the current character in each part. Uppercase variant of the character represents the normal variant and the lowercase represents a negated character. If the part contains the uppercase version of the character than it will be added to the right node, if it contains the lowercase (negated) variant it will be added to the left node, if contains neither, then its added to both nodes, if it contains both, then it is a contradiction and this part can be skipped or if it is the only part than both of the children nodes will be 0. Also, in the case that the whole expression depends on a single character, be in a multiplication ("AB"), or by itself ("A"), the algorithm will evaluate the left, or right side of the node based on the case of the character, with a final node (0 or 1).

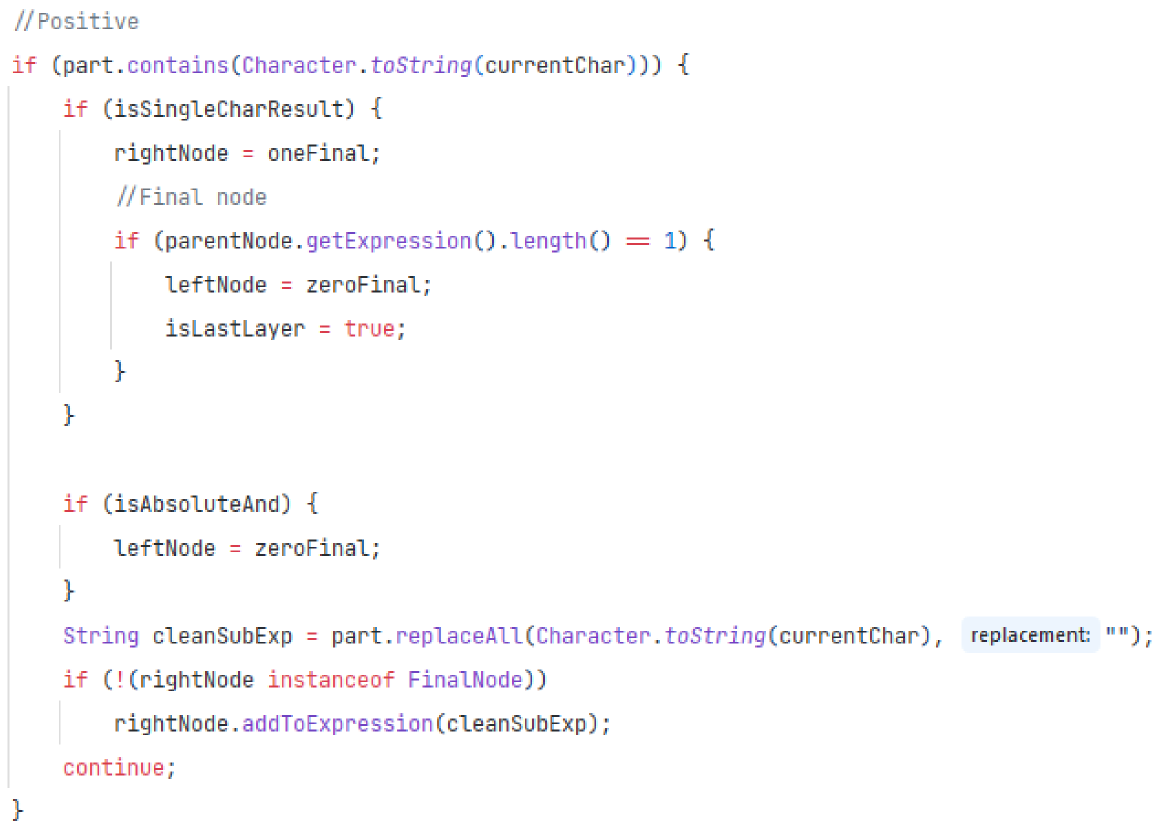
After the expression part is evaluated, before adding to a node, the occurrences of the current character are removed and then the expression is added to the node it belongs to.

Filtering expressions:



```java
boolean isSingleCharResult = (part.length() == 1 && (part.equals(Character.toString(currentChar)) ||
        part.equals(Character.toString(Character.toLowerCase(currentChar)))));
boolean isAbsoluteAnd = !parentNode.getExpression().contains("+") && parentNode.getExpression().length() > 1;
```

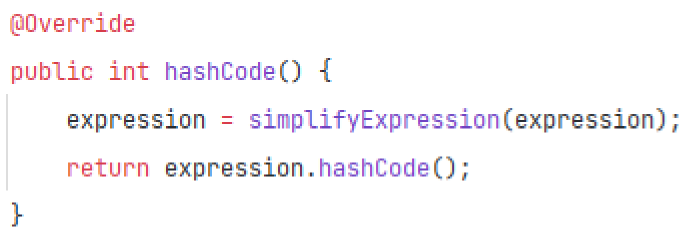
Positive part:



```java
//Positive
if (part.contains(Character.toString(currentChar))) {
    if (isSingleCharResult) {
        rightNode = oneFinal;
        //Final node
        if (parentNode.getExpression().length() == 1) {
            leftNode = zeroFinal;
            isLastLayer = true;
        }
    }

    if (isAbsoluteAnd) {
        leftNode = zeroFinal;
    }
    String cleanSubExp = part.replaceAll(Character.toString(currentChar),  replacement: "");
    if (!(rightNode instanceof FinalNode))
        rightNode.addToExpression(cleanSubExp);
    continue;
}
```

