FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
SLOVENSKÁ TECHNICKÁ UNIVERZITA
Ilkovičova 2, 842 16 Bratislava 4

2021/2022
Data structures and algorithms
Second assignment

Cvičiaci: MSc. Mirwais Ahmadzai
Čas cvičení: Štvrtok 11:00 – 12:50

Vypracoval: Peter Bartoš
AIS ID: 116143

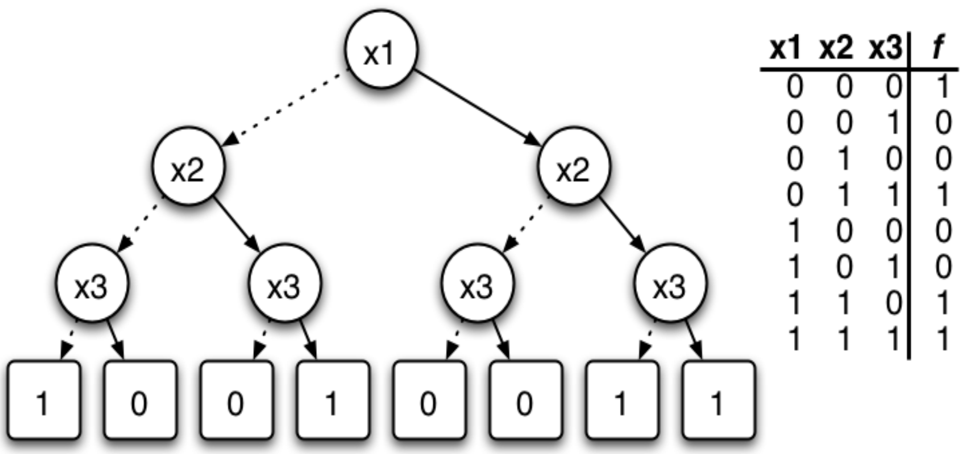
Contents

Binary Decision Diagram

1. Introduction

BDD is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression.

A Boolean function can be represented as a rooted, directed, acyclic graph, which consists of several (decision) nodes and two terminal nodes. The two terminal nodes are labelled 0 (FALSE) and 1 (TRUE). (Wikipedia)



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

2. Own implementation – Reduced BDD

This assignment will be all about implementing a reduced BDD that is used to represent a Boolean function in DNF form. Implementation should always be at least a reduced type of BDD when representing a Boolean function. Boolean function grows exponentially with the number of variables 2^n. That is why it needs to be reduced. For 13 variables there would be 8192 nodes. This is very memory and time heavy and reduction helps with that tremendously.

Firstly, foundation of BDD is very tree alike and almost looks like binary tree. In this implementation 2 structures were defined. One structure represents root and the other represents nodes.

```
typedef struct bdd {
 int numberOfNodes;
 int numberOfVariables;
 char variable;
 char* bfunction;
 struct node* right;
 struct node* left;
 struct node* next; } BDD;
```

```
typedef struct node {
 char variable;
 char* bexpression;
 int value;
 struct node* parent;
 struct node* next;
 struct node* right;
 struct node* left; } NODE;
```

Then there is BDD_create function which gets Boolean function and order of variables from input. In order to work with Boolean function program needs to process the string that represents the Boolean function. The program does this through Shannon decomposition. Shannon decomposition is the identity: $F = x \cdot F_x + x' \cdot F_{x'}$ where F is any Boolean function, x is a variable, x' is the complement of x, and F_x and $F_{x'}$ are F with the argument x set equal to 1 and to 0 respectively. A more explicit way of static the theorem is: $f(X_1, X_2, \dots, X_n) = X_1 \cdot f(1, X_2, \dots, X_n) + X_1' \cdot f(0, X_2, \dots, X_n)$ (Wikipedia)

