Data structures and algorithms

2 assignment – Binary Decision Diagrams

Author: Marek Čederle
Instructor: MSc. Mirwais Ahmadzai
(Monday 11:00)

Theory

Introduction

Binary Decision Diagrams (BDDs) are another important data structure that is widely used in computer science and engineering for various applications, including hardware verification, symbolic model checking, and Boolean satisfiability. BDDs are used to represent Boolean functions and can efficiently handle problems involving large sets of Boolean variables.

The concept of BDDs, including their construction and operations such as variable ordering, reduction, and manipulation, will be explored in this documentation. By the end of this documentation, you will obtain a comprehensive understanding of the BDD data structure.

BDD (Binary Decision Diagram)

Binary Decision Diagrams (BDDs) are a data structure used to represent Boolean functions in a compact and efficient way. They are widely used in computer science and engineering for various applications, including hardware verification, symbolic model checking, and Boolean satisfiability.

A BDD is essentially a directed acyclic graph (DAG) in which each node represents a Boolean variable, and each edge represents the Boolean value (0 or 1) of the variable. The nodes in a BDD are usually arranged in layers, where each layer corresponds to a different variable. The root node of the BDD represents the top-level Boolean function, while the terminal nodes represent the constant Boolean values of 0 and 1.

Shannon decomposition can be used to recursively construct Binary Decision Diagrams (BDDs) by applying a variable ordering, which is a linear ordering of the variables in the Boolean function. This ordering can be decomposed into a binary tree structure, with each internal node representing a Boolean function of two variables, and each leaf node representing a constant Boolean value. Different variable orderings can result in different binary tree structures, which can affect the size and shape of the resulting BDD for the same Boolean function.

BDDs provide several benefits, including compactness, fast evaluation, and efficient manipulation. Since each variable appears only once in the BDD, it can represent very large Boolean functions using a small number of nodes. Furthermore, BDDs can perform Boolean operations, such as conjunction, disjunction, and negation, in time linear in the size of the BDD.

However, BDDs have some limitations, such as exponential worst-case complexity for certain operations and sensitivity to variable ordering. They are also limited in their ability to represent some types of functions, such as those with high-degree polynomials or complex arithmetic operations.

Shannon decomposition

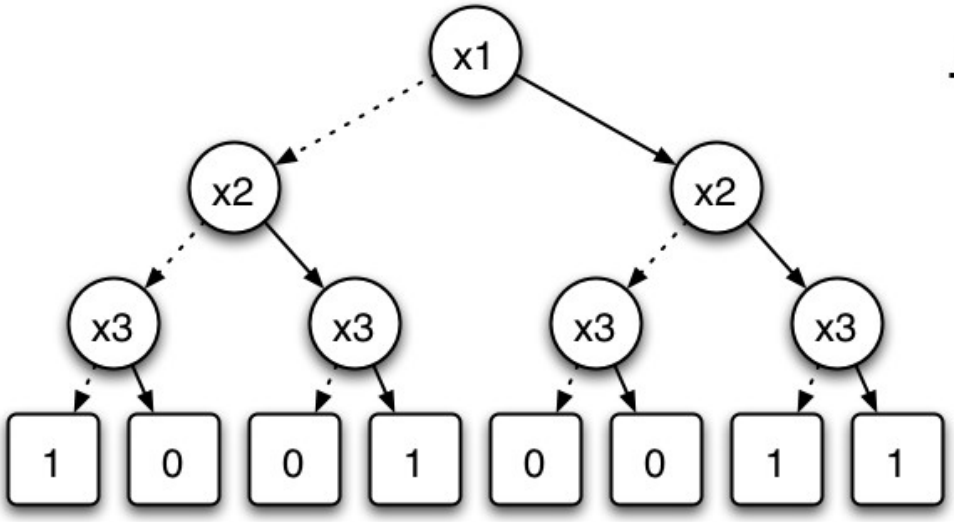
Shannon decomposition is a method used in Boolean function analysis and circuit design that breaks down a given Boolean function into smaller, simpler functions. This decomposition can help simplify complex Boolean expressions and reduce the number of gates required to implement a circuit. The method was developed by Claude Shannon, who is considered one of the fathers of modern digital circuit design.

The Shannon decomposition is based on the concept of variable splitting. In this method, a Boolean function is recursively split into two smaller functions based on a selected variable. This variable is called the "split variable" and is chosen based on certain criteria, such as minimizing the number of gates required to implement the circuit.

The Shannon decomposition involves two steps: variable splitting and recursive decomposition.

- Variable Splitting: The first step in Shannon decomposition involves selecting a split variable. The split variable is chosen from the set of input variables to the Boolean function. The split variable divides the function into two sub-functions, based on the variable's value.
- Recursive Decomposition: The second step in Shannon decomposition is to apply the same process recursively to each sub-function generated in the previous step. This recursive decomposition continues until each sub-function becomes a constant or a variable.

The result of the Shannon decomposition is a tree-like structure called a Binary Decision Diagram (BDD). In this diagram, the nodes represent the split variables, and the edges represent the value of the variable. The leaves of the BDD represent the final output of the Boolean function.



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

My implementation

To implement these data structures in my project I these programs:

- IDE (integrated development environment) – VSCode (Visual Studio Code) version 1.78.1
- Compiler – g++.exe version 8.3.0
- Memory checker – valgrind.exe version 3.18.1 (my code is without memory leaks)
- Graphical representation of measured data - Google Sheets
- Documentation – Microsoft Word

In the next section I will describe only the crucial functions and classes. Other ones will be somewhat explained in the code or not if they are just basic functions.

Node structure

This code defines a struct called "Node", which is used to represent nodes in a binary decision diagram. The struct contains four data members:

- "data", which is of type char, represents the data stored in the node.
- "bfunction", which is of type string, represents the Boolean function associated with the node.
- "l", which is a pointer to the left child of the node.
- "r", which is a pointer to the right child of the node.

