Ryan Mansoor
CID: 01224750

Algorithms and Data Structures Assignment 2 - Spring 2018

Introduction

Given the 'true' outputs of a truth table, this assignment requires a program that can generate the smallest data structure for the table so that it can be evaluated efficiently and use minimal extra memory. This report provides an explanation of the algorithms used to create the functions that build and evaluate these data structures. An experimental evaluation of the program is carried out to test for robustness.

Problem

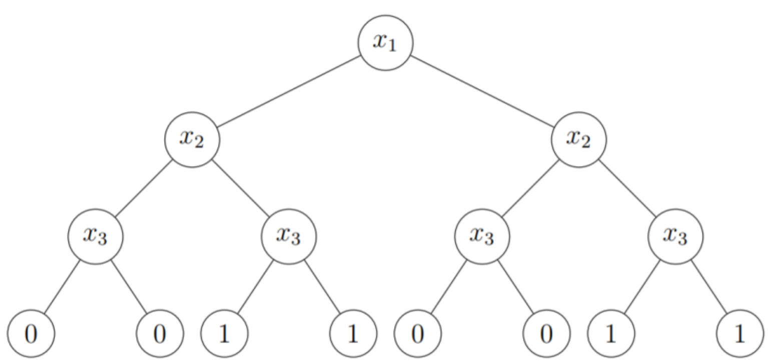
Truth tables will most effectively be stored in a binary tree, as a 0 or a 1 can be translated into the left child and right child of a node. The problem can be best explained using the given example:

Inputs ('true' outputs of truth table): "010", "011", "110", "111".

These inputs represent the following truth table:

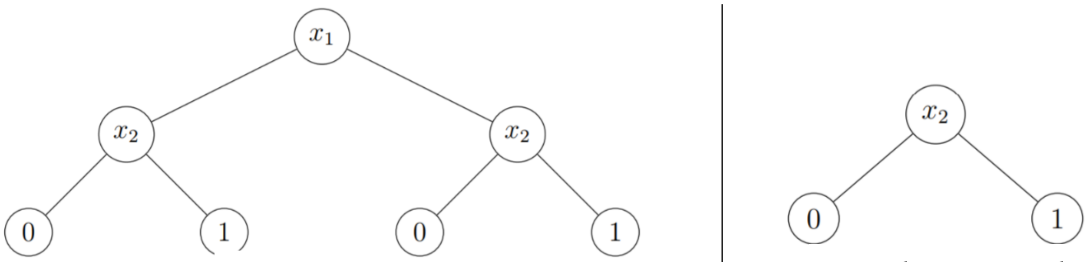
x_1	x_2	x_3	f(x_1, x_2, x_3)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Binary tree containing these variables and outputs:



In the tree above, at non-leaf nodes a 0 indicates traversal into the left subtree and a 1 indicates traversal into the right sub tree. From the above tree we can see that the subtrees of x_1 are the same and the subtrees of x_3 are the same. This means that regardless of the values of x_1 and x_3 they will provide the same output, and therefore they can be replaced by one of their subtrees.

The replacements are as followed:

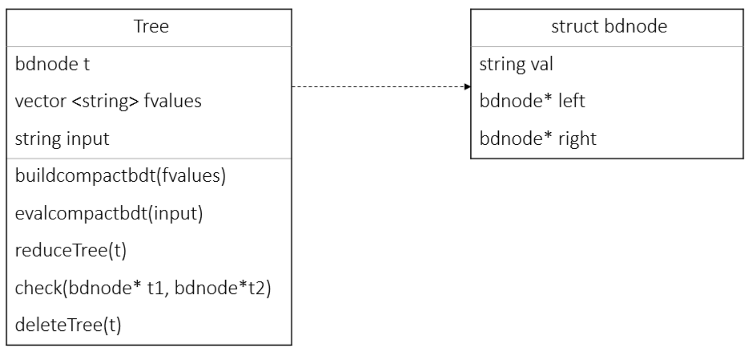


x_3 **removed** | x_3 **and** x_1 **removed**

As the final tree is smaller it can be evaluated faster and it consumes less memory.

Solution

The tree is built by a function named **buildcompactbdt** and the tree is evaluated by a function **evalcompactbdt**. The Unified Modelling Language (UML) diagram below indicates the program's member functions and objects.



Buildcompactbdt

The vector containing the 'true' outputs of the table is named fvalues. The buildcompactbdt function begins by creating a root node and labelling it x1.

```
bdt buildcompactbdt(const std::vector<std::string>& fvalues){
	bdt t = new bdnode();
	t->val="x1";
	bdt current = t;
	if (fvalues.empty()){
		t->val = "0";
		return t;
	}


	const int height = fvalues[0].length();
	const int size = fvalues.size();
	//below are the extreme cases for which the tree reduces to one node
with a 0 or 1
	if(size == pow(2,height)){
		t->val = "1";
		return t;
	}
```

Before building the complete tree, the function checks the extreme cases where all the outputs are 1, or all the outputs are 0, creating a tree of one node containing a 0 or a 1. This is seen above.

