The Algorithm

Compressing Inputs

The Inputs are compressed by working from the least significant bit to the most, the algorithm checks if all of the inputs are the same, with the exception of one variable, if this is the case then they are reduced into one input marked by an 'x' to denote the 'don't cares' e.g. "110" and "111" have a 'don't care' on x3, therefore this is equivalent to "11x" and we don't care whether x3 is a '1' or a '0' in this instance.

E.g. ("011", "010", "111", "110", "101", "100") => ("01x", "11x", "10x")
=>("01x", "1xx")

This compressed form expressed above is then used to build the Tree.

Highest Priority Variable

The order of variables in the Tree is based on the number of don't cares in a column. The priority is the least number of don't cares per variable; if two columns have the same don't care score then the variable with the largest grouping is preferred. For example, if variable x1 has 3 '1's' and 2 '0' inputs, variable x2 has 1 '1' and 4 '0' inputs then x2 has the largest grouping (i.e., 4 '0's') whereas x1 only has 3 '1's'.

The rationale for this logic is that we want the don't cares at the bottom of the Tree, so that they don't need to be expanded back into their original form so as to avoid collisions.

The reason for having grouping as the next priority filter is because this reduces the branching at the top of the Tree meaning that inputs travel down the Tree further together, consequently we generate less nodes.

Splitting Variables into a Zeros and Ones vector

A sub algorithm, using the order as stated above, splits the compressed inputs into two string type vectors, based on the highest priority variables value, the '1's' are grouped together and so are the '0's'. If any of the inputs has this value as an 'x' then it is added to both. Before being added to the vector the chosen variable's location in the input is swapped for an arbitrary value, this is to keep the order more closely related to its respective variable.

Building the Tree

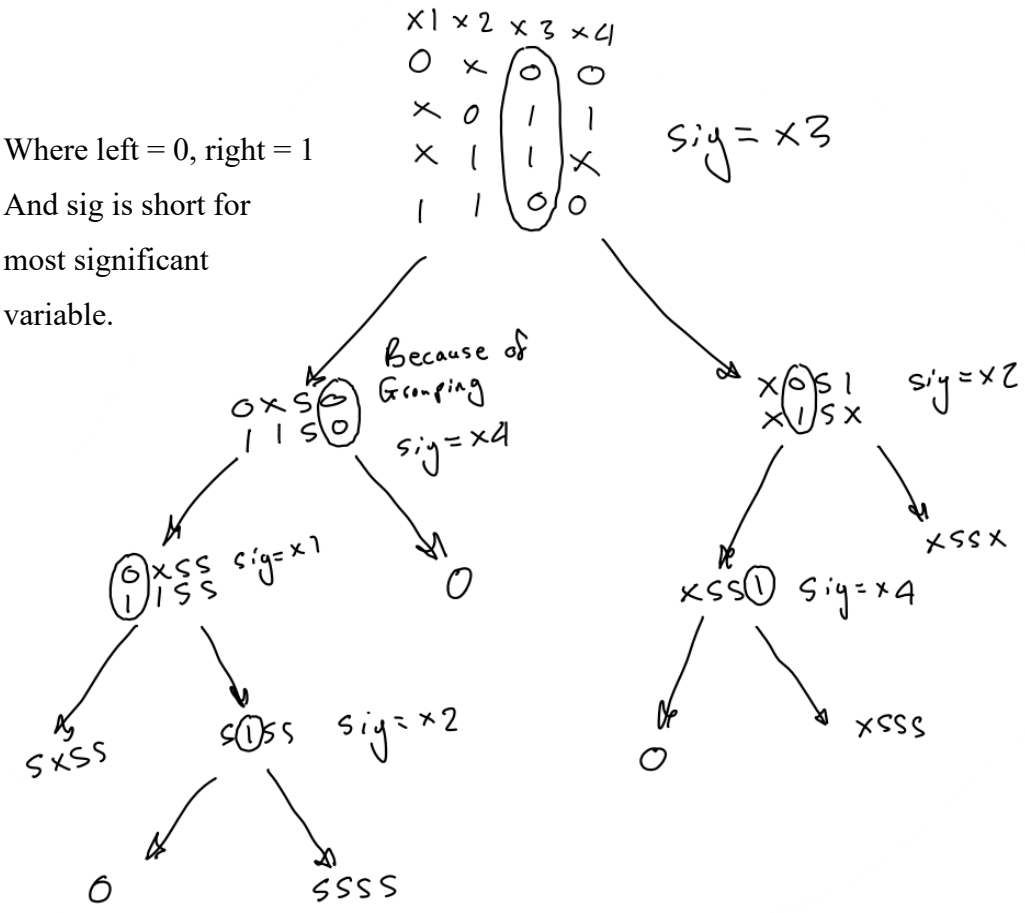
The Tree is built by firstly generating the compressed form, then working out the highest priority variable, next they are entered into an algorithm and the compressed inputs are split into two vectors based on the logic described above. The name of the node is created using the order plus one with a prefix of x, then a node is built with a value of the name and the left node is created by entering the vector of '0's' back into the algorithm with the new highest priority variable; the right is calculated in the same way but with the vector of '1's'. There are three conditions where the recursion is stopped; firstly, when there are only 'x's' left then a 1 node is placed; secondly if there are no string inputs in the '1's' or '0's' vector a 0 node is placed and finally if all the variables have been worked on then a 1 node is placed.