Thanks to this the implementation of dissecting a function in way of Shannon decomposition looks accordingly:

```
char* search_var_exp(char* bfunction, char variable, char state) { //Cuts
out variable from expression
 char* builder = (char*)malloc(100 * sizeof(char));
 char helper[100];
 int i = 0, j = 0, decisioner = 0;
 while (bfunction[i - 1] != '\0') {
     if (state == '!') {
          if ((bfunction[i - 1] != '!' && bfunction[i] == variable) || (i
          == 0 && bfunction[i] == variable))
              decisioner = 1; //if it sees selected variable without
   negation, it doesnt add this expression to builder
     }
     else {
         if (i != 0 && bfunction[i - 1] == '!' && bfunction[i] == variable)
             decisioner = 1; //if it sees negation of selected variable,
             it doesnt add this expression to builder
     }

     if (state != '!') { //leaves out variables that it is supposed to
     leave out
         if (bfunction[i] == '!' && bfunction[i + 1] == variable) {
             i = i + 2;
             decisioner = 1;
         }
         else if (bfunction[i] == variable)
             i++;
     }
     else {
          if (bfunction[i + 1] == variable && bfunction[i] == '!') //leaves
          out variables that is is supposed to leave out
             i = i + 2;
          else if ((i == 0 && bfunction[i] == variable) || (bfunction[i -
          1] != '!' && bfunction[i] == variable)) {
             i++;
             decisioner = 1;
         }
     }
```

```
        helper[j] = bfunction[i];
      if (i != 0 && bfunction[i] == '+') {    //if it sees +, then it knows
      its the end of expression
            if (decisioner != 1) {
                strcat_s(builder, 100, helper);
            }
            else
                decisioner = 0;
            memset(helper, '\0', 100);
            j = -1;
        }
      if (i != 0 && bfunction[i + 1] == '\0') {   //if it sees \0, then it
      knows its the end of the whole expression
            if (decisioner != 1) {
                strcat_s(builder, 100, helper);
            }
            else
                decisioner = 0;
            memset(helper, '\0', 100);
            j = -1;
        }
        i++; j++;
    }
```

2.1 Rules for BDD branching

After having everything setup, then it needs to create the root of the diagram. Creating a root and branching of diagram works the same way as in tree structures. This implementation did it through recursion. The new part is implementing a set of rules for correct decomposition of a function and reduction.

Firstly, all the rules need to be implemented for correct dissection of Boolean function. For start there are rules implemented straight from Boolean algebra, for instance !A*A = 0, !A + A = 0, A+A = 1, !A+!A = 1, and so on.

```
//A+A && !A+!A rule
if ((bfunction[0] == variable && bfunction[1] == '+' && bfunction[2] ==
variable && bfunction[3] == '\0' && node->parent != NULL)
     || (bfunction[0] == '!' && bfunction[1] == variable && bfunction[2] ==
'+' && bfunction[3] == '!'
         && bfunction[4] == variable && bfunction[5] == '\0' && node-
>parent != NULL)) {
     //printf("r01\n");
     node->right = one;//- type S reduction
     node->left = one;//- type S reduction
 }//A+!A rule
 else if ((bfunction[0] == variable && bfunction[1] == '+' && bfunction[2]
 == '!' && bfunction[3] == variable && bfunction[4] == '\0' && node->parent
 != NULL)
 || (bfunction[0] == '!' && bfunction[1] == variable && bfunction[2] == '+'
 && bfunction[3] == variable && bfunction[4] == '\0' && node->parent !=
 NULL)) {
     //printf("r1\n");
     node->right = one;//- type S reduction
     node->left = one;//- type S reduction
 }//A!.A rule
 else if ((bfunction[0] == variable && bfunction[1] == '!' && bfunction[2]
 == variable && bfunction[3] == '\0' && node->parent != NULL)
     || (bfunction[0] == '!' && bfunction[1] == variable && bfunction[2] ==
 variable && bfunction[3] == '\0' && node->parent != NULL)) {
     //printf("r2\n");
     node->right = zero;//- type S reduction
     node->left = zero;//- type S reduction
```

Program also needs to know how to differentiate between not negated variable and negated variable. In code it can be achieved this way:

```
if (node->bexpression[0] == '!') //negation
        node->right = zero;
    else
        node->right = one;

    if (node->bexpression[0] == '!') //negation
        node->left = one;
    else
        node->left = zero;
```

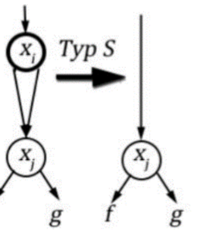
Furthermore, program needs to differentiate between disjunction and conjunction. When there is conjunction, generally, the whole left subtree is going to be 0 and then branch normally on right side. For disjunction the whole right subtree is going to be 1 and left subtree is going to branch normally.

```
else if (strlen(order) > 1 && strchr(bfunction, '+') != NULL &&
strlen(nv_exp1) == 0) {//if no negated variable expression is empty and
there is disjunction,
 //then program can set the whole right branch to zero
    node->right = zero;//- type S reduction
    if (strlen(morder) == 0 && strchr(nv_exp0, '+') == NULL)
{  //automatically can set to one, because of the disjunction of this node
        node->left = one;//- type S reduction
```

