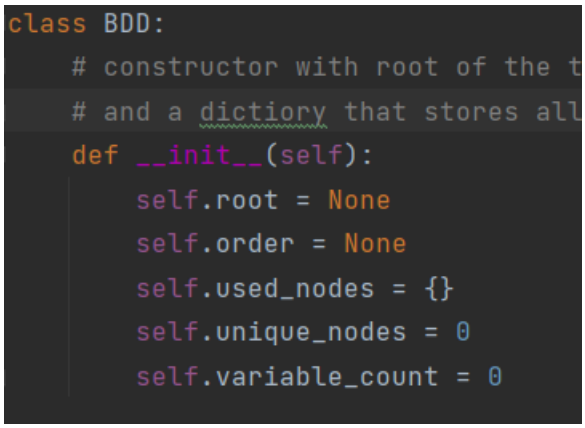
Matúš Rusňák
ID : 116286

Assignment 2

Binary Decision Diagram

In this assignment I have implemented Reduced ordered Binary Decision Diagram – BDD. My BDD will take as an input a string containing a Boolean function and create a tree using Shannon decomposition of the given Boolean function. After creation it will return the root node of the tree. It also contains use function which will return either True or False based on given input. I choose to implement this assignment in python programming language

Tree body



```
class BDD:
    # constructor with root of the t
    # and a dictiory that stores all
    def __init__(self):
        self.root = None
        self.order = None
        self.used_nodes = {}
        self.unique_nodes = 0
        self.variable_count = 0
```

This is my constructor of the BDD tree. It contains root of the tree. Order of Boolean arguments that determine how the tree will be created using Shannon decomposition. A dictionary – used_nodes – where all unique nodes are stored. Number of unique nodes and number of unique Boolean function arguments.

Matúš Rusňák
ID : 116286

Node body

```
class Node:
    def __init__(self, function):
        self.left_child = None
        self.right_child = None
        self.function = function
        self.parent = None
        self.variable_count = 0
```

Every node that is created contains its parent, pointer to left_child that will contain lower node and pointer to right_child that will contain higher node based on Shannon decomposition. Every node also contains number of unique variables in current function. Variable count is later used in function .use() where we traverse the tree.

Tree function: create(„Boolean function“, [order of arguments])

How to use the create function:

```
tree.create("A+B+!CD",['A','B','C','D'])
```

Firstly you need to create a variable that will be an object BDD(). Then you can call the function use and insert arguments. Arguments for Boolean function needs to only contain uppercase alphabet letters and „!“ indicating negation of the Boolean variable. If you want to multiple variables just place them next to each other without any kind of character.

Example: „AB+C+!DE+!F!G“

After you create the correct Boolean function, the second argument is list of the order of inserted Boolean variables for Shannon decomposition.

Example: [‚A‘,‘B‘,‘C‘,‘D‘,‘E‘,‘F‘,‘G‘]

Matúš Rusňák
ID : 116286

Tree function: use(„Boolean variables value“)

How to use the use function

```
tree.use("00010010")
```

Use function will traverse the tree based on the value representing Boolean argument. In every node i stored an integer variable_count which represents the index that that current node takes when using the use function. Since we reduce the tree some variables might become irrelevant so we need to know which Boolean variable represent that current node based on Shannon decomposition. The function will return negative values if input is incorrect. For example not all Boolean variables were given a number or input does not contain only „0“ and „1“. Example of correct input:

Function - [AB+BC+DE], number of unique variables =5, tree.use(„01100“)

Code:

```
def _use(self, current_node, direction):
    if current_node.variable_count==None:
        return current_node.function

    else:
        index = len(direction) - current_node.variable_count
        if direction[index] == "0" and current_node.left_child:
            return self._use(current_node.left_child, direction)
        elif direction[index] == "1" and current_node.right_child:
            return self._use(current_node.right_child, direction)
```

If current direction of the tree is „0“ we traverse the tree to its low child, on the other hand if its „1“ we traverse the tree high child. Once we hit Node(1) or Node(0) we return found value. Results of my use function are tested with my evaluate function.