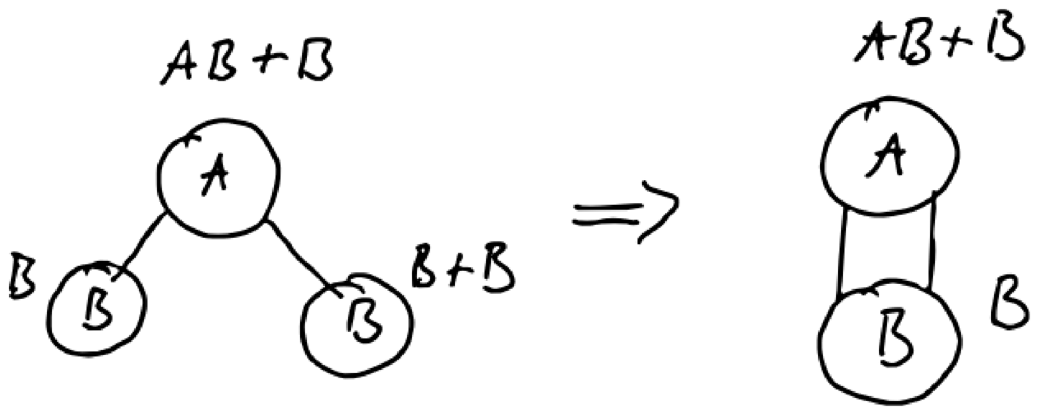
Reduction type I

All of the levels all saved in an array containing the hash maps. Hash maps are used for reduction type I during runtime. Hash map is used from the previous project and is set up in a way that is passes the hashing of the data to the DecisionNode object itself. This property allowed me to simplify that given node's own expression before hashing and simply eliminate duplicate parts inside the expression itself using another temporary inside hash map.



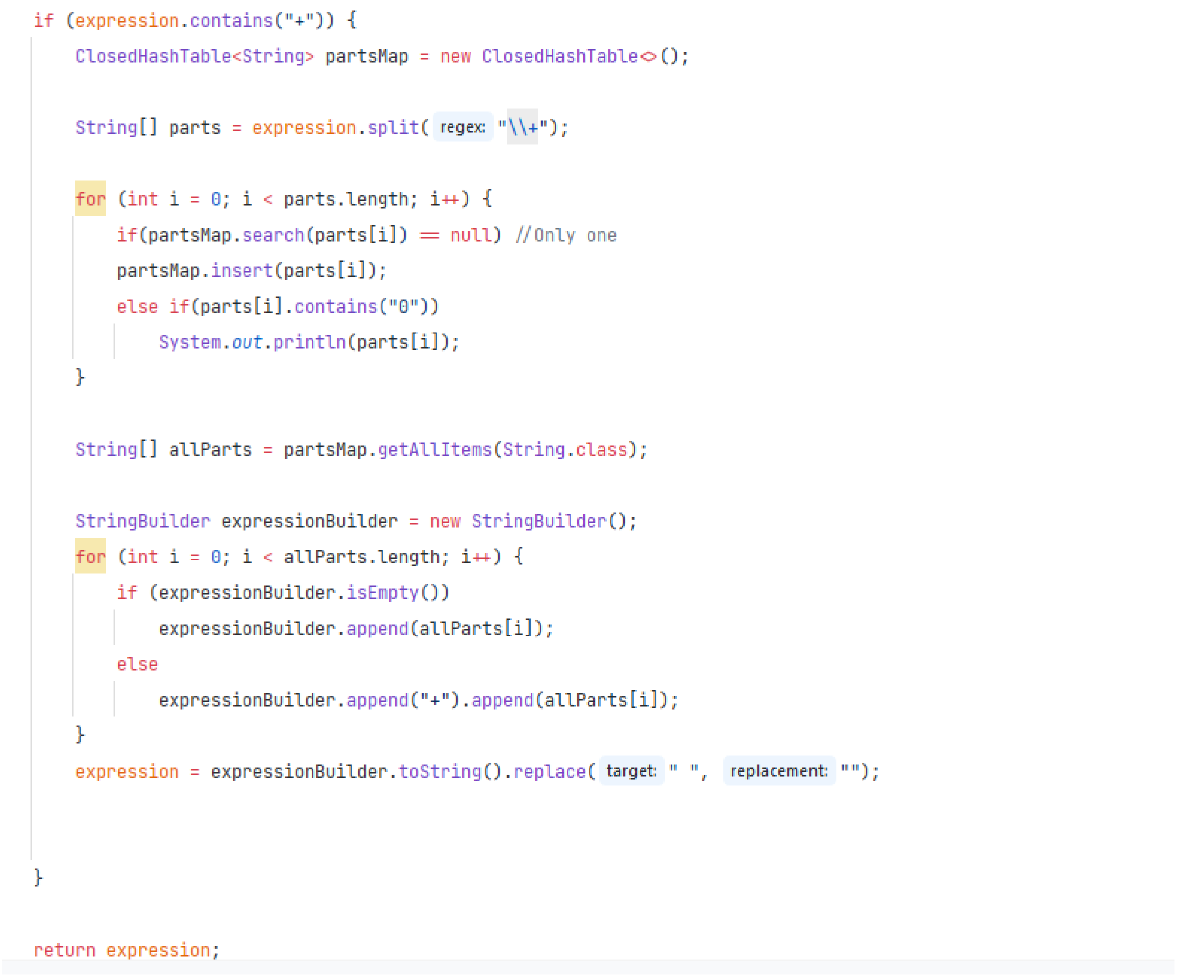
```java
@Override
public int hashCode() {
    expression = simplifyExpression(expression);
    return expression.hashCode();
}
```

The nodes are hashed based on the expression, that way nodes with the same expression will hash to the same hash code. That way we can ensure that on a level one node can exist with the same expression. During generation if the generated node's hash is already in the hash map it retrieves the existing node from the hash map and connects it to the parent, instead of creating a new node. New nodes are always added to the hash map, with the exception of the final nodes (0 and 1).



Here on the diagram, you can see that how the simplification will look, first the first B node is created, then when hashing the second node ("B+B") the expression will be simplified to "B", which already exists in the level so the right node will also point to the same node.