Nested for loops use the 'true' outputs to build nodes labelled x_2, $x_3..x_n$ towards the 'true' outputs (1s). If the loop creates a node x_2 to the right of x_1 because it found that the value of x_1 = 1, it will also create a node containing a "0" to the left of x_1. This creates a slightly minimized tree that can output 1 for all 'true' cases and a 0 immediately if it finds that the current input is not a substring of a 'true' input.

Total number of 'true' outputs.

Number of digits in each input (maximum height of tree).

If the value of the current node is 0, it must have a node on its left that is x_{n+1} (or 1 if it is a leaf node).

If the right child node does not exist yet, create it and set it to 0.

If it is a non-leaf node the left child node must be set to x_{n+1} (create a new node if left node doesn't exist)

If it is a leaf node the left child node must be set to 1 (create a new node if left node doesn't exist)

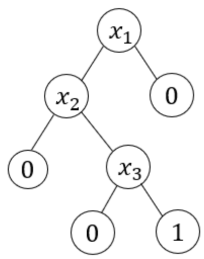
```
for (int i = 0; i<size;i++){
            for (int j = 0; j<height; j++){
                  if (fvalues[i].substr(j,1)=="0"){
                        if(current->right == NULL){
                              bdt temp = new bdnode();
                              temp->val="0";
                              current->right=temp;
                        }
                        if(j<height-1){
                              if (current->left ==NULL){
                                    bdt temp2 = new bdnode();
                                    current->left = temp2;
                              }
                              current ->left->val = "x"+ std::to_string(j+2);
                              current = current->left;
                        }else{
                              if (current->left==NULL){
                                    bdt temp2 = new bdnode();
                                    current->left=temp2;
                              }
                              current->left->val="1";
                              current = t;
                        }
```

The 'for' loop has an identical but opposite if statement if it finds that the value of the current node = 1. This means it will traverse into the right subtree instead of the left. This is seen below:

```
}else if (fvalues[i].substr(j,1)=="1"){
                        if(current->left == NULL){
                              bdt temp = new bdnode();
                              temp->val="0";
                              current->left=temp;
                        }
                        if(j<height-1){
                              if (current->right ==NULL){
                                    bdt temp2 = new bdnode();
                                    current->right = temp2;
                              }
                              current ->right->val = "x"+
std::to_string(j+2);
                              current = current->right;
```

```
                        }else{
                              if (current->right==NULL){
                                    bdt temp2 = new bdnode();
                                    temp2->val="1";
                                    current->right=temp2;
                                    current = t;
                              }else{
                                    current->right->val="1";
                                    current = t;
                              }
                        }
                  }
            }
      }
```

Following this 'for' loop, an example of a tree created for an input "011" is seen below:



However, before the pointer to the tree is returned by the **buildcompactbdt** function further manipulations are done to find whether subtrees of each node are the same, and if they are the tree is reduced.

```
      t = reduceTree(t);
      return t;
}
```

Beneath is the implementation of the reduceTree function.

```
bdt reduceTree(bdt t)
{
      if(t->val=="0"||t->val=="1") return t; //exit case (at leaf node)
      else
      {
            t->left = reduceTree(t->left);
            bdt leftChild = t->left;
            t->right = reduceTree(t->right);
            bdt rightChild = t->right;
            if (check(leftChild, rightChild)){ //check if both subtrees are
the same
                  t->val = leftChild->val;
                  t->left= leftChild->left;
                  t->right = leftChild->right;//left subtree assumes parent
node
                  delete leftChild;
                  deleteTree(rightChild); //right subtree gets destroyed
            }
      }
```

```
	return t;
}
```

The **reduceTree** function is a recursive function that traverses the tree and checks if the two subtrees of a node are the same. If they are the same, the function will replace the parent node with the left subtree and delete the right subtree. The purpose of this replacement is explained in the *Problem* section. **ReduceTree** calls another recursive function **check**, that checks whether two subtrees are the same.

```
bool check(bdt t1, bdt t2)
{
	if (t1 == NULL && t2 == NULL) return true;
	if (t1 == NULL || t2 == NULL) return false;
	return (t1->val == t2->val   &&
			check(t1->left, t2->left) &&
			check(t1->right, t2->right) );
}
```

The function simply traverses both subtrees and returns true if the values in the corresponding nodes are the same and if the leaf nodes all point to NULL. **ReduceTree** also calls a function **deleteTree** whose purpose is to recursively traverse and delete a subtree.

```
void deleteTree(bdt t)
{
	if (t == NULL) return;
	deleteTree(t->left);
	deleteTree(t->right);
	delete t;
}
```

Check and **deleteTree** provide **reduceTree** with the functionality to manipulate the tree.

