Data Structure and Algorithms

Binary Decision Diagram

Illia Ponomarov
Email: xponomarov@stuba.sk
Id: 113047

1. Introduction

What is a Binary Decision Diagram?

Binary decision diagrams provide a data structure for representing and manipulating Boolean functions in symbolic form. They have been especially effective as the algorithmic basis for symbolic model checkers. A binary decision diagram represents a Boolean function as a directed acyclic graph, corresponding to a compressed form of decision tree. Most commonly, an ordering constraint is imposed among the occurrences of decision variables in the graph, yielding ordered binary decision diagrams (OBDD). Representing all functions as OBDDs with a common variable ordering has the advantages that there is a unique, reduced representation of any function, there is a simple algorithm to reduce any OBDD to the unique form for that function, and there is an associated set of algorithms to implement a wide variety of operations on Boolean functions represented as OBDDs. Recent work in this area has focused on generalizations to represent larger classes of functions, as well as on scaling implementations to handle larger and more complex problems.

2. Implementation of Binary Decision Diagram

2. 1 Reduction of Binary Decision Diagram.

In my tree implementation, the tree cannot contain duplicates, but to save memory, my tree is truncated during its creation. To solve this problem, I used a Hash Table.
Each level of my tree contains its own hash table.

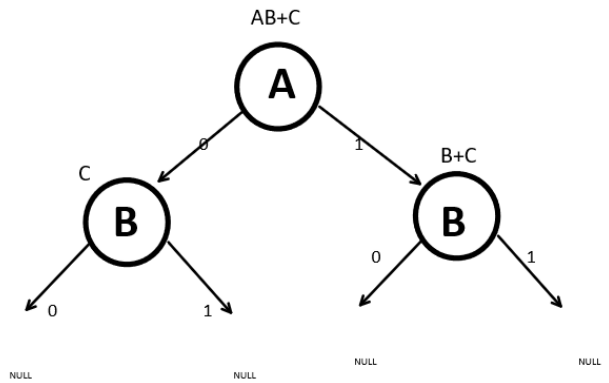
Hash Table:

In order for our tree to contain exclusively unique vertices, before adding a new vertex, I put it in a hash table to check the uniqueness of our vertex.

1. If the vertex is unique, then it is added to the hash table and a new vertex is created.
2. If the vertex already exists, then we do not create a new one, but return a link to the old, already created, vertex.
3. If we have a collision, then to solve it, I use Linear Hashing.
4. Exception, the hash table is not created only for the very first vertex.
   (This doesn't make sense since it's always a unique vertex.)

Example with unique vertices: :

Order: ABC, Function: AB + C

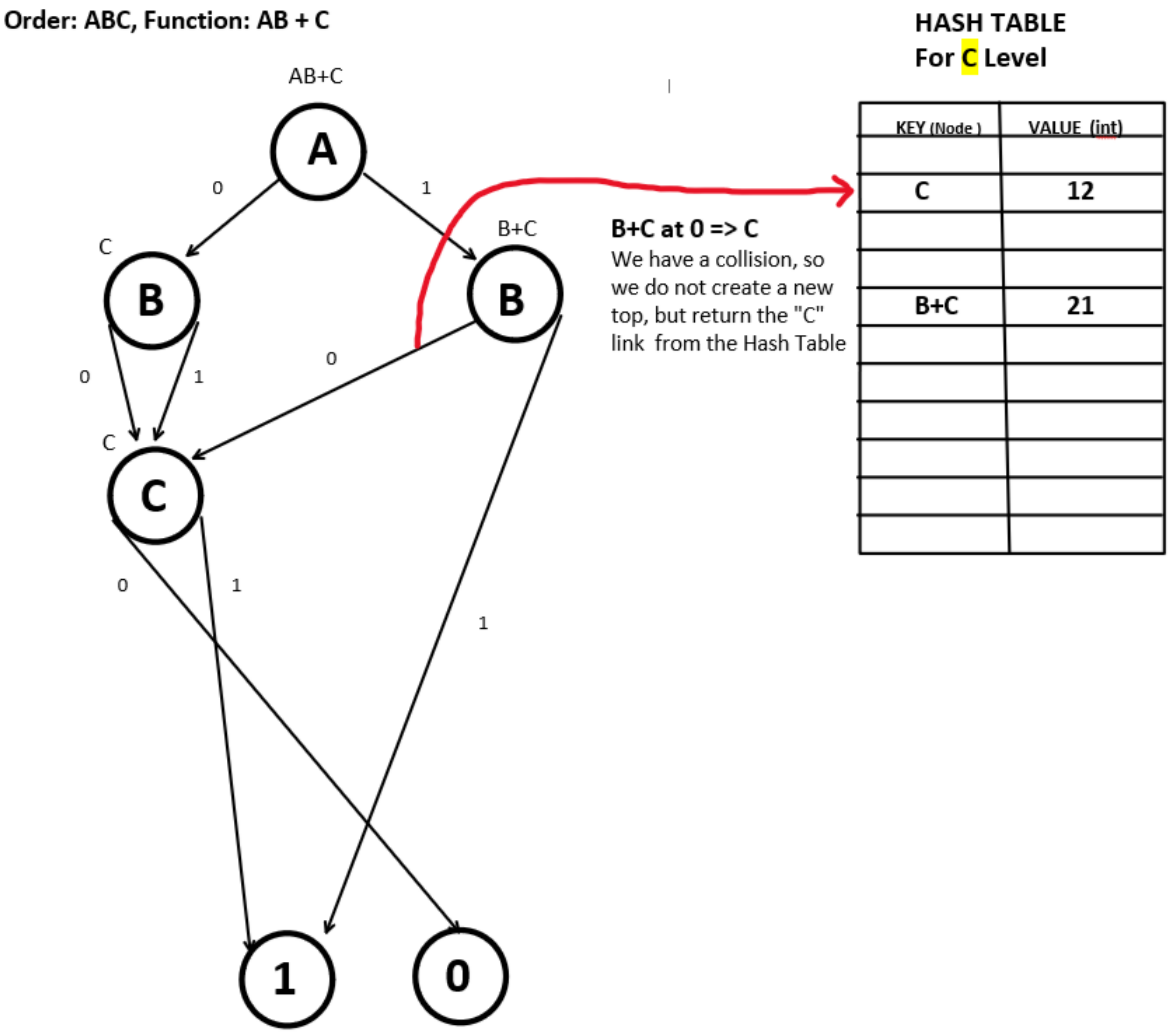


HASH TABLE
For B Level

KEY (Node)	VALUE (int)
C	232
B+C	21

No collision was encountered at this level

Example with getting a copy::



“BDD_create” function.