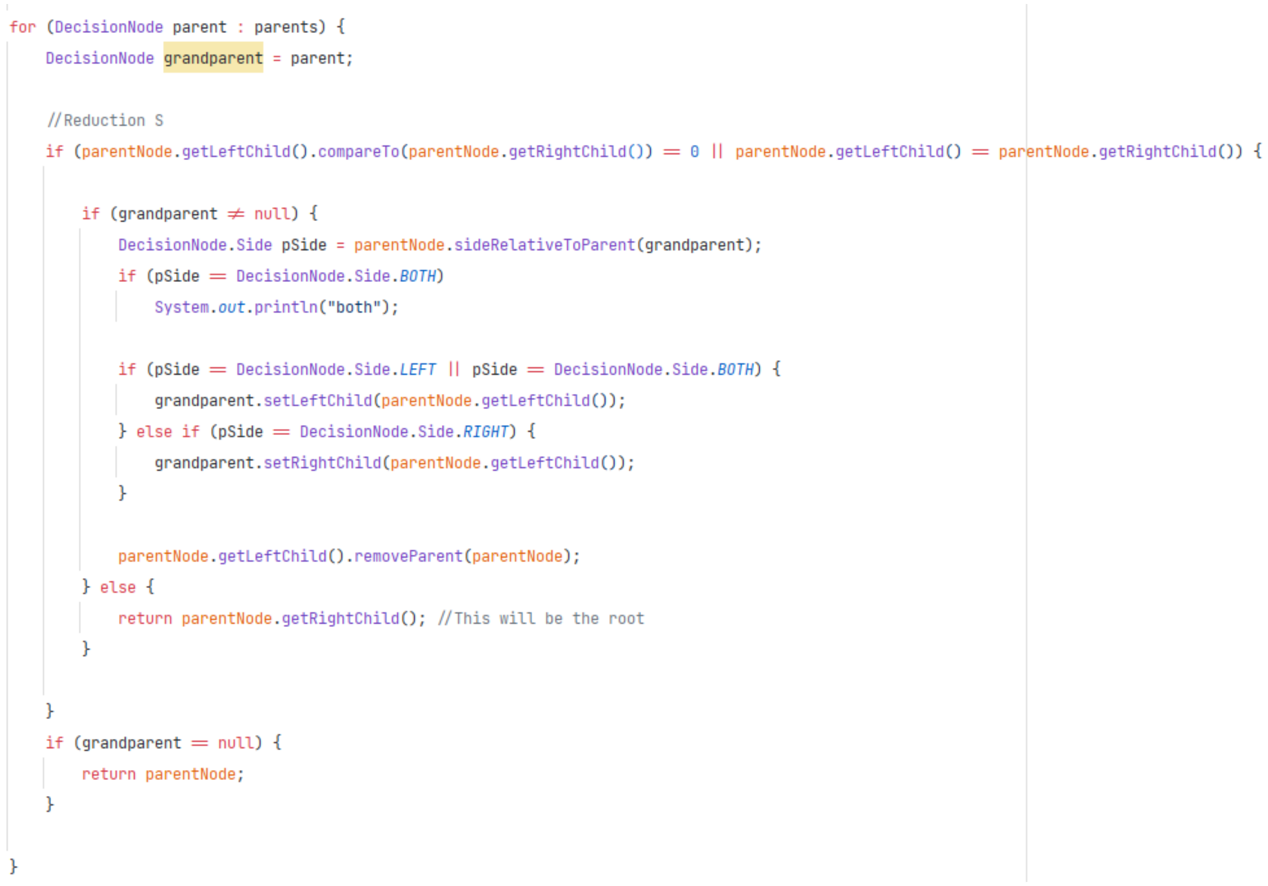
Here on the code sample, you can see the simplification of the expression. This code is from the simplifyExpression method inside a DecisionNode itself. It looks at the expression, if it contains multiple parts, it splits by the ‘+’ character, because we are using DNF expressions. All parts, which weren’t already added adds to a temporary hash map, then reconstructs the simplified expression using a StringBuilder (same as String but more effective in Java, because Strings are immutable) and returns the result. This all happens before hashing.



```
if (expression.contains("+")) {
    ClosedHashTable<String> partsMap = new ClosedHashTable<>();

    String[] parts = expression.split( regex: "\\+");

    for (int i = 0; i < parts.length; i++) {
        if(partsMap.search(parts[i]) == null) //Only one
        partsMap.insert(parts[i]);
        else if(parts[i].contains("0"))
            System.out.println(parts[i]);
    }

    String[] allParts = partsMap.getAllItems(String.class);

    StringBuilder expressionBuilder = new StringBuilder();
    for (int i = 0; i < allParts.length; i++) {
        if (expressionBuilder.isEmpty())
            expressionBuilder.append(allParts[i]);
        else
            expressionBuilder.append("+").append(allParts[i]);
    }
    expression = expressionBuilder.toString().replace( target: " ",  replacement: "");


}

return expression;
```

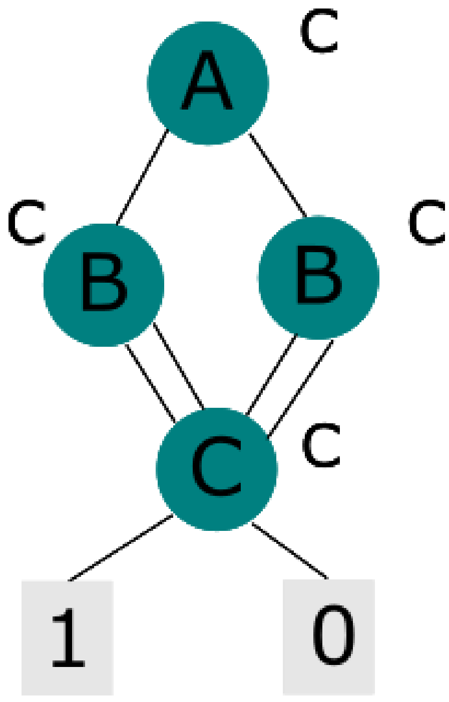
Reduction type S

Reduction S happens after when a level has been generated, we go through all the nodes in that level to generate the next level using the nodes in the previous level. After assigning the new children S reduction is initiated. Reduction S happens in combination iteratively and recursively. First let's start with the problem, this would be a simple process if all items had only one parent and the reduction could be done simply recursively. But reduction I is done before reduction S, so one item has many parents, for this reason we have to go through each parent of each node, do the reduction, remove the reduced parent and finally recurse on the new or left parents until we arrive at the root node (at a node who doesn't have a parent). The reduction is done by comparing a node's two children and if they match (they are compared by expression), if the current node has a parent the side relative to the parent is determined and then one of the children is assigned to its grandparent on the appropriate side. If the current node doesn't have a parent, but both its children are equal, then the root is returned as one of its children. If the grandparent is null and the children aren't equal, the current node is returned as root.