Worked Example

If we have the input:
(“0000”, “0011”, “1011”, “0110”, “0111”, “1110”, “1111”, “0100”, “1100”) in build_bt and “s” is the arbitrary value then the compressed form will be (“0x00”, “x011”,“x11x”, “1100”) then we get the following:

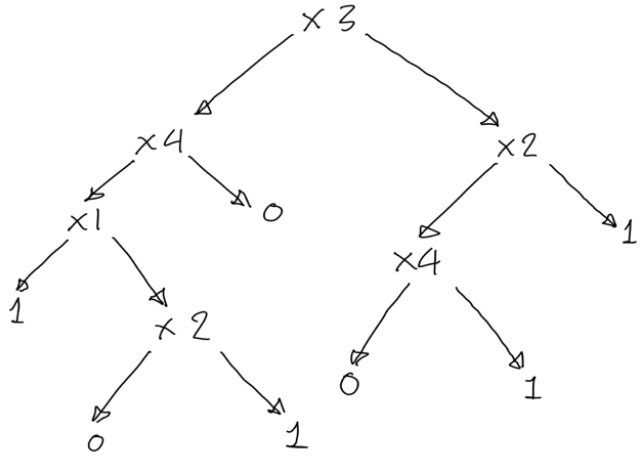
Where left = 0, right = 1

And sig is short for most significant variable.



Where only “s” and “x” are left in the string this is where the value 1 will be designated as you can see that when no strings are sent to a node the value is set to 0. Notice that when sig is the value 0 the string moves left through the Tree; when sig is 1 it moves right.

This is the resulting Tree (which is correct for the given inputs):



Testing and Evaluation

Testing

For testing I developed an algorithm to exhaustively test all possible Trees. The test gives every possible input and then generates the Tree, checking it using the provided function eval_bt. As eval_bt uses the value of the nodes, it in turn checks all the values and locations of the nodes, also checking that the "0" and "1" nodes point to two NULL nodes.

The Test algorithm increments from an n-bit binary number value of 0, where n is the number of variables, to the max value that can be stored in the n-bit number, storing them in a vector called all_high. Next a 2^n-bit binary number called current is created starting at zero where each bit represents a location in the first vector. If the value is one then that string is added to the Input/ones vector; if it is zero then it is added to a different vector named zeros.