1) Algorithm for creating the first vertex:

1) As input we get a DNF function and its Order
2) Creating the very first vertex.
   .
3) We create children for the first vertex.
4) We initialize a hash table for the second level, in which we will store unique elements for each of our levels.
5) Adding references to a dynamic array (to my own implementation of a dynamic array). So that we always remember at what level we stopped.
   (This is so we don't go through all the vertices looking for vertices that don't have children created)
6) We reset the hash table

Algorithm Code:

```
this.order = order;
StringBuilder buffer_order = new StringBuilder(order);
DynamicArray currentStates = new DynamicArray();
int maxSize = (int) Math.pow(2, order.length());


if (root == null) {

    // We initialize a hash table for the second level, in which we will store unique elements for each of our levels.
    hashTable = new HashTable(maxSize);

    StringBuilder buffer = new StringBuilder(order);

    // Creating the very first vertex.
```

```
        this.root = new Node(bfunction, Character.toString(order.charAt(0)));

        // We create children for the first vertex.
        this.root.left = createLeftNode(root, hashTable, bfunction, order);
        this.root.right = createRightNode(root, hashTable, bfunction, order);

        buffer.deleteCharAt(0);
        order = new String(buffer);


        // Adding references of children the first vertex to a dynamic array (to my own implementation of a dynamic array).
        // So that we always remember at what level we stopped.
        //(This is so we don't go through all the vertices looking for vertices that don't have children created)
        currentStates.add(this.root.left);
        currentStates.add(this.root.right);


        // I reset the hash table
        hashTable = null;
    }
```

2) Algorithm of createLeftNode() and createRightNode() functions:

1. In the function argument, we pass the parent of our vertex (for which we want to create a child), the current hash table (to check for uniqueness), and Order the current order.

2. We generate a new DNF function. If we create a left vertex, then we substitute “0” in the current order. If to the right vertex, then we substitute 1 into the current order.

3. We send a new vertex to the hash table to check for uniqueness, and pass a random id to the hash table.

4. If after generating the DNF function we get “0”, then our left vertex will receive a link to the vertex “0”.

5. If after generating the DNF function we get “1”, then our left vertex will receive a link to the vertex “1”.

Code of createLeftNode() and createRightNode() functions

```
//In the function argument, we pass the parent of our vertex (for which we want to create a child),
// the current hash table (to check for uniqueness), and Order the current order.
public Node createLeftNode(Node root, HashTable hashTable, String bfunction, String order){


    //We generate a new DNF function. If we create a left vertex,
    // then we substitute "0" in the current bfunction (DNF function).
    if (order.length() >= 1)
        this.left_bfunction = getDNF(bfunction, Character.toString(order.charAt(0)), 0);


   // We send a new vertex to the hash table to check for uniqueness, and pass a random id to the
hash table.
    if (order.length() > 1)
        root.left = hashTable.insert(random_id, new Node(this.left_bfunction,
Character.toString(order.charAt(1))));

    //We send a new vertex to the hash table to check for uniqueness, and pass a random id to the
hash table.
    //The same, only if our order has a length of 1
    else if (order.length() == 1)
        root.left = hashTable.insert(random_id, new Node(this.left_bfunction,
Character.toString(order.charAt(0))));



   // If after generating the DNF function we get "0", then our left vertex will receive a link to the
vertex "0".
    if (root.left.bfunction.equals("0") ) {
        root.left = zero;
        return  root.left;
    }

   // If after generating the DNF function we get "1", then our left vertex will receive a link to the
vertex "1".
    else if (root.left.bfunction.equals("1")) {
        root.left = one;
        return root.left;
    }

    return root.left;
}

//In the function argument, we pass the parent of our vertex (for which we want to create a child),
```

```
// the current hash table (to check for uniqueness), and Order the current order.
public Node createRightNode(Node root, HashTable hashTable, String bfunction, String order){

    //We generate a new DNF function. If we create a left vertex, then we substitute "1" in the
current bfunction (DNF function).
    // If to the right vertex, then we substitute 1 into the current order.
    if (order.length() >= 1)
        this.right_bfunction = getDNF(bfunction, Character.toString(order.charAt(0)), 1);


    // We send a new vertex to the hash table to check for uniqueness, and pass a random id to the
hash table.
    if (order.length() > 1)
        root.right = hashTable.insert(random_id, new Node(this.right_bfunction,
Character.toString(order.charAt(1))));

    //We send a new vertex to the hash table to check for uniqueness, and pass a random id to the
hash table.
        // The same, only if our order has a length of 1
    else if (order.length() == 1)
        root.right = hashTable.insert(random_id, new Node(this.right_bfunction,
Character.toString(order.charAt(0))));


    //If after generating the DNF function we get "0", then our left vertex will receive a link to the
vertex "0".
    if (root.right.bfunction.equals("0")) {
        root.right = zero;
        return root.right;
    }

    //If after generating the DNF function we get "1", then our left vertex will receive a link to the
vertex "1".
    else if (root.right.bfunction.equals("1")  || (root.right.bfunction.equals(order) &&
root.right.bfunction.length() == 1)) {
        root.right = one;
        return root.right;
    }


    return root.right;
}
```

3) Algorithm for getDNF() functions for left and right nodes :

1) How is work?

This function shortens the current DNF. If we generate a function for the left vertex, then we substitute “0”, if for the right vertex, then “1”.

Example for function “AB+C”, current level “A”.

1) For the left vertex, we substitute “0” instead of “A” and get the result “C”.
2) For the right vertex, we substitute “1” instead of “A! and we get the result "B + C".

2) Algorithm for left node:

1) We split our function by pluses and get an array.
   For example: AB+C => [AB, C].

1) We begin to consider each element for the presence of symbols of the current level.

2) If the current element contains !A , and there are no characters in the string other than “!A” , then we return “1” because our function will definitely get “1” as output.