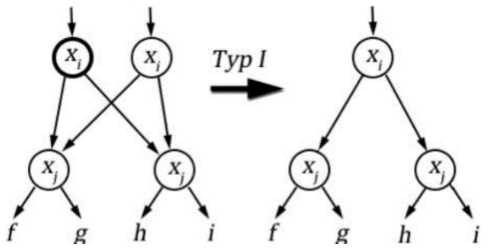
2.2 Reduction of BDD

In this implementation 2 types of reductions were used.

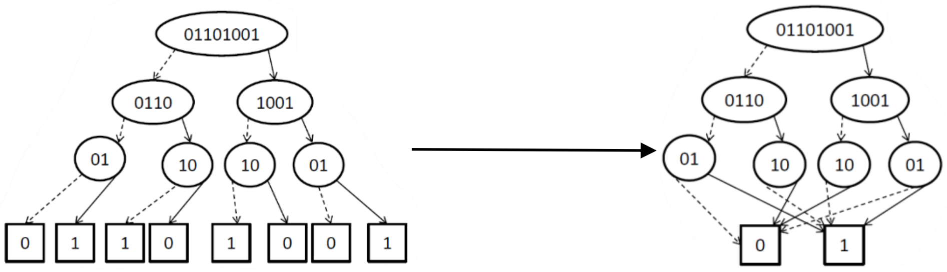
First reduction is type S reduction. It is removal of unnecessary nodes by comparing its descendants.



Second reduction is of type I. It is removing redundant nodes by comparing pairs. In this implementation were used hash tables to compare this redundant nodes and remove them.



Also, the BDD can also be reduced by only implementing 2 leaf nodes that represent zero and one. Then the pre-leaf nodes are going to point to only one 0 node or one 1 node instead of creating every time new one or zero node.



Implementation of reducing the leaf node is quite simple. In the first call of the BDD_create function 1 and 0 nodes are allocated and then passed onto BDD_branching. Implementation of it looks like this:

```
NODE* one = (NODE*)malloc(sizeof(NODE));   //creates 1 node
char* ones = (char*)malloc(2 * sizeof(char)); //one expression
one->variable = '1';
one->value = 1;
NODE* zero = (NODE*)malloc(sizeof(NODE)); //creates 0 node
char* zeros = (char*)malloc(2 * sizeof(char)); //zero expression
zero->variable = '0';
zero->value = 0;
```

This is how reduction I was implemented into the solution:

```
if (hashtable[h] != NULL) {
         if (hashtable[h]->bexpression == nv_exp1) { //type I reduction –
             using hash table
             node->right = hashtable[h];
             BDD_branching(node->right, node, root, node->right
             ->bexpression, morder, one, zero, hashtable, tablesize);
```

Furthermore, there is reduction S implementation in this solution:

```
if (hashtable[h]->bexpression == nv_exp0) { //type S reduction
     node->left = hashtable[h];
     BDD_branching(node->left, node, root, node->left->bexpression, morder,
     one, zero, hashtable, tablesize);
```

Testing

1. Own implementation of tester

In order to properly test the implemented reduced BDD implementation of tester is required. Firstly, program needs to generate a random function and store it in the string. For was implemented `char* random_boolean_function(int nOfVars, char** order) {}` function that generates such output.
Furthermore, a function that has an input of a vector needs to be implemented called BDD_use. Here is sample of code of such function:

```
int BDD_use(BDD* node, char* input) {
    if (node == NULL)
      return -1;
    int counter = 0;
    char value = input[counter];    //first 1 or 0
    int val = value - '0';  //conversion
    counter++;
    if (val == 1 && node->right != NULL) {
      return BDD_use(node->right, input, counter); //recursion to the right
    }
    else if (val == 0 && node->left != NULL) {
      return BDD_use(node->left, input, counter); //recursion to the left
    }
    return -1;
}
```

What's more is that program needs all permutations of vector to properly test the BDD. For this a converter to binary system was implemented, because every iteration of a number that is written in binary represents a different permutation of a vector needed for testing. Sample code of such function looks like this:

```
char* convert2binary(char* vector, int n) { //Simple binary converter
    int i;
    for (i = 0; n > 0; i++) {
       if (n % 2 == 1) vector[i] = '1';
       else vector[i] = '0';
       n = n / 2;
    }
 return vector; }
```

Additionally, program needs another Boolean function solver `void bfunction_solver(char*** bf_results, char* bfunction, char* order, int* nresults)` that is going to determine the output of a Boolean function in DNF. It looks at first conjunction of a function and determines the output based on variables and whether they are negated or not. Then jumps over the "+" symbol and does the same with another conjunction. These vectors are then stored in a two-dimensional array and used later for comparing of outputs from BDD_use.