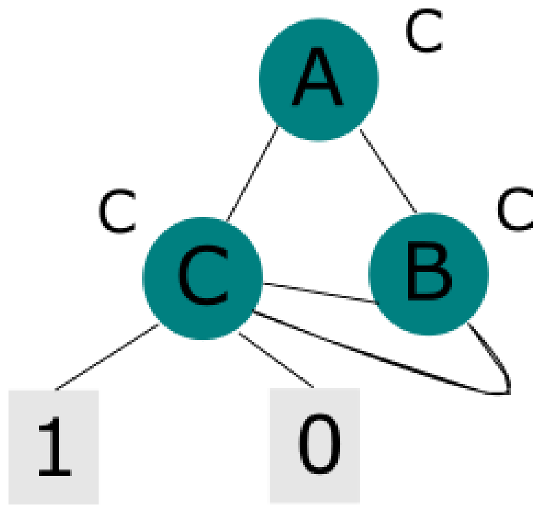


```
for (DecisionNode parent : parents) {
    DecisionNode grandparent = parent;

    //Reduction S
    if (parentNode.getLeftChild().compareTo(parentNode.getRightChild()) == 0 || parentNode.getLeftChild() == parentNode.getRightChild()) {

        if (grandparent ≠ null) {
            DecisionNode.Side pSide = parentNode.sideRelativeToParent(grandparent);
            if (pSide == DecisionNode.Side.BOTH)
                System.out.println("both");

            if (pSide == DecisionNode.Side.LEFT || pSide == DecisionNode.Side.BOTH) {
                grandparent.setLeftChild(parentNode.getLeftChild());
            } else if (pSide == DecisionNode.Side.RIGHT) {
                grandparent.setRightChild(parentNode.getLeftChild());
            }

            parentNode.getLeftChild().removeParent(parentNode);
        } else {
            return parentNode.getRightChild(); //This will be the root
        }

    }
    if (grandparent == null) {
        return parentNode;
    }

}
```

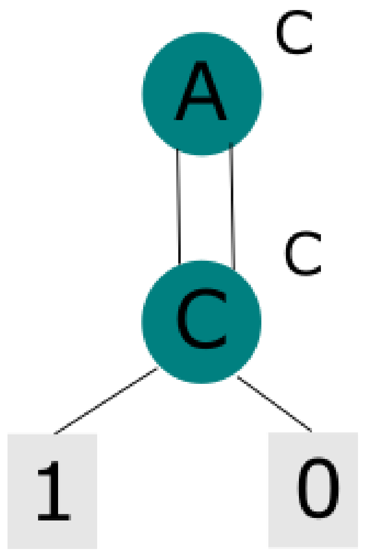
Explanation of the process (probably not the best example given that the two B nodes are equal and will be only one node thanks to reduction I, but for demonstration it will server its purpose):



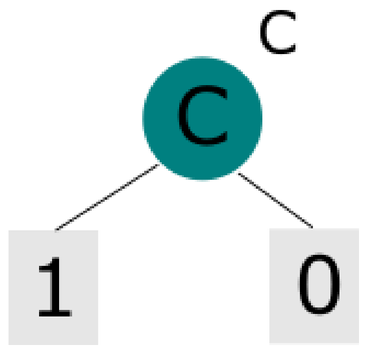
1. Here we have a decision diagram, with the expression "C" and on the "ABC" character set, this diagram is partly reduced using reduction type I. Currently we are at node B.



2. C node will know its parents, even if they are added in later iterations (as probably in this case). C will be passed up to be A's left children as B was also a left children and the checking continues linearly. Now in the next step we will look at the next parent, repeat the same reduction for this side then recurse with the parents of B (the previous parent of C, this works because C is the replacement for B, thus will have the same parent as the previous B had).



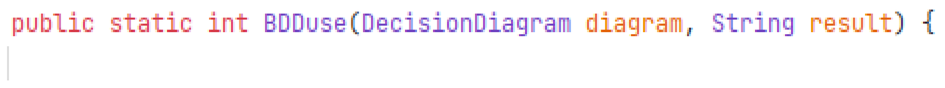
3. We do the same reduction to C's parent, we got here by recursion from the previous step. We reduce it, by returning the right child (either of the nodes will work) as the new root.



4. We have C as the final and new root.

Function BDD_use and traversal of the diagram

BDDuse method is part of the same DecisionDiagram class as a static method, it returns an integer 0,1 based on the result of the evaluation and -1 if an has error occurred.

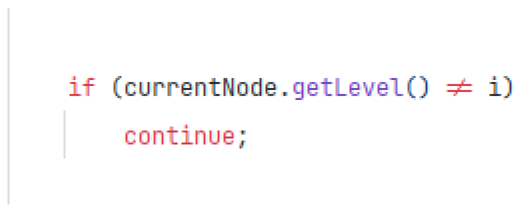


```
public static int BDDuse(DecisionDiagram diagram, String result) {
```

BDDuse requires the decision diagram as its first parameter and a string consisting of values for each charcter based on the order, for which the tree was generated. For example, if the tree was generated with “order” characters being “ABCD” then “1001” would mean that A = 1, B = 0, C = 0 and D = 1.

