Imperial College
London

Programming Assignment 3

ELEC40004 PROGRAMMING FOR ENGINEERS

Electrical and Electronic Engineering Year 1

Diego Van Overberghe
CID: 02030354

1. Introduction

One of the important questions I asked myself was whether to attempt to build a simplified tree straight away, or simplfying a full binary tree, as built in the last assignment.

In the end, I settled with simplification after constructing the tree, for several reasons. Firstly, this meant I could reuse code from the last assignment, code that has passed the required automated tests for the second assignment. Furthermore, I received constructive feedback regarding this code—which I used to make modifications that make the code easier to read and understand. Secondly, the examples of simplfication in the brief show simplification, starting from the fully developed tree, and I decided it would be best to follow the brief as much as possible.

2. The Algorithm

When a `BoolTree` is instantiated, the construction of the tree begins, using a vector of strings representing the rows of the truth table which are true to represent minterms.

The first thing to do is determine how many variables our tree must represent. The length of the string representing the first minterm is read. The length of this string gives the number of variables in the function.

Now, an empty tree of the correct depth is built, using recursive functions, ending all paths of the tree with `0`.

The next step of the algorithm is to set the paths of the tree which represents minterms to `1`, and, at this point, we have the tree as we had it at the end of the last assigment. We can now begin the simplification process.

The algorithm uses two simplification methods, which are each run at least until no more simplification is achieved, and at most once per variable of the binary tree.

The first simplification method looks for adjacent, identical outputs of the truth table. If we can find such a pattern, we know that the last variable's value has no influence on the output of the tree, and we can remove that variable from this path. Let's illustrate this using the first example from the brief.

As can be seen, the value of `x2` is irrelevant when `x1` is `false`. The algorithm detects this by spotting that both paths lead to identical outputs, `0` in this case. In effect we are realising that two adjacent lines of the truth table have the same output, regardless of the value of the last variable. See red in the below truth table. Having seen this, we can simplify because `x2` in this circumstance has no bearing on the function output in that branch of the tree. The simplification process itself involves setting the value of the parent node to the value of the children, and turning that node into a leaf.

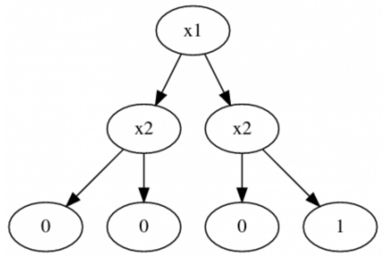

Figure 1: Tree from Assignment Brief

x1	x2	$f(\texttt{x1},\texttt{x2})$
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2: Truth Table for f

The second simplification method seeks for similar redundancy in variables, but instead looks for redundacy higher up in the tree. Consider the following tree diagram, also from the assignment brief, and notice that the value of `x1` has no influence on the final output of the logic function. In the truth table for the function, below, notice how the sections above and below the double horizontal line are identical, this means that the output of the function is independent of `x1` and we can purge it from the tree. Note that the nodes involving variable `x3` from the brief were simplified using the previous method to reach this stage. This time, the simplification itself involves discarding one side of the tree, and orphaning the intermediate node on the side of the tree that was kept. In this case, `x2` is orphaned, and `0` and `1` from the side that was kept become `x1`'s children.

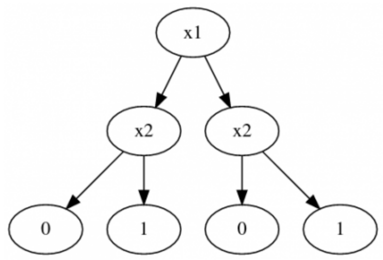

Figure 3: Tree from Assignment Brief

x1	x2	$f(\texttt{x1},\texttt{x2})$
0	0	0
0	1	1
1	0	0
8 1	1	1

Figure 4: Truth Table for f

Now, let's go through a worked example using the following boolean function:

$$f(\mathtt{x1}, \mathtt{x2}, \mathtt{x3}) = (A + B) \cdot \overline{C}$$

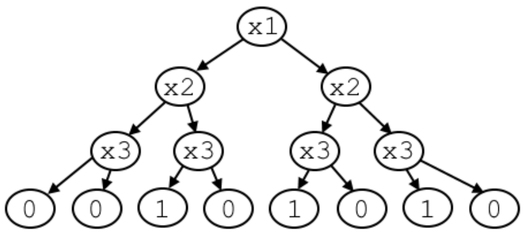

Figure 5: Binary Tree Representing f

Firstly, an empty tree of depth 3 is constructed. That is to say that all leaves has value 0. Then, leaves at the end of branches representing minterms 010, 100 and 110 are set to 1.