   For Example:

   !A + ABCDEFGH + !ABCD = 1 + 11100000+ 1000 or 1 + 0000000 + 1111 or

   1 + 101010101 + 1010 = 1.

   Then no matter what we substitute, we will always get “1” at the output. Then it does not make sense to create further vertices, then we will simply indicate the link to the vertex “1”.

3) If the current element contains !A , but there are characters of other levels in the string, then we simply remove the “!A”, since !0 => 1.

4) If the string contains the symbol of the current “A” level and the symbol of this level with a negation of !A - example: !AAB => AB => 0 , because 1 * 0 or 0 * 0 = 0.

5) If the string does not contain characters of the current level, then we do not change anything in it C => C.

6) If the negated element of the current level does not contain "!A", but contains just a character, then AB => 0.

7) If, after the reduction, there are no elements left in our function, then we return 0.
   For example: AB + AC + AD => 0;

2.1) Algorithm code of getDNF function for Left node:

```
    if (side == 0){

      StringBuilder buffer = new StringBuilder();

    // 1) We split our function by pluses and get an array.
    //For example: AB+C => [AB, C].

      zeroList = bfunction.split("\\+");
      zeroList = removeDuplicate(zeroList);
      String result = "";
      String buffer_string = "";

   // 2) We begin to consider each element for the presence of symbols of the current level.
       for (String str: zeroList) {
                        if(str.contains("!"+order+"+"+order) || str.contains(order+"+"+ "!" +
order))
          return "1";



 // 3) If the current element contains !A , and there are no characters in the string other than "!A" ,
 // then we return "1" because our function will definitely get "1" as output.
 //For Example:
 //
 //  !A + ABCDEFGH + !ABCD  =  1 + 11100000+ 1000   or
 //   1 + 0000000 + 1111  or
 //   1 + 101010101 + 1010  =   1.
 //
 //Then no matter what we substitute, we will always get "1" at the output.
// Then it does not make sense to create further vertices, then we will simply indicate the link to the
vertex "1".
    if (str.contains("!" + order)) {


// 4) If the current element contains !A , but there are characters of other levels in the string,
 // then we simply remove the "!A", since !0 => 1.
          if (str.equals("!"+order))
              return "1";

      str = str.replaceAll("!" + order, "");


  // 5) If the string contains the symbol of the current "A" level and the symbol of this level with a
negation of !A -
  // example: !AAB => AB => 0 , because 0 * 1 or 0 * 0 = 0.
```

```
      if (str.contains(order))
            continue;

 // 6) If the string does not contain characters of the current level, then we do not change anything
in it C => C.
      else
        result+= str + "+";
                                }

 // 7)  If the negated element of the current level does not contain "!A",
  // but contains just a "A", then AB => 0.
   else if (str.contains(order) )
           continue;
// 8) If the string does not contain characters of the current level,
   // then we do not change anything in it C => C.
    else
      result += str + "+";

       }



      buffer = new StringBuilder(result);

    if(buffer.lastIndexOf("+") != -1)
                                          buffer.deleteCharAt(buffer.length()-1);

 // If, after the reduction, there are no elements left in our function, then we return 0.
//For example: AB + AC + AD => 0;
         if (result.length() == 0)
              return "0";

         // DNF correction
         if (result.length() > 1){

            result.replace("++", "+");

        buffer = new StringBuilder(result);
                                  buffer.deleteCharAt(buffer.length() - 1);
    result = new String(buffer);
     }

    return result;

  }
```

3) Algorithm for right node:

1) We split our function by pluses and get an array.
   For example: AB+C => [AB, C].

2) We begin to consider each element for the presence of symbols of the current level.

3) If the current element contains A , and there are no characters in the string other than “A” , then we return “1” because our function will definitely get “1” as output.

   For Example:

   A + ABCDEFGH + !ABCD = 1 + 11100000+ 1000 or 1 + 0000000 + 1111 or

   1 + 101010101 + 1010 = 1.

   Then no matter what we substitute, we will always get “1” at the output. Then it does not make sense to create further vertices, then we will simply indicate the link to the vertex “1”.

4) If the current element contains !A , but there are characters of other levels in the string, then we simply remove the “!A”, since !0 => 1.

5) If the string contains the symbol of the current “A” level and the symbol of this level with a negation of !A - example: !AAB => AB => 0 , because 0 * 1 or 0 * 0 = 1.

6) If the string does not contain characters of the current level, then we do not change anything in it C => C.

7) If the negated element of the current level does not contain "!A", but contains just a character, then AB => 0.

8) If, after the reduction, there are no elements left in our function, then we return 0.
   For example: AB + AC + AD => 0;