The first constructor initializes the data member variables with default values. The second constructor initializes the "data" member variable with the passed parameter value, and the "l" and "r" pointers are initialized to nullptr.

```
struct Node
{
    char data;
    string bfunction;
    Node *l;
    Node *r;

    Node()
    {
        this->data = '\0';
        l = nullptr;
        r = nullptr;
    }

    Node(char data)
    {
        this->data = data;
        l = nullptr;
        r = nullptr;
    }
};
```

BDD functions

BDD class and important members

This is a class definition for a Binary Decision Diagram (BDD). It contains several data members and member functions:

- Node *root: A pointer to the root node of the BDD.
- int var_count: The number of variables in the Boolean function that the BDD represents.
- size_t size: The number of nodes in the BDD. Initialized to 2, because there are always two nodes in a BDD: 0 and 1.
- string poradie: The variable ordering used to construct the BDD.
- Node *false_node: A pointer to the node representing the constant 0.
- Node *true_node: A pointer to the node representing the constant 1.
- std::unordered_map<string, Node *> function_map: A map that stores the BDDs for each Boolean function represented as a string.
- std::unordered_map<Node *, char> printedNodes: A map that stores the nodes that have been printed when the BDD is displayed.
- std::unordered_map<Node *, char> deletedNodes: A map that stores the nodes that have been deleted when the BDD is destroyed.

The class also contains member functions to create, display, and destroy BDDs.

```
class BDD

{
public:
    Node *root = nullptr;
    int var_count;   // get value in bdd create
    size_t size = 2; // number of nodes in BDD 2 because of 0 and 1 nodes
    string poradie;

    Node *false_node = new Node('0');
    Node *true_node = new Node('1');

    // Map to store the function and root pointer
    std::unordered_map<string, Node *> function_map;
    std::unordered_map<Node *, char> printedNodes;
    std::unordered_map<Node *, char> deletedNodes;
.
.
.
};
```

BDD create

This function creates a Binary Decision Diagram (BDD) based on the Boolean function represented by a string bfunkcia and a given variable ordering represented by another string poradie. The BDD is created by calling the recursive_create() function with the Boolean function and the current variable index initially set to 0. After the BDD is created, the size of the BDD is computed and stored in the size attribute of the BDD class.

If the root of the BDD is a constant node, either '0' or '1', then the size attribute is set to 1 because there is only one node in the BDD. Finally, the function returns a pointer to the BDD.

```
BDD *BDD_create(string bfunkcia, string poradie)
{
    BDD *bdd = new BDD();
```

```
    bdd->poradie = poradie;
    bdd->var_count = poradie.length();
    bdd->root = bdd->recursive_create(bfunkcia, 0);
    if (bdd->root->data == '1' || bdd->root->data == '0') // if it is
tautology or contradiction
    {
        bdd->size = 1;
    }
    return bdd;
}
```

BDD create with best order

The BDD_create_with_best_order function tries to create a Binary Decision Diagram (BDD) for a given Boolean function bfunkcia with the best variable ordering. It does so by trying n^2 permutations of the variables count in the Boolean function and then creating a BDD for each permutation using the BDD_create function. It then selects the BDD with the smallest number of nodes as the BDD with the best ordering and returns it.

If the size of the BDD created with the current variable ordering is smaller than the size of the best BDD found so far, the current BDD is selected as the new best BDD.

```
BDD *BDD_create_with_best_order(string bfunkcia)
{
    // finding the best order
    // using n^2 permutations

    srand(10);
    string poradie = getPoradie(bfunkcia);

    /*uncomment 1*/
    // cout << "getPoradie: " << poradie << endl;

    BDD *best_bdd = nullptr;

    for (size_t i = 0; i < poradie.length() * poradie.length(); i++)
    {
        string temp_poradie = poradie;

        std::random_shuffle(temp_poradie.begin(), temp_poradie.end());

        BDD *temp_bdd = BDD_create(bfunkcia, temp_poradie);
        size_t best_bdd_size = best_bdd == nullptr ? 999999999 : best_bdd-
>size;
        // cout << "bfunkcia: " << bfunkcia << endl;
        // cout << "poradie: " << temp_poradie << endl;
        // cout << "size: " << temp_bdd->size << endl;
        if (temp_bdd->size < best_bdd_size)
        {
            // cout << "-------------------------" << endl;
            // cout << "MAM LEPSIE PORADIE" << endl;
            // cout << "-------------------------" << endl;
            delete best_bdd;
            best_bdd = temp_bdd;
            continue;
        }
        delete temp_bdd;
    }
```

```
    /*uncomment*/
    // cout << "--------------------------" << endl;
    // cout << "BEST" << endl;
    // cout << "best poradie: " << best_bdd->poradie << endl;

    return best_bdd;
}
```

BDD use

This is a function for using a BDD (Binary Decision Diagram) to evaluate a given input. The function takes in a BDD pointer and a string of inputs (0s and 1s) as input parameters.

The function first checks whether the input size matches the variable count of the BDD, and if not, it returns -1 indicating an incorrect input size.

The function then initializes the current node to be the root of the BDD and iterates through each character of the input string. If the current node is already a terminal node (i.e., its data value is 0 or 1), the function immediately returns the data value of the current node. Otherwise, the function checks whether the current node corresponds to the current input variable (based on the variable order), and traverses to the corresponding child node based on the input value (0 or 1). If the input value is neither 0 nor 1, the function returns -1 indicating an incorrect input value.

Finally, the function returns the data value of the last node reached in the BDD traversal, which represents the output of the BDD for the given input.

```
char BDD_use(BDD *bdd, string vstupy)
{
    if (vstupy.length() != (size_t)bdd->var_count)
    {
        // Incorrect input size
        return -1;
    }

    Node *current_node = bdd->root;

    for (size_t i = 0; i < (size_t)vstupy.length(); i++)
    {

        if (current_node->data == '1' || current_node->data == '0')
        {
            return current_node->data;
        }

        if (current_node->data != bdd->poradie[i])
        {
            continue;
        }

        if (vstupy[i] == '0')
        {
            current_node = current_node->l;
        }
        else if (vstupy[i] == '1')
        {
            current_node = current_node->r;
        }
        else
```

```
        {
            // Incorrect input value
            return -1;
        }
    }

    return current_node->data;
}
```

Other functions

Shannon expansion

This code defines a function doShannonExpansion that takes a boolean function in the form of a string bfunkcia and a variable var as inputs and returns the two Shannon expansions of the function with respect to the variable in the form of strings *left and *right.