Matúš Rusňák
ID : 116286

Reduction

Merge isomorphic nodes

In BDD tree we commonly encounter isomorphic nodes. Those nodes take up additional memory so we want to eliminate them. I eliminate them by checking if current created node has been already created. If yes then i don't create current node but point to that already created node in memory. Code:

```
def reduce(self, node):
    if node is None:  # if we pass None do nothing
        return
    if self.used_nodes.get(node.function) is None:  # if
        self.used_nodes[node.function] = node
        return node
    else:  # if node was created, return created node
        used_node = self.used_nodes.get(node.function)
        return used_node
```

In the body of my tree class I store a hash map(used_nodes). Hash map stores all unique nodes created. By checking if given node is in hash map I know if i need to create a node or just point to already created node. This function is always called when new node is being created. Node creation example:

```
current_node.right_child = (self.reduce(Node("+".join(right_child))))
```

Eliminate redundant nodes

When nodes left child and right child are the same, the current node is redundant. The way I eliminate redundant nodes is that I set the nodes parent pointer to node left or right child. This basically removes the redundant node and saves memory.

```
def check_redundant(self, node):
    try:
        if node.left_child == node.right_child:
            return True
        return False
    except:
        return False
```

This function checks if nodes children are equal. If yes it means given node is redundant and returns True.

Matúš Rusňák
ID : 116286

```
if self.check_redundant(current_node) == True:

    if current_node == current_node.parent.left_child:
        current_node.parent.left_child = current_node.left_child
        current_node = current_node.left_child
        self._create(current_node.function, current_node,
                     order[1:])  # recursion to create children for

    else:
        current_node.parent.right_child = current_node.right_child
        current_node = current_node.right_child
        self._create(current_node.function, current_node,
                     order[1:])  # recursion to create children for
```

In my create function I call check_redundant and based on its result I eliminate redundant node. Once a node is reduced i call private create function on current node which is now None to create its children.

When node is redundant, I remove it from my dictionary used_nodes to clear memory.

```
self.used_nodes.pop(current_function) # clearing memory
```

Time complexity

The best case scenario is O(n) where n is the length of Boolean function. This occurs when Boolean function is either a tautology or a contradiction. The worst case scenario is $O(2^m \times n)$ where m is the number of unique variables and n is the length of Boolean function. This occurs when no reduction takes place and the Boolean function is traversed for each node.

Matúš Rusňák
ID : 116286

Testing

For testing correctness of my BDD I implemented my own evaluate function which is located in main.py and demo_main.py. For my time and reduction test I generated 100 random Boolean functions. All data used for testing are stored in /testData. Boolean functions are made from 5,9,13,14 and 15 unique variables. Example: „AD+BC+C+DC+EA“ contains 5 unique variables. Generated functions always contain at least one set of unique variables. For each of 100 i tested every possible use combination ranging from 2^5 possibilities to 2^{15} possibilities. Each time using my own evaluate function to check if created tree return correct use value. All tests passed.

Test results

Num of unique variables *1	one function (seconds) *2	creation time (seconds) *3	use time (seconds) *4	Average reduction (%) *5
5	0.0219066143	0.0000587714	0.0000015629	81.42857142
9	0.0449365925	0.0000856374	0.0000021152	96.00586510
13	10.1895639371	0.0012277522	0.0000030200	99.59280961
14	24.7490310144	0.0014945092	0.0000029472	99.66210706
15	68.3901624218	0.0020699575	0.0000032781	99.85933408

Test table legend

1 – column represents from how many unique variables generated boolean function contains

2 – Test1:column represents average time for creating, using and evaluating one boolean function with all possible use combinations.

3 – Test2: column represents average time for creation of tree with number of unique variables in left column.

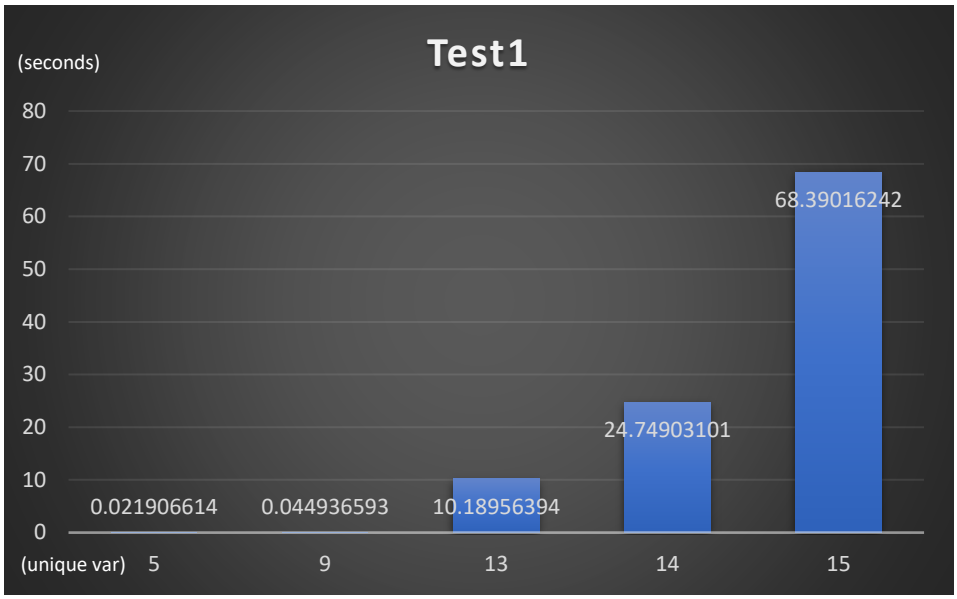
4 – Test3:column represents the average time for using(traversing the tree) to get the result of a boolean function with given use values.