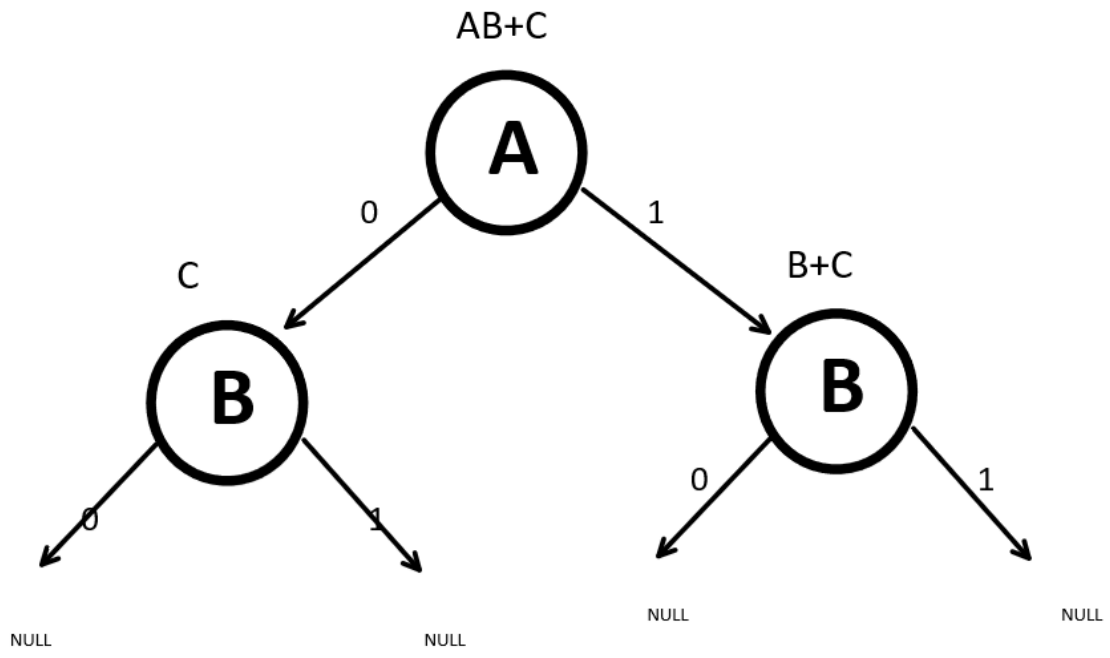
3.1) Algorithm code of getDNF for Right node:

```
if (side == 1)
{
    StringBuilder buffer = new StringBuilder();
```

```
        //We split our function by pluses and get an array.
        //For example: AB+C => [AB, C].

        oneList = bfunction.split("\\+");
        oneList = removeDuplicate(oneList);
        String result = new String();
        String buffer_string = new String();


        //We begin to consider each element for the presence of symbols of the current level.

        for (String str: oneList) {

            //If the current element contains A , and there are no characters in the string other than "A" ,
            // then we return "1" because our function will definitely get "1" as output.
            if (str.contains(order) && str.length() == 1)
                return "1";

            //If the current line has a negation in a character, then we delete that line.
            //For example: !ABC+C => C
            if (str.contains("!" + order))
                continue;


            //If the current element contains A , and there are no characters in the string other than "A" , then we return "1" because our function will definitely get "1" as output.
                //
                //For Example:
                //
                //A + ABCDEFGH + !ABCD  =  1 + 11100000+ 1000   or  1 + 0000000 + 1111  or
                //
                //1 + 101010101 + 1010  =   1.
                //
                //Then no matter what we substitute, we will always get "1" at the output.
                // Then it does not make sense to create further vertices, then we will simply indicate the link to the vertex "1".

            else if (str.contains(order)) {
                if (str.equals(order))
                    return "1";

                result += str.replaceAll(order, "") + "+";
            }
            //If the string does not contain characters of the current level, then we do not change anything in it C => C.
            else
```

```
                result += str + "+";
        }

        // If, after the reduction, there are no elements left in our function, then we return 0.
        //For example: AB + AC + AD => 0
        if (result.equals(""))
                return "0";

        if (result.length() > 1){
                result.replace("++", "+");
                buffer = new StringBuilder(result);
                buffer.deleteCharAt(buffer.length() - 1);
                result = new String(buffer);
        }

        return result;

}
```

Current BDD :



1) Algorithm for creating the remaining vertices.

1) After setting the first three vertices, we create a loop in which we iterate over our elements from the dynamic array (children of the previous vertex) that we added. Now they become parents and we create children for them.

2) After creating children for these vertices, we add them to the dynamic array too, to repeat the same algorithm.

3) When we have created all the children for the current vertices, we reset the hash table for the current level.

4) Children become parents and the current level changes.

1.1) Algorithm code for creating the remaining vertices.

```
//After setting the first three vertices, we create a loop in which we iterate over our elements from the dynamic array (children of the previous vertex) that we added.
// Now they become parents and we create children for them.

for (int i = 0; i < currentStates.size(); ) {


    if (!(currentStates.get(i).bfunction.equals("0")) &&
!(currentStates.get(i).bfunction.equals("1")) && (currentStates.get(i).left == null &&
currentStates.get(i).right == null)) {

        currentStates.get(i).left = createLeftNode(currentStates.get(i), hashTable,
currentStates.get(i).bfunction, order);
        currentStates.get(i).right = createRightNode(currentStates.get(i), hashTable,
currentStates.get(i).bfunction, order);

        //After creating children for these vertices, we add them to the dynamic array too, to repeat the same algorithm.
        child.add(currentStates.get(i).left);
```

```
            child.add(currentStates.get(i).right);
        }

        //When we have created all the children for the current vertices, we reset the hash table for
the current level.
         if (i == currentStates.size() - 1) {


             //Children become parents and the current level changes.
            currentStates = child;
            hashTable = new HashTable(maxSize);
            i = 0;

            if (order_builder.length() >= 1)
                order_builder.deleteCharAt(0);

            if (order_builder.length() == 0)
                break;

            child = new DynamicArray();
        }
        else
            i++;

        if (order_builder.equals(""))
            break;

        order = new String(order_builder);
}


return new BinaryDecisionDiagram(root);
```

4) Implementation of Linear Hash Table.

The size of the hash table is calculated by the formula 2^n , since each vertex has at least two vertices, and with each level, the number doubles.

4.1) Hash Function

The hash function divides the hash code by the modulo table size.

Code:

```
public  int hashFunction(String value){
   return hashCode(value) & capacity;
}
```

:

4.2) Hash Code

To get the hash code, I used the algorithm of summing products over the entire text of the string. For example, if an instance s of class java.lang.String would have a hash code h(s) defined

h(s)=s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

where terms are summed using Java 32-bit integer addition, s[i] denotes the i-th character of the string, and n is the length of s.

Code:

```
public  int hashCode(String value){
   int hash_so_far = 0;
   final char[] c_string = value.toCharArray();

   long p_pow = 1;

   //s[0]*31^(n - 1) + s[1]*31^(n - 2) + ... + s[n - 1]
   for (int i = 0; i < value.length(); i++) {
       hash_so_far = (int) ((hash_so_far + (c_string[i] - 'a' + 1) *
p_pow) % m);
       p_pow = (31 * p_pow) % m;
   }

   return hash_so_far;
```

```
}
```

4.3) Insert to Hash Table

Algorithm:

1) We calculate the hash function for a regular string.

2) If there is no collision, then we create a new node and add it to the hash table, and return a reference to it.

3) There is a collision:

   1) If the elements are the same, then we return a link to the top in the hash table.

   2) If the elements are different, then we will go through the table until we find an empty cell.

4) If loadFactory is greater than 0.75, then the resize() function is called, which doubles the size of the table.