The function loops over the characters in the input string bfunkcia, and determines whether each character corresponds to the variable var, a negation of the variable, a sum symbol or the end of the string.

When encountering the variable or its negation, the function determines whether it appears on the left or right side of the expansion based on whether the previous variable was on the left or right. If the variable does not appear in the expression, then the part of the expression so far is pushed to both sides. When encountering the sum symbol, the part of the expression so far is pushed to the current side.

Once the function has processed the input expression, it sorts each side by length and concatenates the terms with a sum symbol between them to form the Shannon expansions. If a side is empty, it is set to "0", and if a side contains only a single term, it is set to that term.

```
void doShannonExpansion(const string bfunkcia, const char var, string
*left, string *right)
    {
        string part = "";
        vector<string> left_side;
        vector<string> right_side;
        vector<string> *current_side = nullptr; // Pointer to the side that
is currently being filled

        for (size_t i = 0; i <= bfunkcia.length(); i++)
        {
            const char c = bfunkcia[i];
            if (c == tolower(var)) // bolo by aj tak lower case
            {
                // verify (aA) case and it can be deleted because it is a
contradiction
                if (current_side == &left_side)
                {
                    // skip the part
                    part = "";
                    current_side = nullptr;
                    while (bfunkcia[i] != '+' && bfunkcia[i] != '\0')
                    {
                        i++;
                    }
                    continue;
```

```
                }
                current_side = &right_side;
                continue;
            }

            if (c == toupper(var))
            {
                // to iste ako vyssie
                if (current_side == &right_side)
                {
                    // skip the part
                    part = "";
                    current_side = nullptr;
                    while (bfunkcia[i] != '+' && bfunkcia[i] != '\0')
                    {
                        i++;
                    }
                    continue;
                }

                current_side = &left_side;
                continue;
            }

            if (c == '+' || c == '\0')
            {
                // If the variables was not found in the expression, push
part to both sides
                if (current_side == nullptr)
                {
                    pushToSide(&left_side, part);
                    pushToSide(&right_side, part);
                    part = "";
                    current_side = nullptr;
                    continue;
                }
                // Otherwise push to the current side
                pushToSide(current_side, part);
                part = "";
                current_side = nullptr;
                continue;
            }

            part += c;
        }

        *left = "";
        *right = "";

        //-----------left side----------------

        // If the side is empty, set it to 0
        if (left_side.empty())
        {
            *left = "0";
        }
        // If the side is 1 or 0, set it to 1 or 0
        else if (left_side.at(0) == "1" || left_side.at(0) == "0") //
left_side[0]
        {
            *left = left_side.at(0);
```

```
        }
        else
        {
            // Sort the side by length and concatenate it (lambda function
for sorting)
            std::sort(left_side.begin(), left_side.end(), [](const string
&a, const string &b)
                      { return a.length() < b.length(); });

            for (size_t i = 0; i < left_side.size(); i++)
            {
                (*left).append(left_side.at(i));
                if (i != left_side.size() - 1)
                {
                    (*left).append("+");
                }
            }
        }

        //-----------right side----------------

        // If the side is empty, set it to 0
        if (right_side.empty())
        {
            *right = "0";
        }
        // If the side is 1 or 0, set it to 1 or 0
        else if (right_side.at(0) == "1" || right_side.at(0) == "0")
        {
            *right = right_side.at(0);
        }
        else
        {
            // Sort the side by length and concatenate it
            std::sort(right_side.begin(), right_side.end(), [](const string
&a, const string &b)
                      { return a.length() < b.length(); });
            for (size_t i = 0; i < right_side.size(); i++)
            {
                (*right).append(right_side.at(i));
                if (i != right_side.size() - 1)
                {
                    (*right).append("+");
                }
            }
        }
    }
```

Recursive create

The input parameters are a Boolean function represented as a string (bfunkcia) and an index indicating the order of variables (orderIndex). The function returns a pointer to the root node of the binary tree.

The function first checks if the function is a constant function (0 or 1) and returns the corresponding node if it is. Then, it checks if the function has already been created and stored in the function_map map, which maps a string key (composed of the function and the variable) to a pointer to the corresponding node. If the node exists, it returns it.

If the function is not a constant and has not been created, the function expands it using Shannon expansion with respect to the variable corresponding to the current orderIndex. This creates two new functions, which are recursively passed to create the left and right subtrees of the current node.

If the left and right subtrees are identical, the function only creates one node for both subtrees, since they represent the same function. If the left and right subtrees are not identical, the function creates a new node with the current variable as the data and the original function as the bfunction. The node is stored in the function_map for later use.

If the left and right subtrees are the same node, the function returns one of the nodes and deletes the other one. This ensures that the binary tree is reduced and does not have any duplicate nodes.

Finally, the function returns the root node of the binary tree.

```
Node *recursive_create(string bfunkcia, const size_t orderIndex)
{

    // terminate end nodes
    if (bfunkcia == "0")
    {
        return false_node;
    }
    if (bfunkcia == "1")
    {
        return true_node;
    }

    const char var = poradie[orderIndex];
    const string key = bfunkcia + "|" + var; // basically a hash

    // Check if the function is in the function_map
    if (function_map.find(key) != function_map.end())
    {
        return function_map[key];
    }

    // Expand
    string bfunkcia_0 = "";
    string bfunkcia_1 = "";
    doShannonExpansion(bfunkcia, var, &bfunkcia_0, &bfunkcia_1);
    // cout << "bfunkcia: " << bfunkcia << endl;
    // cout << "var: " << var << endl;
    // cout << "bfunkcia_0: " << bfunkcia_0 << endl;
    // cout << "bfunkcia_1: " << bfunkcia_1 << endl;
    // cout << "======================" << endl;
    // If the functions are the same, create only one root in it's place
    if (bfunkcia_0 == bfunkcia_1)
    {
        return recursive_create(bfunkcia_0, orderIndex + 1);
    }

    // Create the node
    Node *node = new Node();
    node->data = var;
    node->bfunction = bfunkcia;
    size++;
    function_map[key] = node;
    node->r = recursive_create(bfunkcia_1, orderIndex + 1);
    node->l = recursive_create(bfunkcia_0, orderIndex + 1);
```

```
    // If the nodes are the same, return one of them
    if (node->r == node->l)
    {
        function_map[key] = node->r;
        delete node;

        size--;
        return function_map[key];
    }

    return node;
}
```