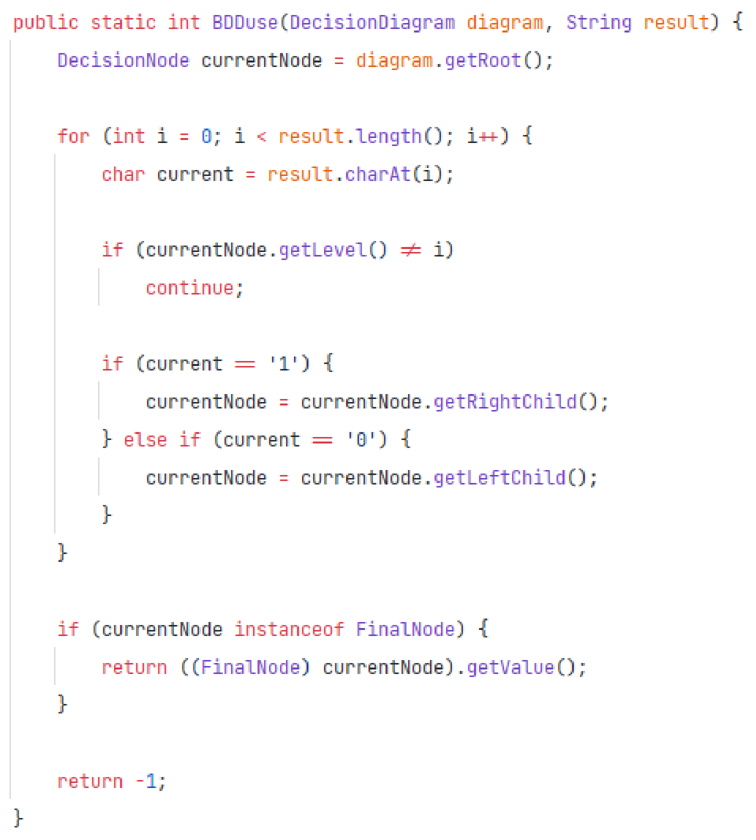
This methods traversal of the diagram is based on level as we will have reduced decision diagrams, so we can’t expect to just traverse in the default way by going left or right based on the current input number (right for 1, and left for 0).

Going by levels is a very similar approach, every node has its own level, it is determined at creation and before any kind of reduction, and is not changed. During reduction, nodes are only passed upward, so the level will always stay correct, as when doing reduction type S the upper nodes are the redundant ones. So, when we want to traverse the diagram, we first look at the level of the node, if the level of the node doesn’t equal the current inputs order number, we wait an iteration as the input is irrelevant (in both cases will be the same).



```
    if (currentNode.getLevel() ≠ i)
        continue;
```

If the level and the order number matches, we branch left if the current input is 0, or right if is 1. If we iterated through the whole input string, we check the type of the resulting node from the branching. If the final node is an instance of FinalNode (type of DecisionNode) than we arrived at the result, which we return with getValue() method from the node. In other case we return -1, as something has gone wrong.

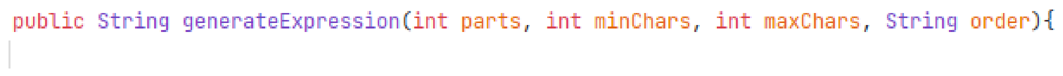


```java
public static int BDDuse(DecisionDiagram diagram, String result) {
    DecisionNode currentNode = diagram.getRoot();

    for (int i = 0; i < result.length(); i++) {
        char current = result.charAt(i);

        if (currentNode.getLevel() != i)
            continue;

        if (current == '1') {
            currentNode = currentNode.getRightChild();
        } else if (current == '0') {
            currentNode = currentNode.getLeftChild();
        }
    }

    if (currentNode instanceof FinalNode) {
        return ((FinalNode) currentNode).getValue();
    }

    return -1;
}
```

Testing

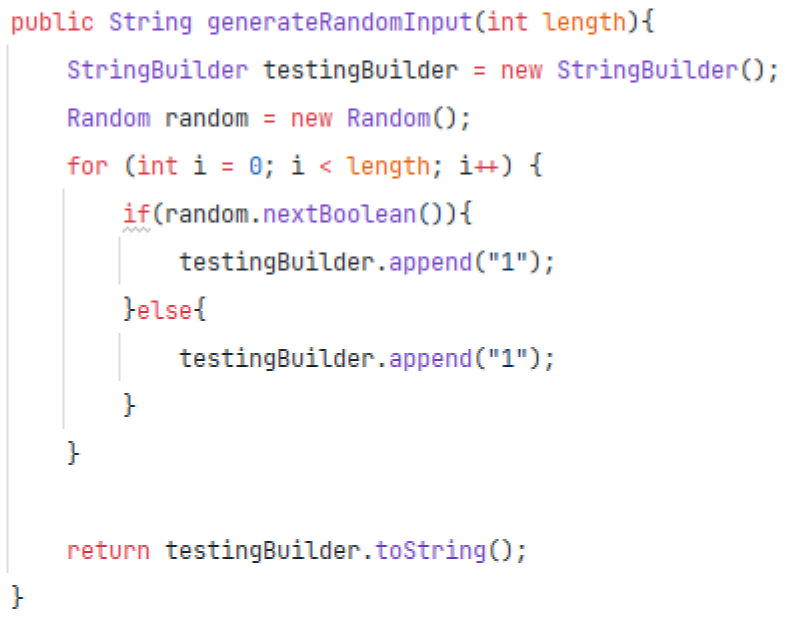
The testing of the binary decision diagram is split into two parts. On part is testing the performance of the creation and the use of the diagram for a random input, here we don't care about the correctness of the result, we just test the creation and the use time. The other case in correctness testing, here we have smaller inputs, but we test the correctness of each and every combination of input and output. All test cases are generated automatically.

For testing there is a class called Tester, which has a method for generating random expressions with parameters such as minimum character per part, part number, max length per part and order of course.