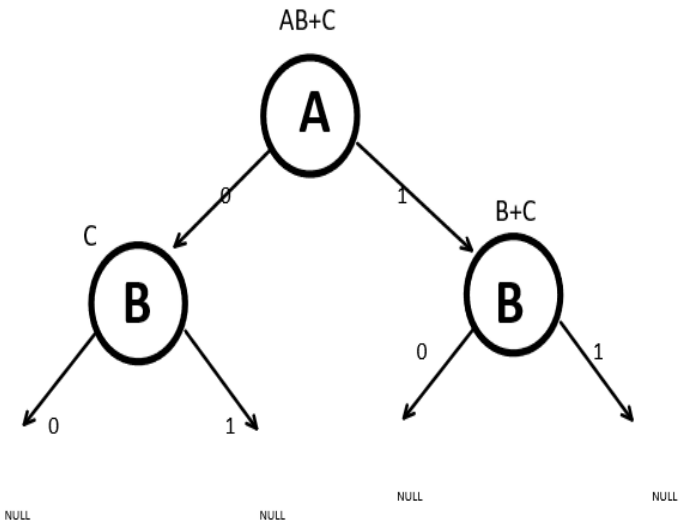
```
public Node insert(int key, Node node){

   // We calculate the hash function for a regular string.
   int index = hashFunction(node.bfunction);

   if (index == hashtable.length)
       resize();

   HTObject  htObjectNormal = hashtable[index];


   //If there is no collision,
   // then we create a new node and add it to the hash table, and return
a reference to it.
   if (htObjectNormal == null) {
       hashtable[index] = new HTObject(key, new Node(node.bfunction,
node.order));
       Main.MEMORY++;
       if (index >= hashtable.length - 1 )
           resize();


       return hashtable[index].value;
   }
```

```
   if ( (htObjectNormal != null)) {

       if (!htObjectNormal.value.bfunction.equals(node.bfunction)) {

           //There is a collision:
           //2) If the elements are different, then we will go through
the table until we find an empty cell.
           for (int i = index; i < hashtable.length; i++) {
               if (hashtable[i] == null) {
                   hashtable[i] = new HTObject(key, new
Node(node.bfunction, node.order));
                   Main.MEMORY++;
                   if (i >= hashtable.length - 1)
                       resize();;

                   return hashtable[i].getValue();
               }
           }
       }


       //There is a collision:
       //1) If the elements are the same, then we return a link to the
top in the hash table.

       return hashtable[index].value;
   }



   ///If loadFactory is greater than 0.75, then the resize() function is
called,
   // which doubles the size of the table.
       size++;

       double loadfactory = (1.0 * size) / capacity;

       if (loadfactory > this.loadFactor)
           resize();



 return null;
}
```

Create BDD step by step:

1) Initially, when a tree is created, the very first root node and its two children are created. The children will be added to the hash table to check if the vertices are unique.
   Now we have moved to level B. **Reset the Hash Table**

Order: ABC, Function: AB + C



HASH TABLE
For B Level

KEY (Node)	VALUE (int)
C	232
B+C	21

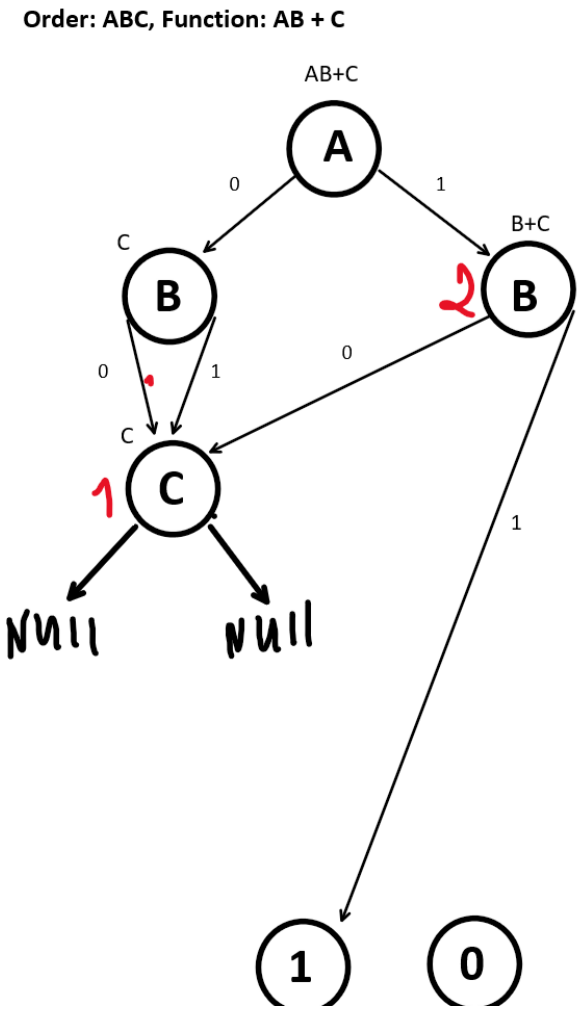
No collision was encountered at this level

2) Now we will create vertices for level C.
Vertex 1. Has a DNF function “C”, since we were at level B, and therefore nothing changes for us and we create a vertex with the same DNF.

Pinnacle 2.
When creating the left vertex, we substitute "0" into the function "B+C" and get the value "C". We add a table to the hash, and we see that such a vertex already exists, so we do not create a new vertex, but make a link to the already created vertex 1.

When creating the right vertex, we substitute "1" into the "B+C" function.
Since we have B == 1, then our function will in any case have a result of 1.

For example:
B + C ⇒ 1 + 0 or 1 + 1 ⇒ 1
In any case, we get “1”, so it doesn't make sense for us to create new vertices, and then we attach a link to the vertex “1”.
а создавать новые вершины, и тогда мы прикрепляем ссылку на вершину “1”.