Testing functions

Evaluate

This function takes in a boolean function bfunkcia, an input string input, and a variable order string poradie. It evaluates the boolean function based on the input and returns the result as a character ('0' or '1'). If the input string is not the same length as the variable order string or the input contains characters not found in the variable order string, it returns -1.The function iterates through each clause and evaluates if it is true based on the input. If any clause is true, it returns '1'. If none are true, it returns '0'.

```
char evaluate(string bfunkcia, string input, string poradie)
{
    string binary_bfunction;
    int index;

    if (input.length() != poradie.length())
    {
        cout << "Wrong input" << endl;
        return -1;
    }
    // make binary bfunction based on input i.e. bfunction is a+b+c, input
is 111 and poradie is abc then binary_bfunction will be 1+1+1

    bool isClauseTrue = false;
    for (size_t i = 0; i <= bfunkcia.length(); i++)
    {
        char c = bfunkcia[i];

        if (c == '+' || i == bfunkcia.length())
        {
            if (isClauseTrue)
                return '1';
            continue;
        }

        index = poradie.find(tolower(c));

        if (index == -1)
        {
            cout << "Wrong input" << endl;
            return -1;
        }

        char inputChar = input[index];
```

```
        if ((islower(c) && inputChar == '0') || (isupper(c) && inputChar ==
'1'))
        {
            isClauseTrue = false;
            while (bfunkcia[i] != '+' && ++i != bfunkcia.length())
                ;

            continue;
        }
        isClauseTrue = true;
    }

    return '0';
}
```

BDD test

The "testBDD" function takes a BDD (binary decision diagram), a Boolean function string, and a string representing the order of the variables in the function as inputs. It then generates all possible input combinations based on the number of variables in the function, evaluates the Boolean function for each input combination, and compares the result with the corresponding evaluation from the BDD. If the BDD evaluation does not match the expected evaluation, it outputs an error message and returns false. Otherwise, it returns true.

```
bool testBDD(BDD *bdd, string bfunkcia, string poradie)
{

    // int maxInputov = 1 << poradie.length();
    // 2^n inputs because of n variables and it is doing truth table
    int maxInputov = pow(2, poradie.length());

    for (int i = 0; i < maxInputov; i++)
    {
        string input = std::bitset<64>(i).to_string();
        input = input.substr(input.size() - poradie.length());
        char eval = evaluate(bfunkcia, input, poradie);
        char bddEval = BDD_use(bdd, input);
        if (eval != bddEval)
        {
            cout << " BDD WRONG EVALUATION" << endl;
            cout << "Failed on input: " << input << "| GOT: " << bddEval <<
" | EXPECTED: " << eval << endl;
            return false;
        }
    }
    return true;
}
```

Dataset generator

This program generates dataset to file “expressions-DNF.txt” with unique expressions in my DNF format. In my format negation is represented by uppercase character and non-negated variable is represented by lowercase character. The function first seeds the random number generator using the current time multiplied by n and a random value generated by rand(). It generates two strings of length n, one containing lowercase letters and one containing uppercase letters. These two strings are then concatenated to create an alphabet string of size 2*n. This alphabet string is then duplicated

n times. The string is shuffled randomly using random_shuffle. The string is then split into parts. Each part is of random size and it is checked for duplicate variables. If any duplicate variables are found, they are removed. The parts are then concatenated using the Boolean OR operator "+" to create the final Boolean function. The testing program then draws input data from this file.

Testing

In my main function, I'm creating a file called "expressions-DNF.txt", where I'm generating Boolean functions with 11 to 15 variables, and writing them to the file. Then, I'm opening the file and reading each line. For each line, I'm extracting the poradie and bfunkcia values, and using them to create a BDD object with BDD create function as well as BDD create with best order function. I'm then calling the testBDD function to test the BDD's evaluation. Finally, I'm calculating some statistics such as the average time complexity, average space complexity, and average reduced ratio of the BDD, and printing them to the console.

It is important to note that the specific tests I ran and the results I obtained depended on the input files I created and the hardware on which I ran my tests.

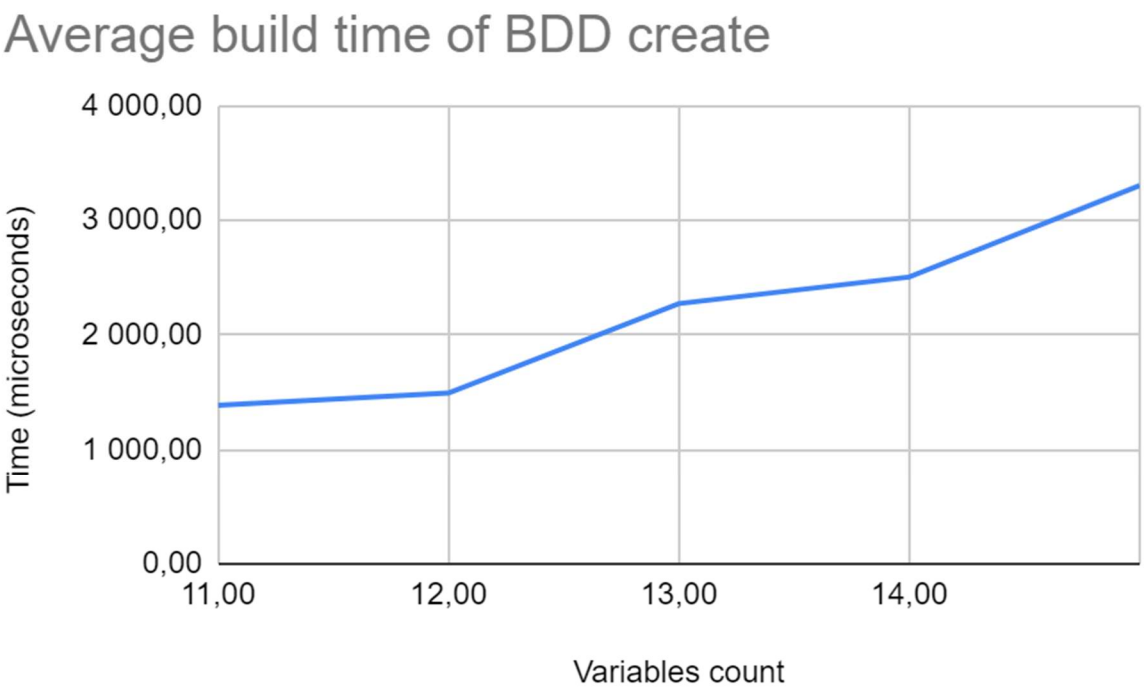
Results of the tests are shown in Comparison section.