Also, the tester needs another utility function that is going to compare the vector from bfunction_solver and a vector that was used in BDD_use to determine, if they are the same or not. If they are the same, then BDD is done well and if not, then BDD is not branching correctly. The simple implementation of such a checker looks like this:

```
int check_BDD_use(char *vector, char** bfunction_results, int nresults, int
length) { //Simple checking of results
  int i = 0, j = 0, decisioner = -1;
  char helper[30];
  for (i = 0; i < nresults + 1; i++) {
     for (j = 0; j < length; j++) {
         if (i == nresults + 1) {
             helper[j] = '\0';
             vector[j] = '\0';
             break;
         }
         helper[j] = bfunction_results[i][j];
         if (bfunction_results[i][j] == 'X')
             helper[j] = vector[j];
     }
     if (strncmp(helper, vector, length) == 0)
         return 0;
  }
  return -1;
}
```

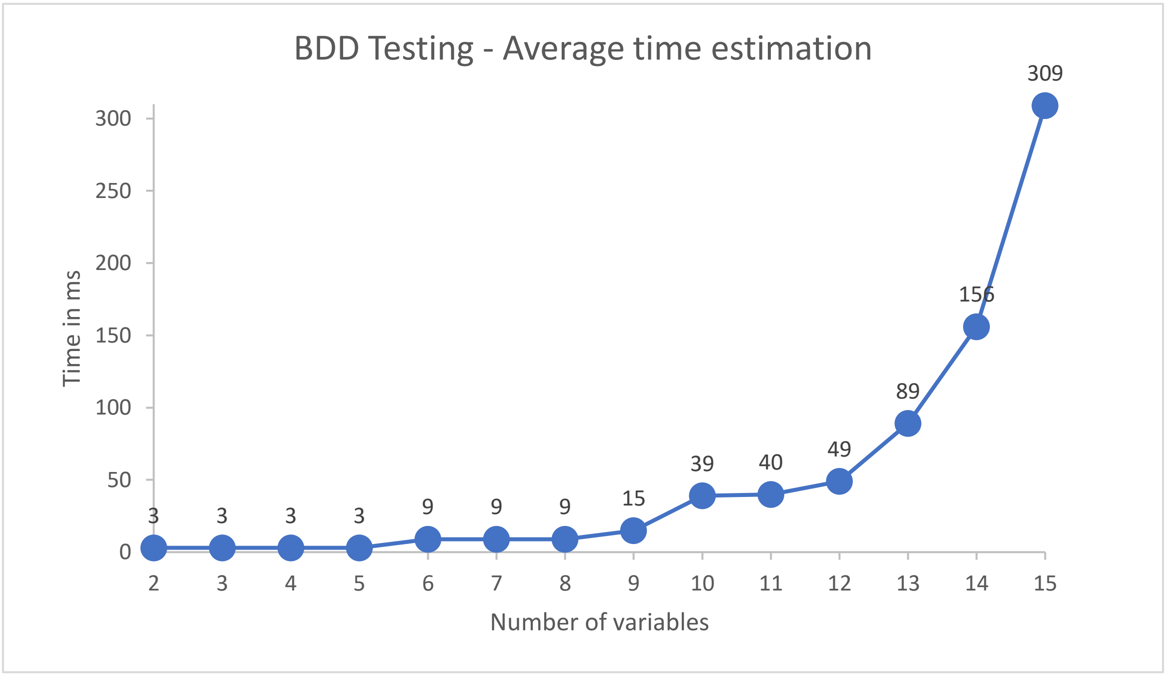
After all of this preparation the program can test the BDD in `void bfunction_tester (char* order, char *bfunction_test, char *order_test, BDD* bdd, int nresults)` function that calls all those utility functions and determines the outcome of BDD testing.