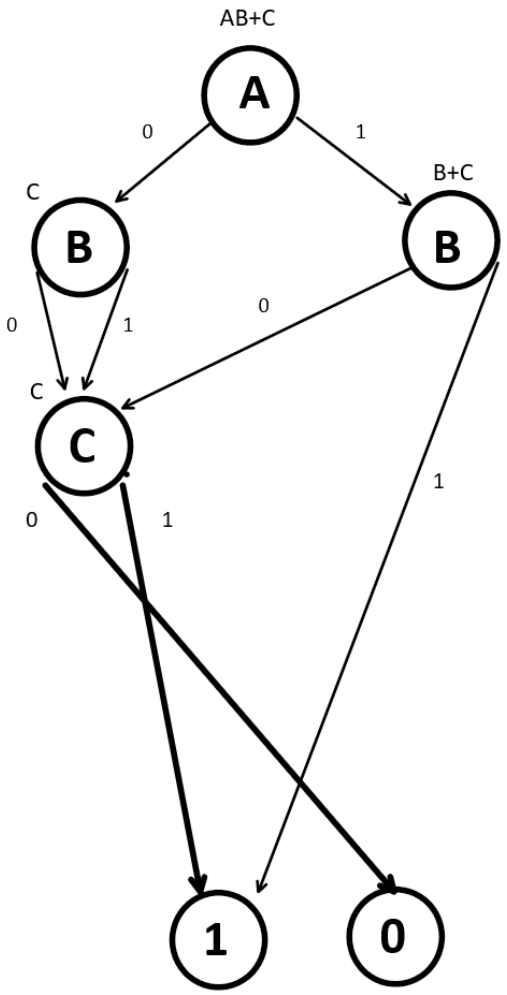
B+C at 0 => C
We have a collision, so we do not create a new top, but return the Node link from the Hash Table

HASH TABLE
For B Level

KEY (Node)	VALUE (int)
C	12
B+C	21

3) Now we are at the level.
   At this level, we have only the top “C” left.
   We add a table to the hash, since we have no collisions.
   Set the link of the left vertex to "0" and the right vertex to "1"

Order: ABC, Function: AB + C



HASH TABLE
For B Level

KEY (Node)	VALUE (int)
C	12

“BDD_use” function

1) Algorithm:

1) The function takes in the function argument: BDD tree and input_values.
The BDD tree is the tree that will be searched.
input_values are the values we will be searching for, such as "100" or "010".

2) Compare string lengths , **input_values** and order length. They must be the same length.

3) We declare a loop, and go through each character of the input_values string.

4) If the symbol is “0”, then we turn left, along the tree.

5) If the symbol is “1”, then we turn right, along the tree.

6) If the current position is at vertex “0”, then we return 0.

7) If the current position is at vertex “1”, then we return 1.

8) If we did not find anything, then we will return -1.

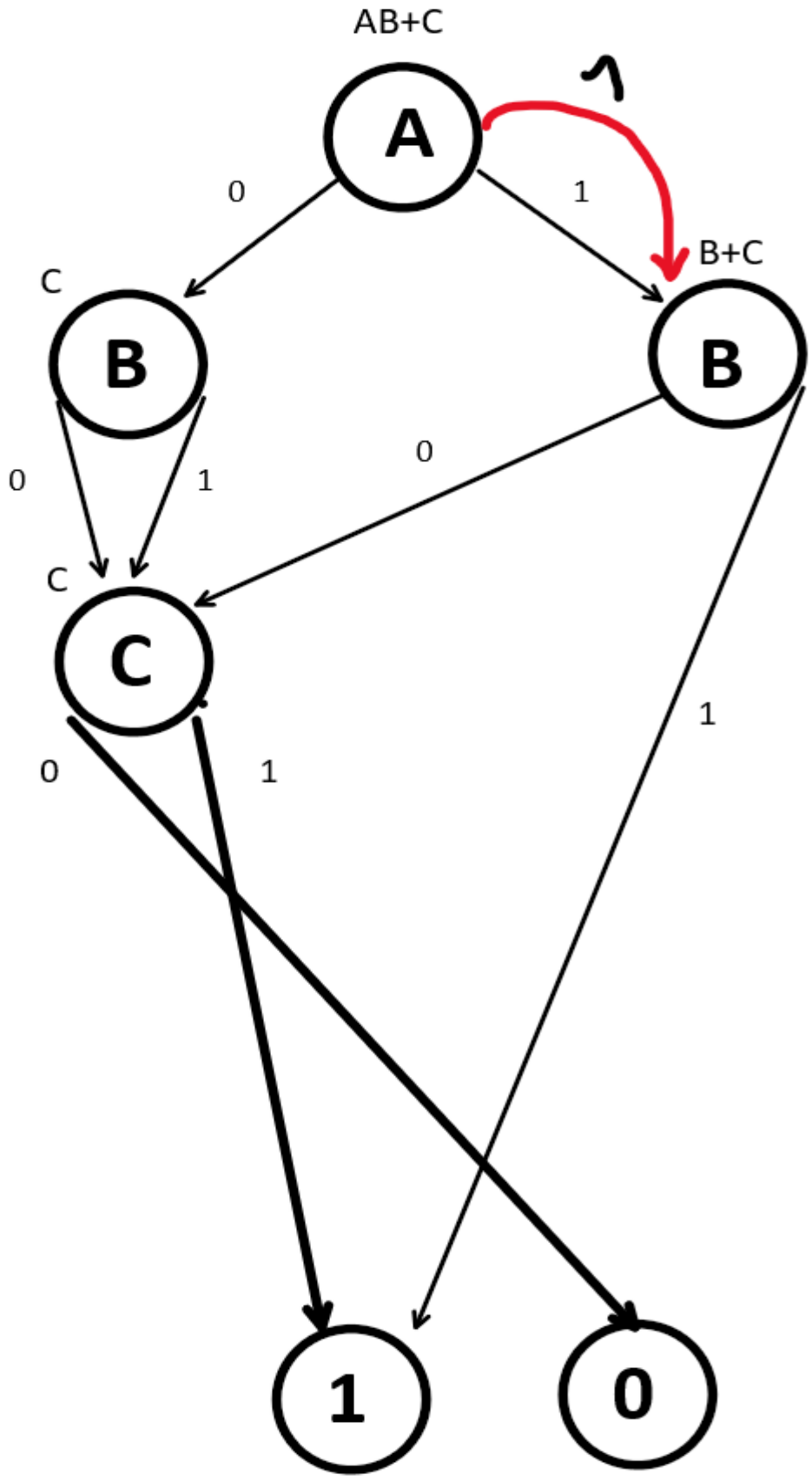
2) BDD_use code:

```
public int BDD_use(BinaryDecisionDiagram binaryDecisionDiagram, String
input_values){

   Node current = binaryDecisionDiagram.root;

   if (input_values.length() != this.order.length()) {
           System.err.println("Order and Way should be identical.");
           return -1;
   }

       for (Character number: input_values.toCharArray()) {


               if (number.equals('0'))
                       current = current.left;

               else if (number.equals('1'))
```