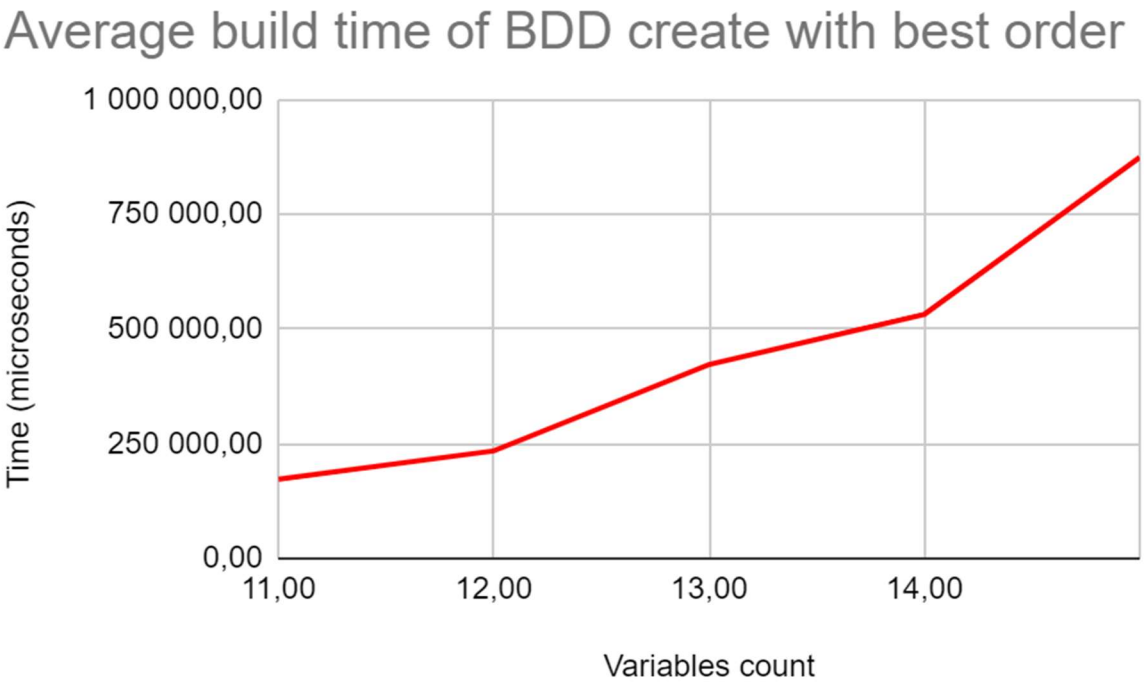
Comparison

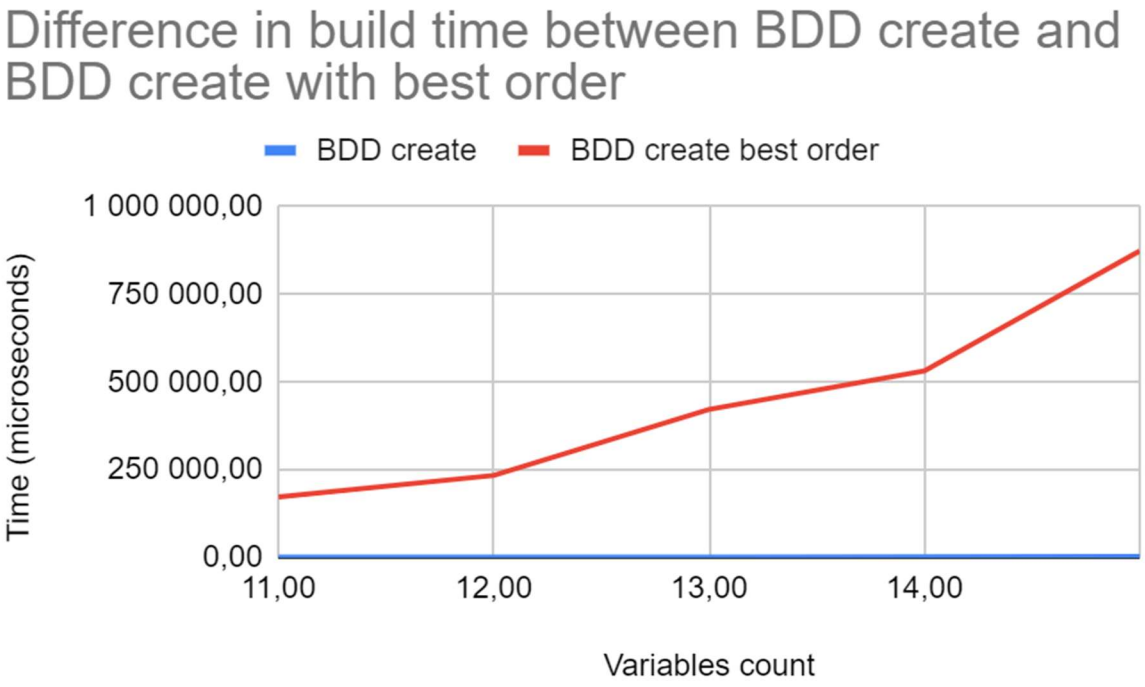
As you can see from the graphs BDD create has much lower time complexity compared to BDD best order. The time complexity of BDD best order is exponential. Decomposition in worst case have O((2^(n+1))-1) space complexity.