5 – Test4: column represents average reduction of the tree. Reduction = (number of nodes with reduction) / (number of nodes without reduction). I get nodes with reduction by getting the length of dictionary with unique nodes from my tree. And number of nodes without reduction is calculated by $(2^{n+1} -1)$ where n represents number of unique variables in given boolean function

Matúš Rusňák
ID : 116286

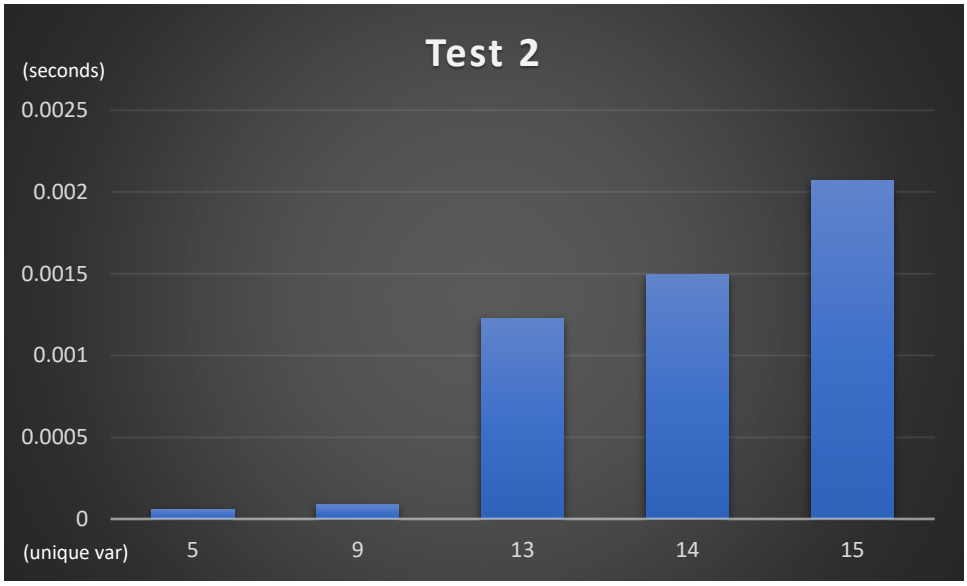
Comparison

Test1 – all combinations



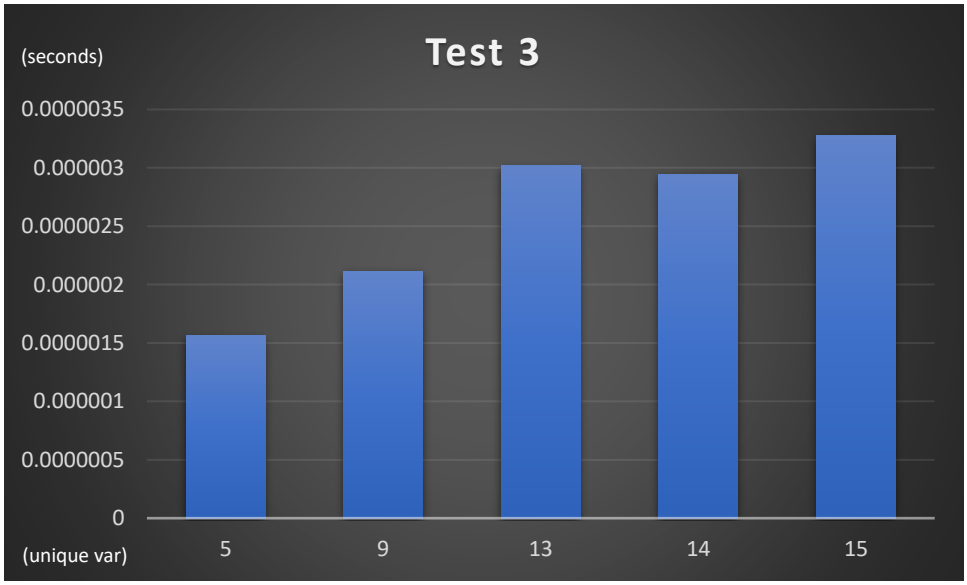
This test results make sense. Since checking the tree for 5 variables is nothing compare to checking the tree for 15 variables which is 2^{15} possible combinations. For this test I am once again reminding that these data take into account use,create and evaluate function.