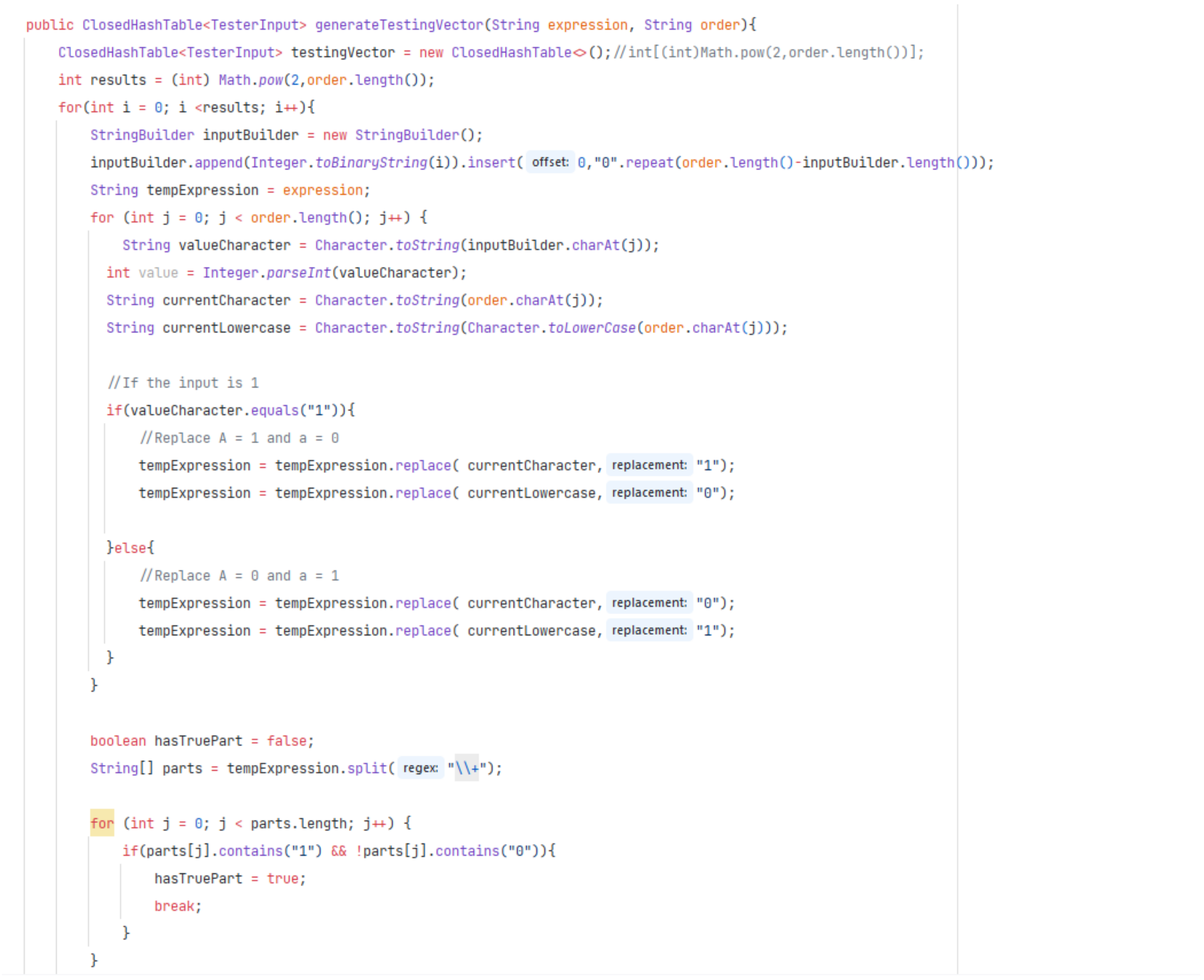
```java
public String generateExpression(int parts, int minChars, int maxChars, String order){
```

In order to test BDDuse's with a simple random input there is a helper method generateRandomInput, which will generate a random input consisting of 0s and 1s with the length on N.



```java
public String generateRandomInput(int length){
    StringBuilder testingBuilder = new StringBuilder();
    Random random = new Random();
    for (int i = 0; i < length; i++) {
        if(random.nextBoolean()){
            testingBuilder.append("1");
        }else{
            testingBuilder.append("1");
        }
    }

    return testingBuilder.toString();
}
```

There is a method for generating the testing numbers and their results by replacing the expression characters one-by one with 1 or 0, depending on the input and the case of the character. For example, if the input is 1 then the uppercase version of the current character will be one and the lowercase will be 0, in other case if the input is 0, then the reverse will happen. When all characters are replaced then we evaluate the expression part by part until we arrive at the end or get a part with a value of 1. A part will have a value of 1, if it doesn't contain any 0s. Finally we add the input and the output to the results hash table and continue for the remaining combinations. The number of test cases will be 2^N, where N is the unique character count (called as “order”).



```
public ClosedHashTable<TesterInput> generateTestingVector(String expression, String order){
    ClosedHashTable<TesterInput> testingVector = new ClosedHashTable<>();//int[(int)Math.pow(2,order.length())];
    int results = (int) Math.pow(2,order.length());
    for(int i = 0; i <results; i++){
        StringBuilder inputBuilder = new StringBuilder();
        inputBuilder.append(Integer.toBinaryString(i)).insert( offset: 0,"0".repeat(order.length()-inputBuilder.length()));
        String tempExpression = expression;
        for (int j = 0; j < order.length(); j++) {
            String valueCharacter = Character.toString(inputBuilder.charAt(j));
          int value = Integer.parseInt(valueCharacter);
          String currentCharacter = Character.toString(order.charAt(j));
          String currentLowercase = Character.toString(Character.toLowerCase(order.charAt(j)));

          //If the input is 1
          if(valueCharacter.equals("1")){
              //Replace A = 1 and a = 0
              tempExpression = tempExpression.replace( currentCharacter, replacement: "1");
              tempExpression = tempExpression.replace( currentLowercase, replacement: "0");

          }else{
              //Replace A = 0 and a = 1
              tempExpression = tempExpression.replace( currentCharacter, replacement: "0");
              tempExpression = tempExpression.replace( currentLowercase, replacement: "1");
          }
        }

        boolean hasTruePart = false;
        String[] parts = tempExpression.split( regex: "\\+");

        for (int j = 0; j < parts.length; j++) {
            if(parts[j].contains("1") && !parts[j].contains("0")){
                hasTruePart = true;
                break;
            }
        }
```