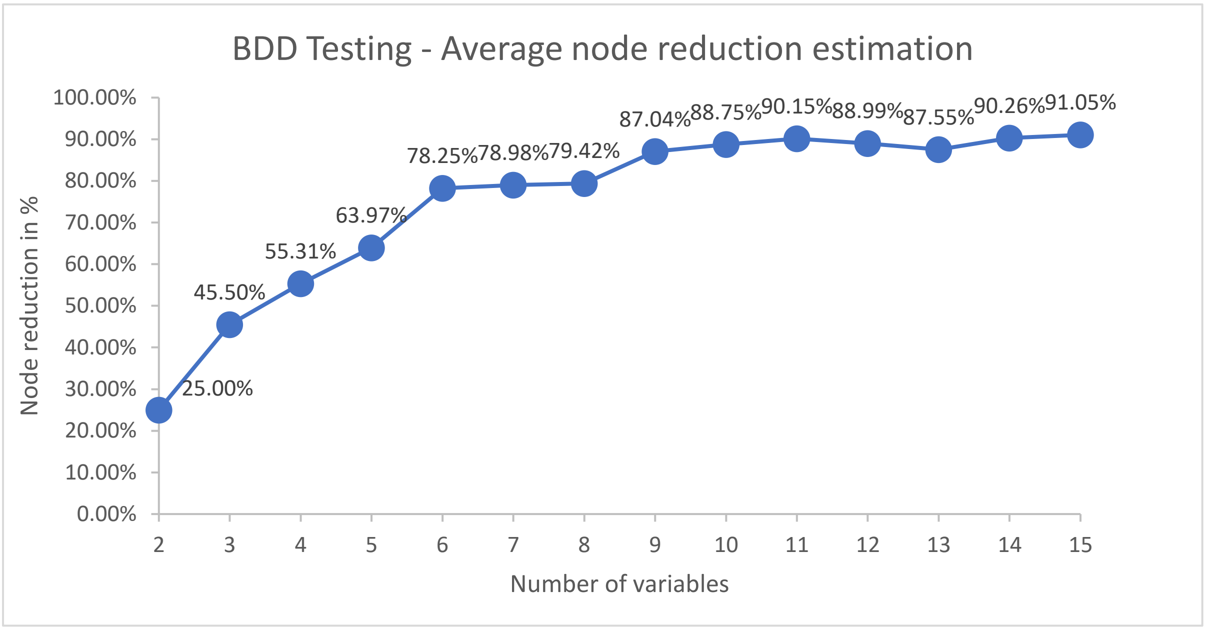
2. Tester results

The tester also measured the reduction of nodes, time complexity and documentation also contains memory complexity estimation of the implementation. Randomly generated function are created and then creation of BDD is precisely measured thanks to time.h library. The tester can generate desired number of BDD creations and after every creation, BDD is tested and then freed. All this data is stored and then presented in following tables and graphs.

Results:

BDD Creation	Number of variables	Number of random functions	Average time estimation (ms)	Average node reduction estimation
test1	2	100	3	25.00%
test2	3	100	3	45.50%
test3	4	100	3	55.31%
test4	5	100	3	63.97%
test5	6	100	9	78.25%
test6	7	100	9	78.98%
test7	8	100	9	79.42%
test8	9	100	15	87.04%
test9	10	100	39	88.75%
test10	11	100	40	90.15%
test11	12	100	49	88.99%
test12	13	100	89	87.55%
test13	14	100	156	90.26%
test14	15	100	309	91.05%





Conclusion

I know that solution is correct, because of the way I implemented Boolean function solver in testing. It looks at every product of the function and determines the output thanks to that product. For example, the function looks like this: ABC+AB. It looks at the first product ABC and knows, that for output 1 it needs A to be 1, B to be 1 and C to be 1. Everything else does not matter for this product ABC. Then it stores a vector of 111 and in function bfunction_tester compares the BDD_use vector that happened to have output 1 to all these stored vectors from Boolean function solver. If the vector is found in those solutions, then program knows, that BDD is correct. This happens to all possible permutations of vector that goes into BDD_use and gets checked.
Time complexity of this solution is estimated to be $O(\log(n))$. Memory complexity is estimated to be $O(n * \log{(n)})$. In my opinion, my implementation of this assignment is pretty efficient when I compare it to the first assignment. When it comes to testing evaluation, the bigger Boolean function became and the more variables it had, then in BDD_create it could reduce more and more nodes and stayed efficient even for large inputs. Overall, I found my implementation very well done, because it is 100% correct and time complexity and memory complexity is fairly low.