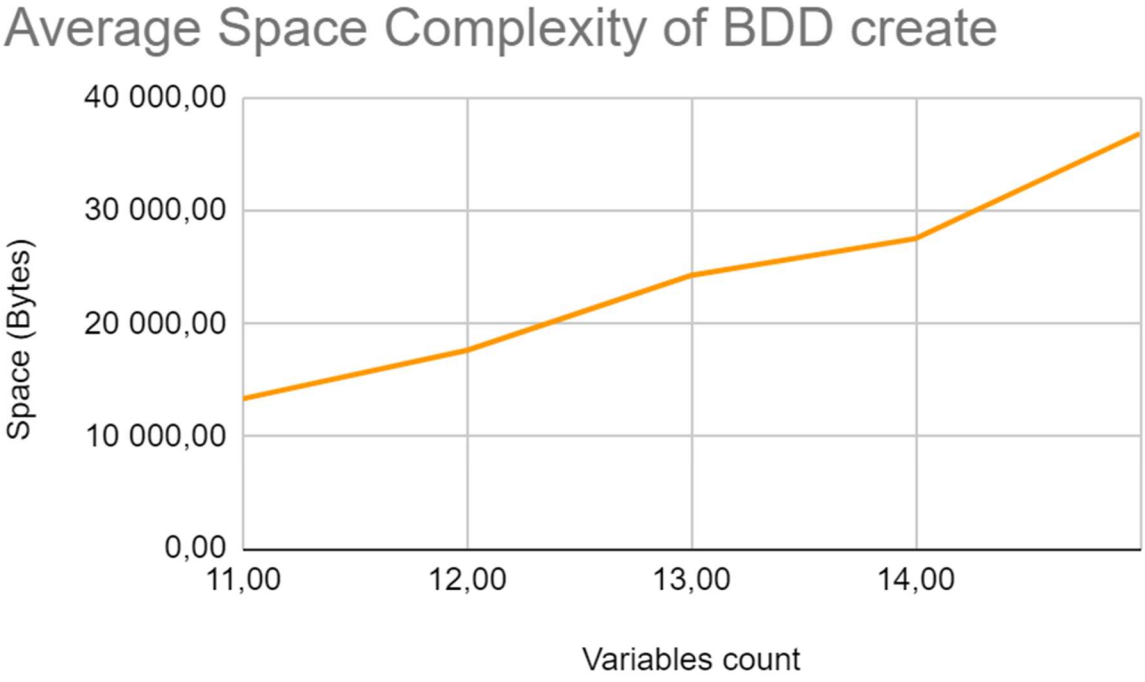
It can be noted that BDD consistently gets outperformed by BDD best order in terms of the space complexity and reduction rate, but is much better in time complexity than BDD best order. For instance, BDD best order was found to take 125 times, 158 times, and 186 times longer than BDD to create BDDs for variables 11, 12, and 13, respectively.

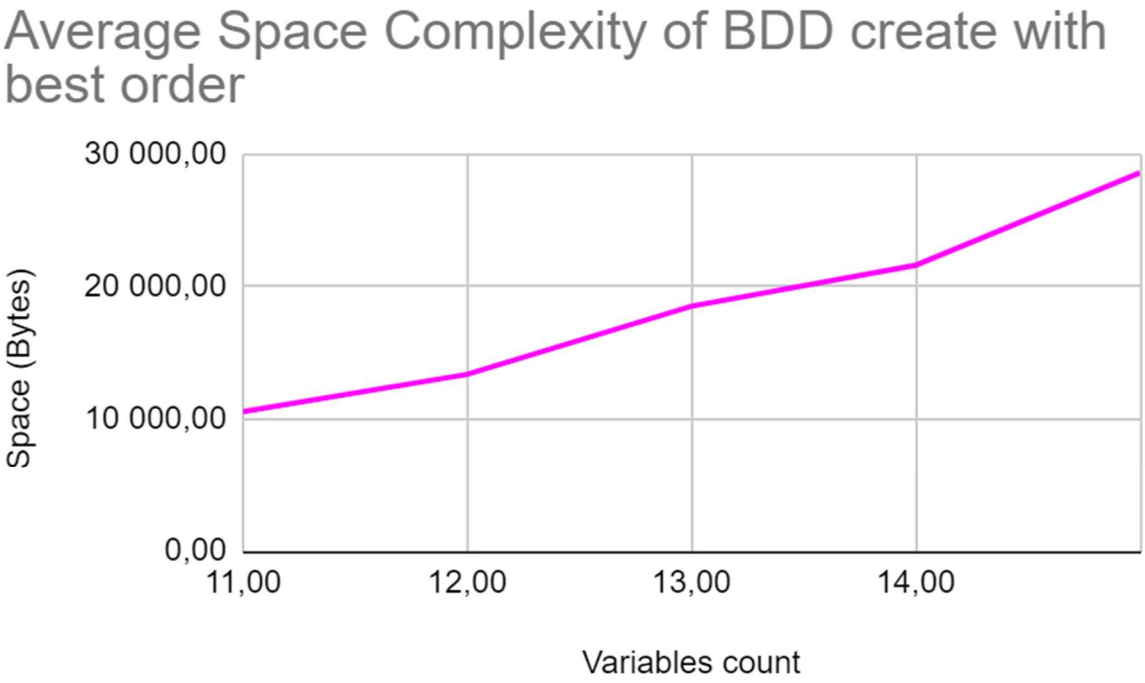
Here are graphs and table comparing BDD with BDD created with “best order”:

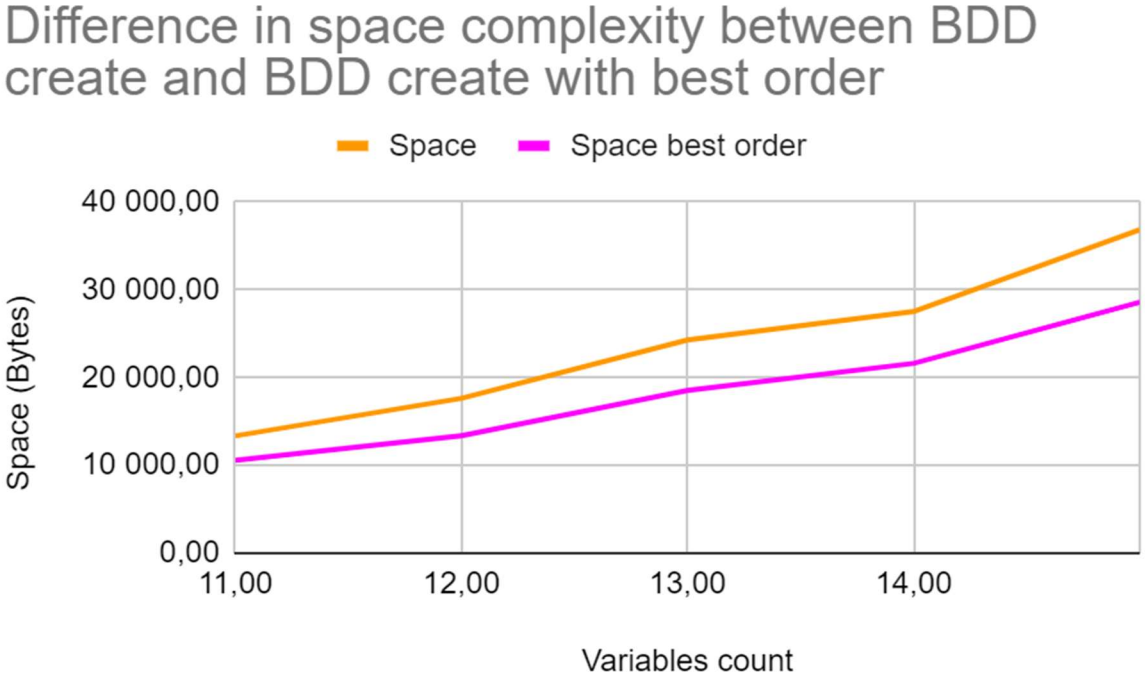

Average build time of BDD create
4 000,00
3 000,00
2 000,00
1 000,00
0,00
Time (microseconds)
11,00
12,00
13,00
14,00
Variables count

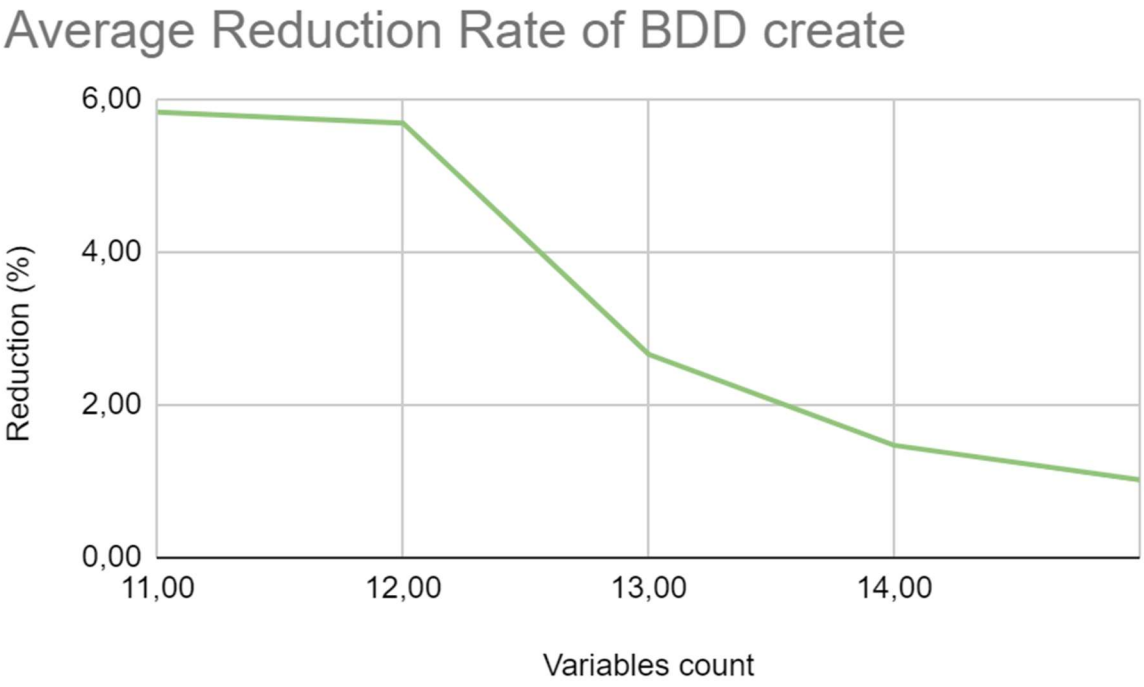

Average build time of BDD create with best order
1 000 000,00
750 000,00
500 000,00
250 000,00
0,00
Time (microseconds)
11,00
12,00
13,00
14,00
Variables count