Once the function exists **reduceTree**, **buildcompactbdt** can return the compacted tree with a pointer to its root.

Evalcompactbdt

The **evalcompactbdt** function takes an input, and outputs whether the tree contains a 1 or 0 for the specified input.

```
std::string evalcompactbdt(bdt t, const std::string& input){
	bdt temp = t;
	for (int i = 0; i<input.length();i++){
		if ("x"+ std::to_string(i+1)==temp->val){
			if(input.substr(i,1)=="0"){
				temp = temp->left;
			}else if (input.substr(i,1)=="1"){
				temp = temp->right;
			}
		}
	}
	return temp->val;
}
```

Ryan Mansoor
CID: 01224750

The outer 'if' statement compares the index of the input and the value contained in the node, and only carries out the statement if they are equal. This is because a reduced tree may not require all values of $x_1..x_n$ to evaluate the function. Without this 'if' statement a tree a root node x_2 may have an input value for x_1 traverse the pointer, leading to incorrect answers and errors. The inner 'if' statement traverses a pointer to the left of the tree if it detects a 0 and right if it detects a 1. The tree will reach the appropriate leaf node and will output a string containing "1" or "0".

Testing

Testing was carried out on the program by providing a range of inputs and finding the outputs. First, fvalues were instantiated with the desired 'true' outputs. Then **buildcompactbdt** builds the tree for these fvalues. A recursive function **countnodes** recursively goes through and counts the nodes in the tree, and outputs it as an indication of the size of the tree.

```
int countnodes(bdt t)
{
    if (t == NULL)
    {
        return 0;
    }
    else
    {
        return 1 + countnodes(t->left) + countnodes(t->right);
    }
}
```

Then, by using a recursive function named **verifybdt**, all possible binary combinations of the length of the input strings are processed through **evalcompactbdt** to verify if the outputs were as expected. **verifybdt** can be seen below.

```
void verifybdt(int length, std::string inputCase, bdt t)
{
      if (t == NULL)
      {
      }
      else if (length == 0)
      {
            std::cout << inputCase + "| " + evalcompactbdt(t, inputCase) <<
std::endl;
      }
      else
      {
            verifybdt(length-1, inputCase + "0", t);
            verifybdt(length-1, inputCase + "1", t);
      }
}
```

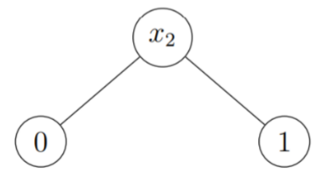
These two functions allow us to determine the tree has the correct size and the correct values.

Ryan Mansoor
CID: 01224750

Example:

Load fvalues with inputs "010, 011, 110, 111".

Expected Tree:



Run **buildcompactbdt**

Run **countnodes**

```
count = 3
```

Run **verifybdt**

```
000| 0
001| 0
010| 1
011| 1
100| 0
101| 0
110| 1
111| 1
```

Similarly, more inputs are tested and the results are given in the table below. Note that count is the number of nodes in the tree.

Test Title	**Inputs** (fvalues)	**Expected Outputs**	**Result**
Example Test	010 011 110 111	Count = 3 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Extreme Case 0	none	Count = 1 Verifybdt outputs 0 Node containing val = 0	Pass
Extreme Case 1	00 01 10 11	Count = 1 Verifybdt outputs 1 for all values	Pass
Test 1	1010101010 1110101010 1010001010 1010001110	Count = 45 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Test 2	0101 1010 0111 1110 1111	Count = 15 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Test 3	10111011 10110001 10101001	Count = 31 Verifybdt outputs 1 for fvalues and 0 for others	Pass

Test 4	0000 0100 0110 0111 1001 1010 1111	Count = 25 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Test 5	001 111 010 000 110	Count = 9 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Test 6	001 111 010 000 101	Count = 9 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Test 7	11101 11110 11011 10111 01111 11111	Count = 29 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Test 8	1001010100011 0111010100011 0100110010010 1110100011010 0101011101010 0101010101010	Count = 109 Verifybdt outputs 1 for fvalues and 0 for others	Pass
Test 9	10010101011 10011010101 10011100101 10100101010 10100101110 10100101000 11001101010 11110011010 11000001111 11000000011 11101010011 11100110110 01110110101 01111101110 01001101101 01101010110 00100100011 01010101010 01010110110 01100110011	Count = 255 Verifybdt outputs 1 for all 20 fvalues and 0 for others	Pass