Performance testing

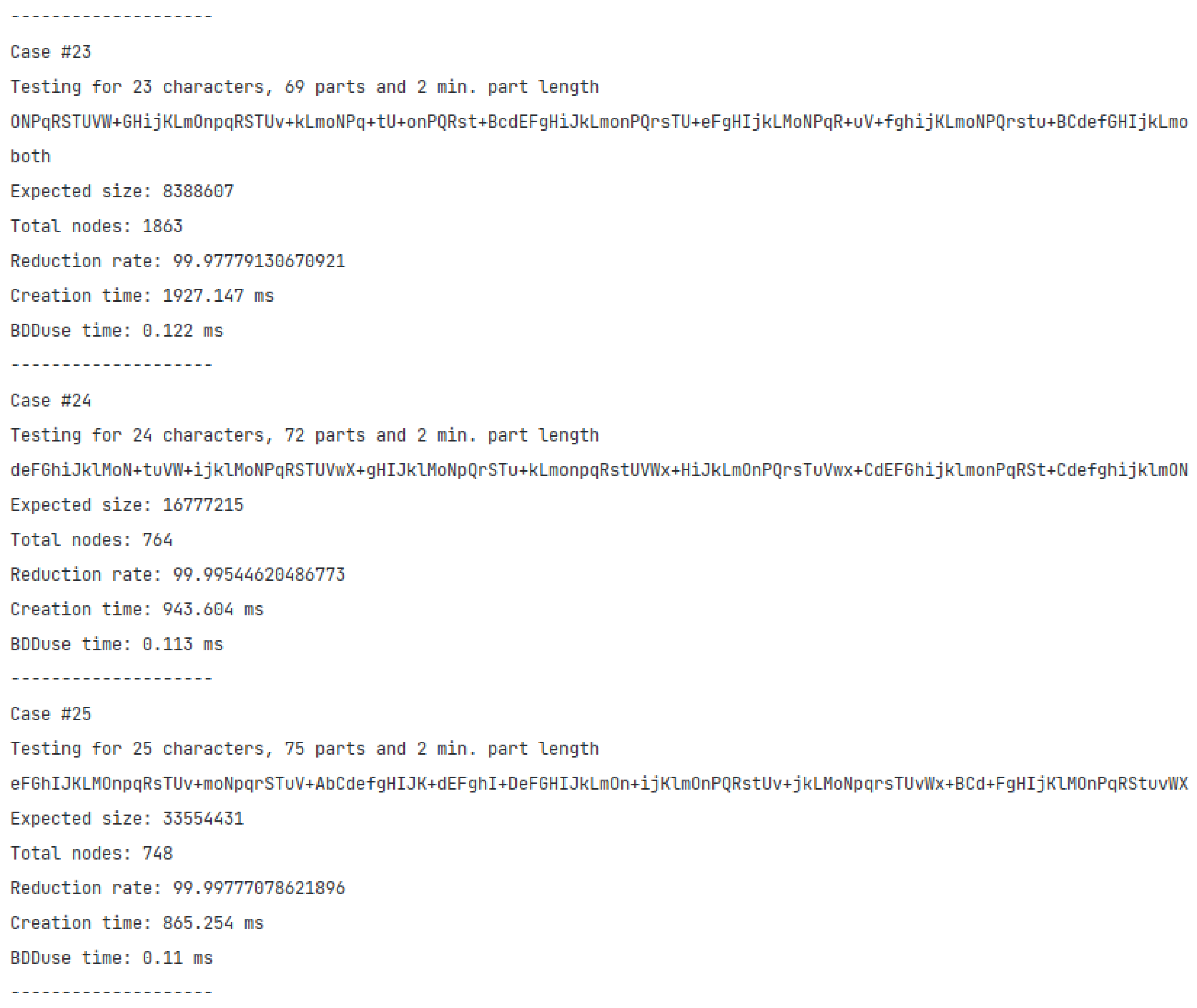
Performance testing is meant to test the performance, reduction rate of the BDDcreate method and the performance of the BDDuse method. Here the expressions are much larger. Testing goes up to 25 characters and up to 75 parts with different lengths. The expressions are generated automatically and randomly using the tester class. First, we measure the generation time then we test the usage time with a sample input.

Character count	2	4	8	12	18	20	21	23	25
BDD_create (ms)	47	19	46	131	382	511	467	804	946
Reduction (%)	33	60	90.58	98	99.87	99.96	99.97	99.99	99.99
BDD_use (ms)	0.11	0.07	0.09	0.13	0.11	0.12	0.10	0.12	0.11

For this testing case the character count was linearly incremented (1-25), the part count was always 3 times the current character count, the min part length was 2 and the max part length was the character count (N).