Simplification now begins, first checking for adjacency. We can see the first two leaves from the left share a parent and their values, so their parent's value is set to the child's value. On the second pass, there is no more simplification, so we move to the next simplification /*method.

It can be seen that the four right hand leaves are independent of `x2` and so we can eliminate that variable in that path, this is the second simplification method. The next pass does not simplify further and the final version is given below.

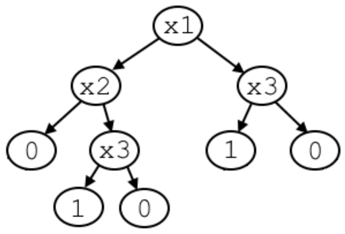

Figure 6: Simplified Binary Tree Representing f

3. Testing and Evaluation

An important part of testing is being able to visualise the tree. Quite some time was spent attempting to create a function that would print the tree, but I ended up simply using VS Code's tooltips when hovering over variables in debug mode. This allowed me to inspect the state of the tree, without writing a printing function, although it would still have been a luxury to have such a function.

I began writing the program using the provided examples as a basis of what kind of simplification to perform, so tested those function first, and after some work, reduced as much as they did in the examples provided.

Writing the actual simplification methods was not too complicated, but and example of an optimisation that was found was regarding the loops that govern how many times the simplification methods are applied.

Initially, the methods would run as many times as there are layers as this is theoretically the most times that they could be run. For instance, consider a tree where all leaf values are 1 and we have three variables. The first simplfication method (adjacentcy) would simplify this to simply 1 after three simplification passes. However, at the time I failed to consider the fact that if the simplification is unsuccesfull once, there is no point trying again. This can save quite a lot of time as the simplification functions are recursive and for bigger trees can significantly slow down execution.

To gain an idea of the performance of the simplification, a speadsheet was used to graph the reduction ratio. Given a number n of variables, the full tree will contain $2^{n+1} - 1$ nodes. This follows from the fact that a tree with one variable has 3 nodes, and adding a variable increases the tally by 4, then 8 and so on. With a fixed number of minterms, for instance 4, the reduction ratio increases as we add variables. Most likely this is due to the fact that when we add variables, and not minterms, we have more possibilty for dependence between the minterms and hence simplification opportunity. The following graph was generated by getting 4 random paths for minterms with 3, 4, 5, 6 and 7 variables.

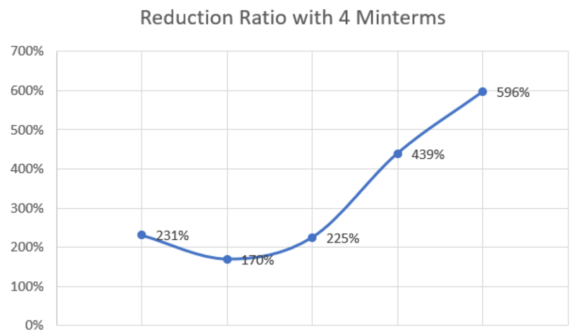

Figure 7: Graph Showing Reduction Ratio with 4 Minterms from 3 Variables to 7 Variables

4. Closing Words

Before closing, let's reconsider the inital decision that was made to simplify the tree after constructing the full version. Using this paradigm, signifcant nodal simplification has been achieved, especially when the number of minterms is smaller than the number of variables, as our Excel adventure demonstrated. However, doing this implies that there is one aspect that prevents further simplification, and that is the fact that the order of variables is set in stone. That is to say that `x2` cannot be the parent node of `x1` for example. Of course, this means that there will be scenarios where better simplification could be achieved if we had a way of finding which variable was the best to select when creating children. The `eval_bt` function is flexible as the index corresponding to the variable number will be read to decide to go left or right.

However, even considering the potential for greater simplification, there comes a point where one must consider tradeoffs. This report has to be short. My explaination of a simple alogrithm barely fits into the size constraint, and thoroughly explaining a much more complicated alogrithm, more likely than not, will be impossible in this limited environment.