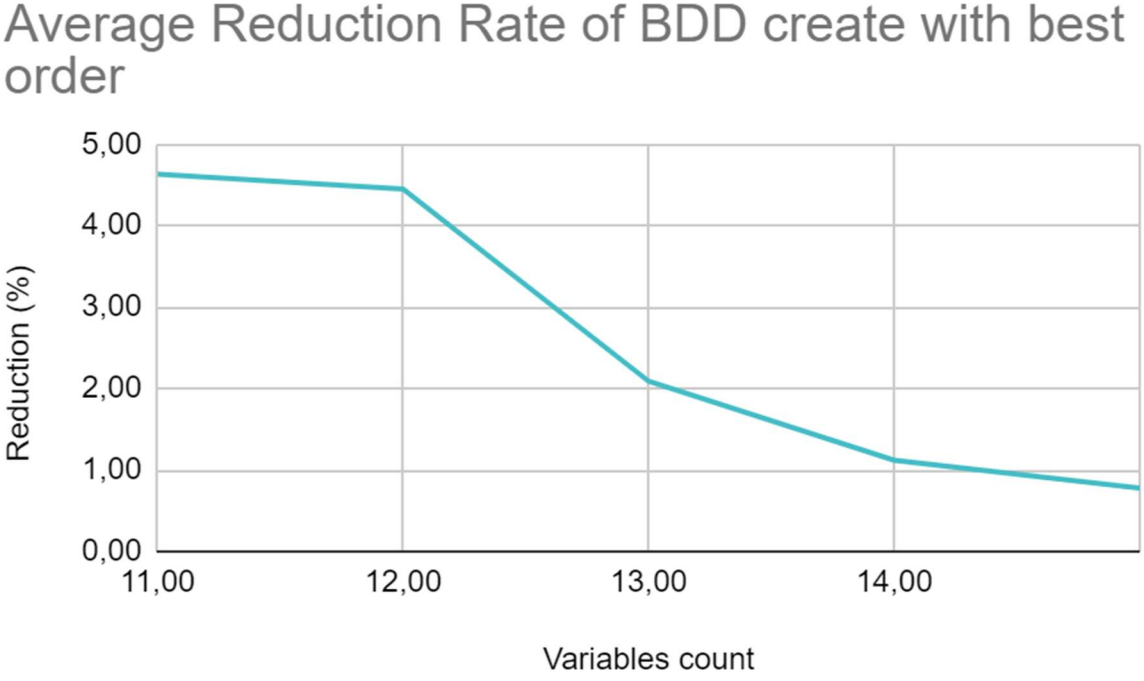

Difference in build time between BDD create and BDD create with best order
BDD create
BDD create best order
1 000 000,00
750 000,00
500 000,00
250 000,00
0,00
Time (microseconds)
11,00
12,00
13,00
14,00
Variables count

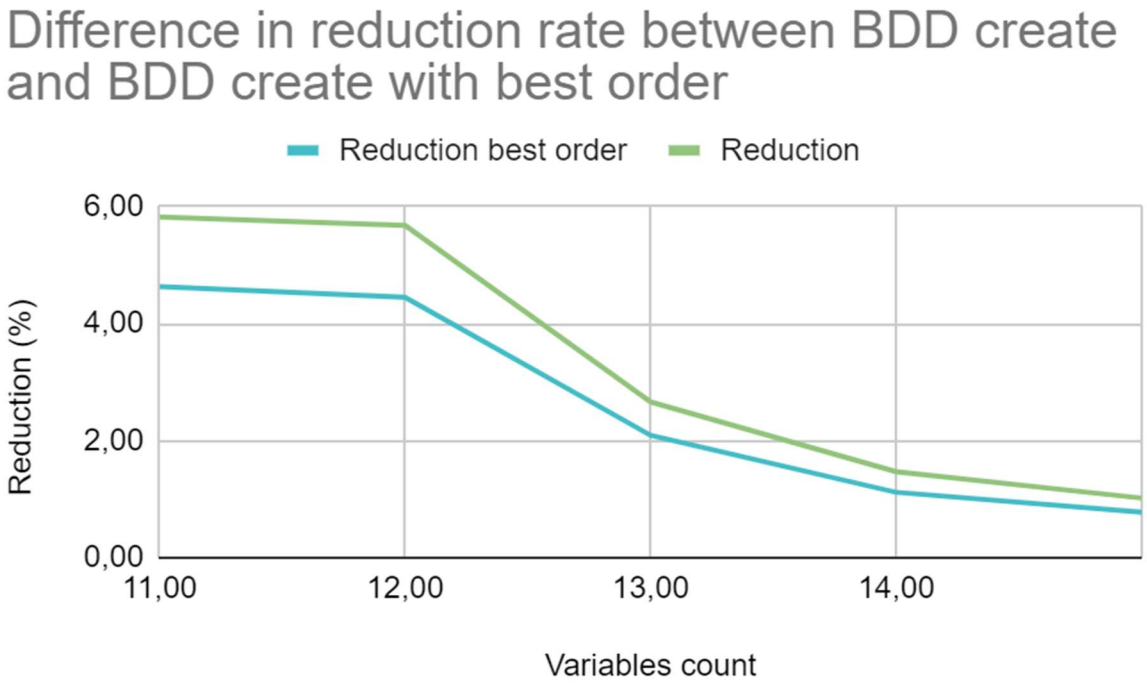

Average Space Complexity of BDD create
40 000,00
30 000,00
20 000,00
10 000,00
0,00
Space (Bytes)
11,00
12,00
13,00
14,00
Variables count


Average Space Complexity of BDD create with best order
30 000,00
20 000,00
10 000,00
0,00
Space (Bytes)
11,00
12,00
13,00
14,00
Variables count


Difference in space complexity between BDD create and BDD create with best order
Space
Space best order
40 000,00
30 000,00
20 000,00
10 000,00
0,00
Space (Bytes)
11,00
12,00
13,00
14,00
Variables count


Average Reduction Rate of BDD create
6,00
4,00
2,00
0,00
Reduction (%)
11,00
12,00
13,00
14,00
Variables count


Average Reduction Rate of BDD create with best order
5,00
4,00
3,00
2,00
1,00
0,00
Reduction (%)
11,00
12,00
13,00
14,00
Variables count



	BDD (average times in microseconds, space in Bytes, reduction in %) based on 100 runs					
Variables	BDD create	BDD create best order	Space	Space best order	Reduction	Reduction best order
11	1 387,00	172 994,00	13 356,00	10 563,00	5,82	4,63
12	1 493,00	234 401,00	17 638,00	13 381,00	5,68	4,45
13	2 274,00	422 564,00	24 286,00	18 525,00	2,66	2,09
14	2 505,00	531 840,00	27 527,00	21 615,00	1,47	1,12
15	3 303,00	873 559,00	36 824,00	28 584,00	1,02	0,78

Conclusion

In summary, Binary Decision Diagrams (BDDs) are a data structure used to represent Boolean functions in a compact and efficient way. They are widely used in computer science and engineering for various applications, including hardware verification, symbolic model checking, and Boolean satisfiability. Shannon decomposition can be used to recursively construct BDDs by applying a variable ordering, which is a linear ordering of the variables in the Boolean function. This ordering can be decomposed into a binary tree structure, with each internal node representing a Boolean function of two variables, and each leaf node representing a constant Boolean value. Different variable orderings can result in different binary tree structures, which can affect the size and shape of the resulting BDD for the same Boolean function.

References

Presentations from DSA lectures

https://en.wikipedia.org/wiki/Boole%27s_expansion_theorem

https://en.wikipedia.org/wiki/Binary_decision_diagram

https://www.slideshare.net/RajeshYadav49/reduced-ordered-binary-decision-diagram-devi

https://www.youtube.com/watch?v=_C9yLZ2BGBA