Test 2 – creation time



This test result again show promising values. As we need more unique variables the tree needs to create more nodes. Then do all the reductions to make it as small as possible. My creation function is called recursively so more and more recursion have to happen in order to create the tree

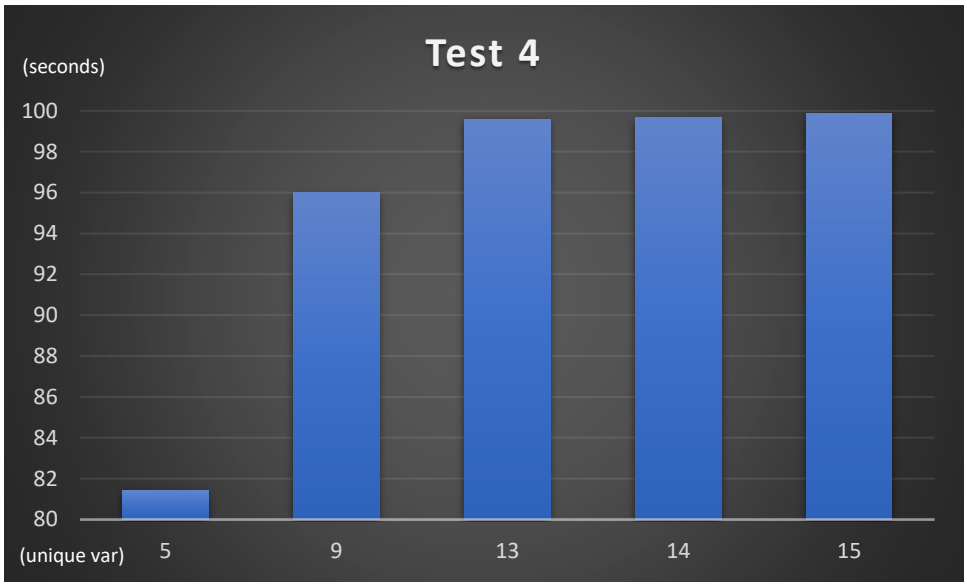
Test 3 – use time



This test result again show promising values .Worst time complexity for the use function is O(n). We have to iterate n times to get return value from the tree. So values should be higher when inserting more unique nodes to the tree.

Matúš Rusňák
ID : 116286

Test 4 - reduction



These results are what we theoretically expect. When more unique variables are given, more nodes are created. Even more already created nodes are created. So by not actually creating them but just setting pointers to them we reduce the number of nodes. For example the last row of not reduced BDD contains 2^n nodes with values 0 and 1. By reducing them we save a lot of memory.

Evaluate function

```python
variable15_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']
def evaluate(function, values):
    result = 0
    variable_values = {}
    for i in range(len(values)):
        variable_values[variable15_list[i]] = values[i]

    function = function.split("+")
    for i in range(len(function)):
        temporary_result = 1
        for j in range(len(function[i])):
            if function[i][j] == "!":
                continue
            else:
                if function[i][j - 1] == "!":
                    current_value = variable_values.get(function[i][j])
                    if current_value == "1":
                        current_value = 0
                    else:
                        current_value = 1
                    temporary_result = temporary_result * current_value
                else:
                    temporary_result = temporary_result * int(variable_values.get(function[i][j]))
        result = result + temporary_result

    if result >= 1:
        return True
    else:
        return False
```

This the function evaluate that I used in every test to check if my created tree returned correct values

Matúš Rusňák
ID : 116286

Conclusion

Almost all test that were conducted showed results that were theoretically correct. The reduction works as it should and average times are also looking promising. I got my test result from code located in main.py All tests ran for about 2 hours. Which is why I created demo test called demo_main.py . Running this program will test all those tests above with 5 and 9 unique variables. S