Results

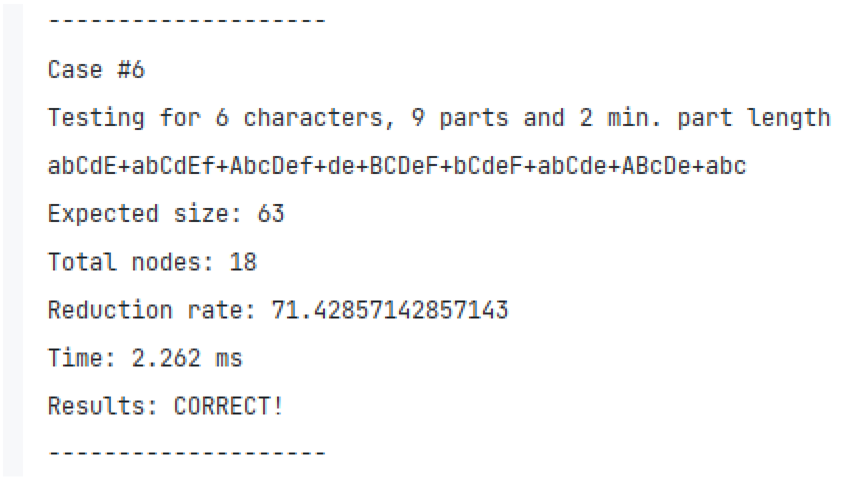
The result is that as the character count increased the BDDcreate time increased almost n^2 but as the character count got larger the time complexity started to look like more of an exponential growth, the BDDuse function stayed almost constant, and the reduction rate increased significantly with the character count.



```
--------------------
Case #23
Testing for 23 characters, 69 parts and 2 min. part length
ONPqRSTUVW+GHijKLmOnpqRSTUv+kLmoNPq+tU+onPQRst+BcdEFgHiJkLmonPQrsTU+eFgHIjkLMoNPqR+uV+fghijKLmoNPQrstu+BCdefGHIjkLmo
both
Expected size: 8388607
Total nodes: 1863
Reduction rate: 99.97779130670921
Creation time: 1927.147 ms
BDDuse time: 0.122 ms
--------------------
Case #24
Testing for 24 characters, 72 parts and 2 min. part length
deFGhiJklMoN+tuVW+ijklMoNPqRSTUVwX+gHIJklMoNpQrSTu+kLmonpqRstUVWx+HiJkLmOnPQrsTuVwx+CdEFGhijklmonPqRSt+CdefghijklmON
Expected size: 16777215
Total nodes: 764
Reduction rate: 99.99544620486773
Creation time: 943.604 ms
BDDuse time: 0.113 ms
--------------------
Case #25
Testing for 25 characters, 75 parts and 2 min. part length
eFGhIJKLMOnpqRsTUv+moNpqrSTuV+AbCdefgHIJK+dEFghI+DeFGHIJkLmOn+ijKlmOnPQRstUv+jkLMoNpqrsTUvWx+BCd+FgHIjKlMOnPqRStuvWX
Expected size: 33554431
Total nodes: 748
Reduction rate: 99.99777078621896
Creation time: 865.254 ms
BDDuse time: 0.11 ms
--------------------
```

Correctness testing

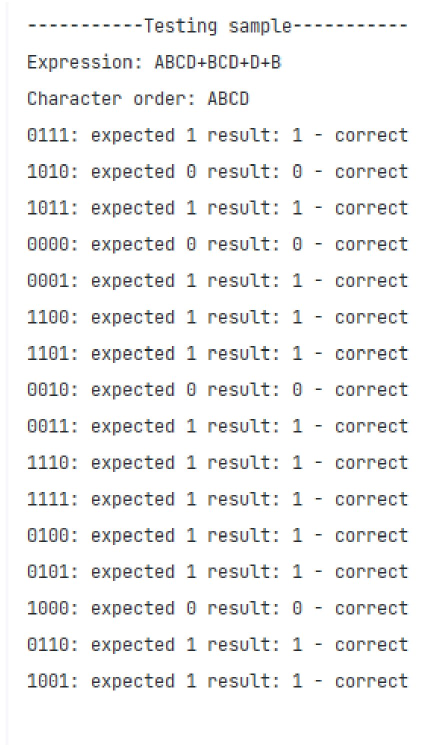
Here we besides testing the performance the correctness of the input is tested using pre generated result vectors. In this case the testing expressions were smaller, but each combination of input for each expression was generated and tested. The testing time wasn't measured, only the BDD_create and reduction rate.



```
--------------------
Case #6
Testing for 6 characters, 9 parts and 2 min. part length
abCdE+abCdEf+AbcDef+de+BCDeF+bCdeF+abCde+ABcDe+abc
Expected size: 63
Total nodes: 18
Reduction rate: 71.42857142857143
Time: 2.262 ms
Results: CORRECT!
--------------------
```

The program automatically compares all results and if all of the results match the diagram is evaluated as current.

Here we can see on this sample how it works:



```
-----------Testing sample-----------
Expression: ABCD+BCD+D+B
Character order: ABCD
0111: expected 1 result: 1 - correct
1010: expected 0 result: 0 - correct
1011: expected 1 result: 1 - correct
0000: expected 0 result: 0 - correct
0001: expected 1 result: 1 - correct
1100: expected 1 result: 1 - correct
1101: expected 1 result: 1 - correct
0010: expected 0 result: 0 - correct
0011: expected 1 result: 1 - correct
1110: expected 1 result: 1 - correct
1111: expected 1 result: 1 - correct
0100: expected 1 result: 1 - correct
0101: expected 1 result: 1 - correct
1000: expected 0 result: 0 - correct
0110: expected 1 result: 1 - correct
1001: expected 1 result: 1 - correct
```

Each result is compared and evaluated, "expected" is what the testing vector expects and the "result" is the result of the BDDuse method.

Results

All testing results were evaluated correct, the maximum number of unique characters tested was 15, maximum part number was 30. Here the relation between the reduction rate and the time/space complexity and character count stayed the same seen in performance testing, although the reduction rate was smaller with smaller inputs.

Test results evaluation

The test results show that the BDDcreate and BDDuse methods functioned correctly. All generated and tested diagrams were correct. The performance testing shows that as the unique character count increased the time complexity increased similarly to n^2, but the reduction rate increased exponentially. The more characters were used the space complexity in relation to the reduction rate started to approach n^2.

Time complexity: $O(n^2)$
Space complexity: $O(n^2)$