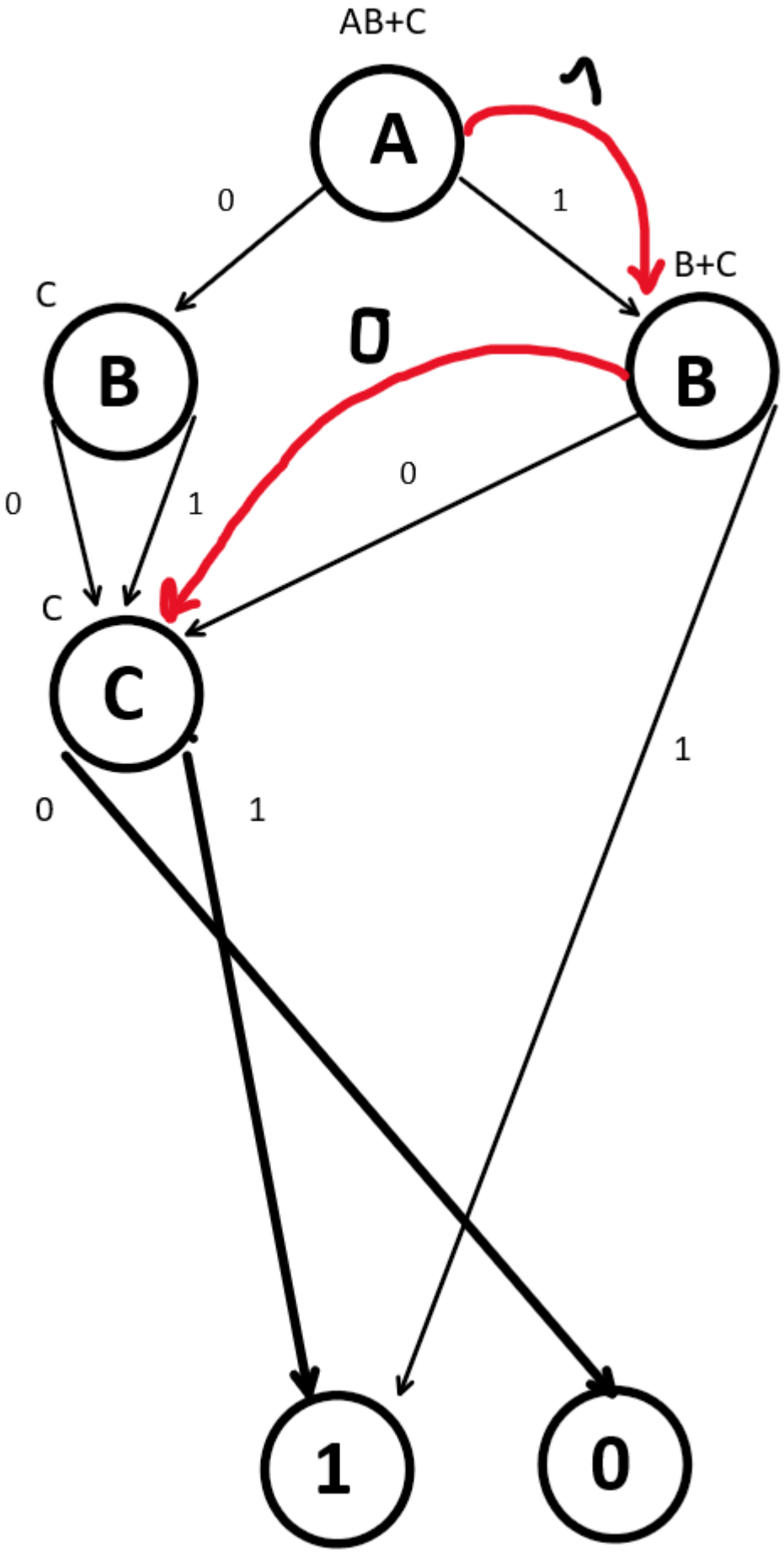
```
                        current = current.right;

                if (current == zero) {
                        return 0;
                }
                else if (current == one) {
                        return 1;
                }


        }


    return -1;
}
```

Algorithm of Search in BDD step by step:

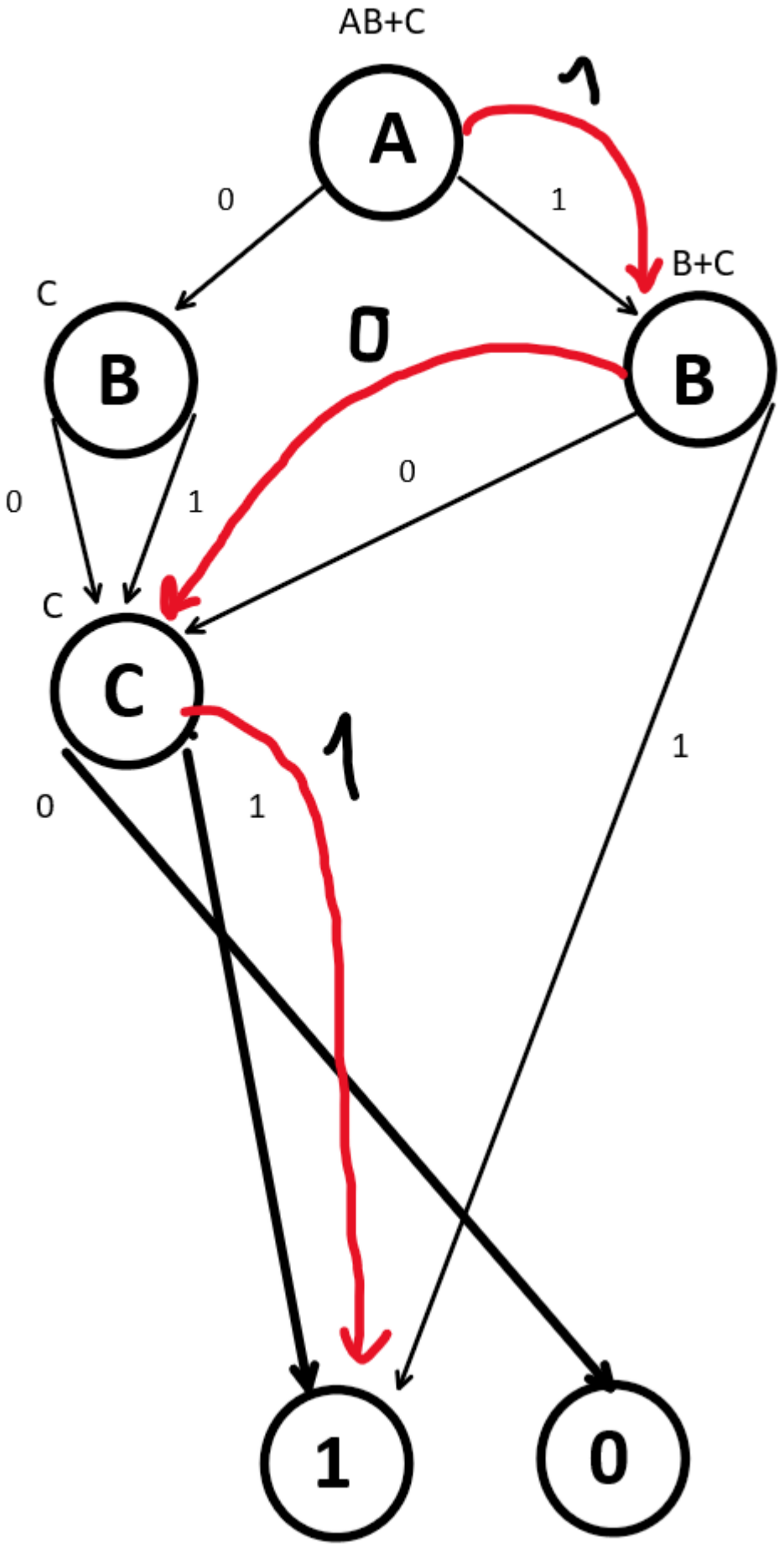
Input_values: 101



Input_values: 101



Input_values: 101



Program Testing

Testing algorithm:

1) A random DNF function is generated.

2) A truth table is generated according to its order (order = ABC, random dnf = AB + C).

   For example: If our order is “ABC”, then a truth table for three elements will be generated.

   We substitute all our values from the truth table into our randomly generated DNF function, and we get the result that we want to find in the tree.

   For example:

A	B	C	AB+C
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

3) Next, the BDD_use () function is called, into which it is written to the input data from the truth table (000, 001, 010 ...).

   If we return values that are false to the result in the truth table, then an error is displayed, otherwise the tree will continue to search the tree,
   until we go through all the values in the truth table.

4) We repeat the above algorithm a thousand times.

Time and Memory.

The table below shows DNF tests of more than 100 expressions, and more than 1000 trees have been tested.

Size of Order	The number of trees	Applied BDD_create() time in seconds	Applied BDD_use() time in seconds	Average reduction success
3	1000	0,000125	0,000076	87,5000 %
4	1000	0,000125	0.000067	93,76 %
5	1000	0,000145	0.000063	93,74987 %
6	1000	0,000159	0,000109	92,75543 %
7	1000	0,000230	0,000078	92,96875 %
8	1000	0,000396	0,000070	91,40625 %
9	1000	0,000634	0,000072	88,47656 %
10	1000	0,001207	0,00092	84.86328 %
11	1000	0,002568	0,000135	80,93164 %
12	1000	0.006512	0,000188	78,73242 %
13	1000	0.011501	0.000109	74.33496 %

"Average reduction success" I count, according to the formula:

max - the maximum possible number of vertices without reduction.
notCreated = the number of vertices not created).

Reduction = (notCreated / max) * 100;

An example of one DNF function that is being tested:

IDRYK!L!B!A+IFD!Y!L!NB!A+IF!YKMNBA+!WY!KLBA+!I!D!YK!L!MN!B+!D
!W!RKLM!NB!A+IDR!K!L!MB!A+!ID!K!L!BA+!IDYKLN!A+W!M!NB!A+DW!R
!K!LM!A+!F!DW!RN+!I!FRY!KL!B+I!F!K!MN!BA+K!N!B+IFDWYL+!I!M+!IYK
!N!A+!MNB+IR!LM!BA+D!WK!M+!WR!L!MN!BA+I!FW!YKLA+I!D!R!MNA+I!
RYLA+!WKL!NA+W!KM!A+!F!DR!NB+F!YKLM!NBA+!YB!A+F!WK!LM!B+!I!
WY!K!L!N+!IFN!B+DW!N!A+W!Y!M!N+IFDWMB+!IWR!K!L!M+D!R!KM!N!B
+I!RNB!A+I!FDR+IRKMA+!I!D!L!M!A+DWRKL!B+!F!D!RY!K!LBA+!RYKN!A
+F!W!LB+IF!DLNBA+FRK!L!M+!R!K!M+F!RLMBA+IF!R!KNB+IF!D!RY!K!L
MN!BA+!I!WR!L!MA+I!D!W!NB+I!DWRY!KM!NBA+!F!D!RL!MNA+IFW!RY!
K!L!N!A+I!D!WY!K!MNA+I!F!R!YKL!M!B+DWRY!BA+D!RY!KN+I!F!RKLMB
+!F!DR!Y!K!M!N!B+DRYLNA+!FWRN!B+!FRYN+Y!M!NA+RYL!N+RK!L!M!N
BA+I!D!WY!K!L+FRL!MA+!IFDR!YN!A+!I!D!WKM+!F!L!M!N!A+!WK!A+FW!
Y!LNB+!I!YK!A+IF!WR!L!NB!A+IFW!M!N+!F!D!RYK!L!NA+D!Y!KNBA+I!F!
DWR!YLB!A+!DW!RL!NB!A+D!WRK!B+!IF!R!L!N+!I!W!RLBA+!I!FW!RYK!L
+!FR!Y!K!B+!FR!KM!NA+F!KL!MN!A+IFDRYNB+!F!R!YK!LBA,