# Assignment 2 Report

## Part I

## Explanation of the algorithms of function

*bdt buildcompactbdt(const std::vector<std::string>& fvalues)*

Firstly, this function builds an uncompacted data structure with every variables in order using function *bdt buildbdt(const std::vector<std::string>& fvalues)*, which is identical to the function with the same name in Assignment 1. It builds the internal nodes containing “x1”, “x2” … “xn” and then puts“0” or “1” in leaf nodes. Each internal node has a left pointer pointing to the 0 branch and a right pointer pointing to the 1 branch.

Secondly, it uses another function *void compactbdt (bdt& t)* to compact the data structure. The binary tree should be compacted from its root because there is no constraint on the order of variables. Therefore, the input of this function is always a node pointer representing a balanced tree which has completely same structure of its left and right subtree (same height, same and ordered variable at each internal level), only the leaf nodes may have different value. If each corresponding pair of leaf nodes of left and right subtrees also has same value, the left and right subtrees are completely the same, for example, a tree (x1) shown in Figure 1,

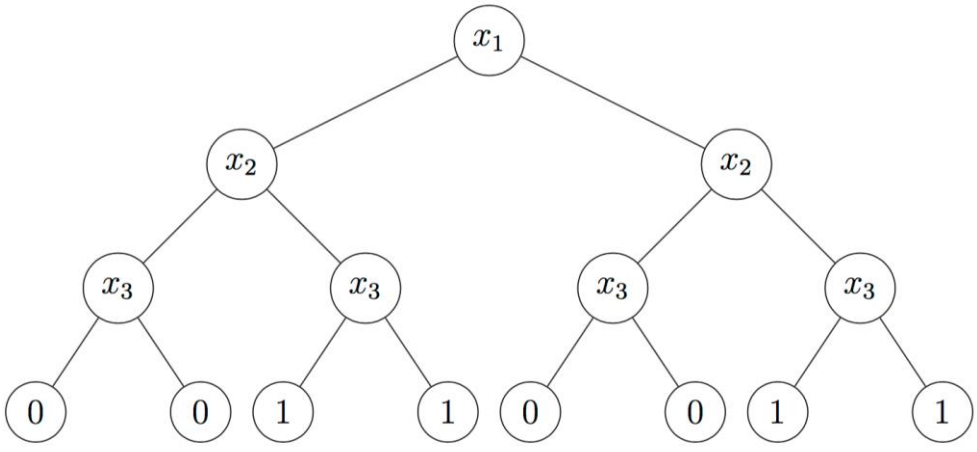



Figure 1.

It can be compacted, in other words, replaced by another tree (x2) shown in Figure 2.

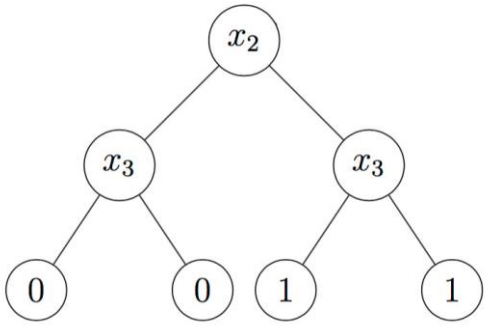



Figure 2.

How to implement the function: *void compactbdt (bdt& t)*

If the balanced tree can be compacted:

1. Its left and right pointer must not point to “NULL”, otherwise, it has already been compacted to the largest extent and cannot be compressed any more.

2. It has same left and right subtree (same in both structure and value). Here, I used a recursive function *bool sametree (bdt t1,bdt t2)* to check the sameness. It firstly checks the left subtrees, then the root node by using function *bool sameroot(bdt t1,bdt t2)* to check if the values inside are the same , then the right subtrees. If any inconsistence is detected, this function will return 0, otherwise, return 1.

3. To carry out the replacement, the function needs to delete one of its subtrees (I chose to delete the right one) using a function *void deletetree(bdt t)*, then replace the tree node pointer by its left pointer, then delete the previous tree node. The procedure is illustrated in Figure 3.

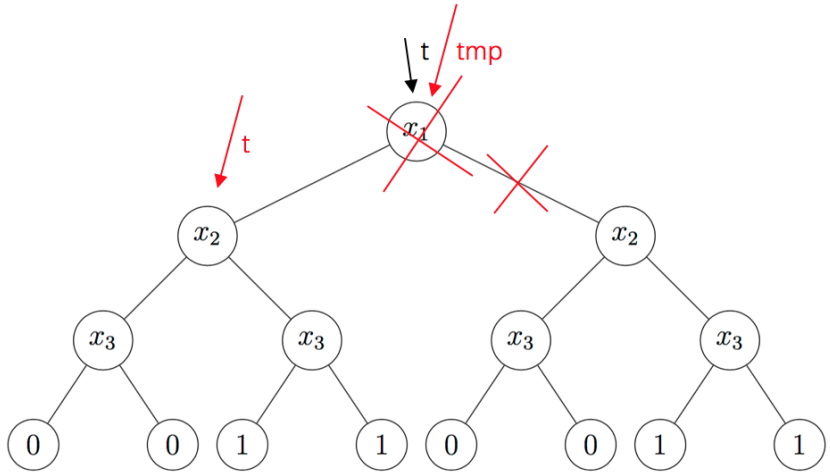



Figure 3.

4. At last, compact the new tree “t” again recursively.

If the balanced tree cannot be compacted:

1. Its left and right subtree are not completely the same.

2. Try to compact its left subtree then right subtree, which is also a recursive procedure and calls the function itself.

Part II

## Explanation of the algorithms of function

*std::string evalcompactbdt(bdt t, const std::string& input)*

Evaluating a function represented as a compact data structure is different from a normal one. As the order of variables are not successive, but still ascends. Therefore, it is enough to get to the result by only taking the elements of the input string required by the internal nodes of the tree.
In order to implement this function:

1. Firstly, a while loop and a mobile temporary pointer (“tmp”) pointing to the tree node check whether the value of the node is a string of “0” or “1”.

2. If it isn’t, find which variable “xn” it needs to decide which way to go, and then find the corresponding element of the input string to be evaluated.
   If the element is “1”, change “tmp” to its right pointer, otherwise, to the left one.

3. Keep going until the value pointed by “tmp” is “0” or “1”and return the string.

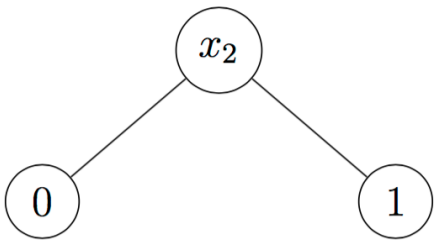
Part III

Experimental evaluation:

I evaluated this Boolean function:

| $x_1$ | $x_2$ | $x_3$ | $f(x_1, x_2, x_3)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

The compact tree should look like this:




After building the compact tree

"*bdt bdt = buildcompactbdt(input)*", then using the recursive function *void printtree(bdt t)* to print left subtree, then the node (*void printroot(bdt t)*), then the right subtree, I got the correct output:

```
0
x2
1
```

Then I evaluated all the possible inputs from "000"to "111"

"*std::cout << evalcompactbdt(bdt, "000") << std::endl;*"

I got the output:

0

0

1

1

0

0

1

1

Which was consistent with the results shown in the form.