E.g. 3 variables:
all_high = ("000","001","010","011","100","101","110","111")
Arbitrary value of current = ("01000110")
Input/ones = ("001", "101", "110")
zeros = ("000","010","011","100","111")

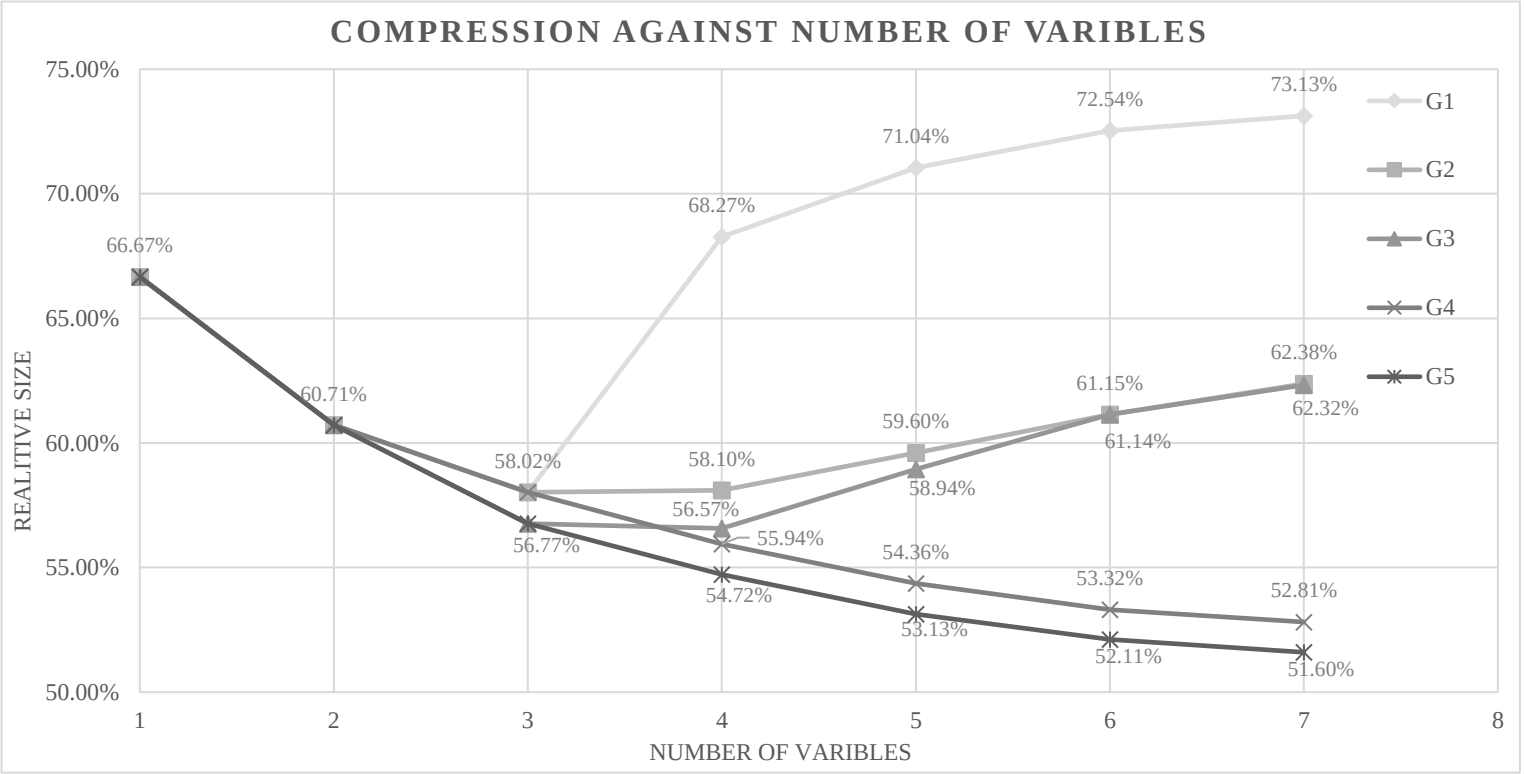
Then a Tree is built using the value of Input, this Tree is then checked for everyone and zero location by using eval_bt and iterating through ones and zeros checking for the correct output; outputting an error message if the test fails and a success message if it passes.

The available processing power allowed me to check every Tree for up to four variables - my test was successful for all inputs of four variables or less. For Trees of more than four variables I created a random sample of 1 million different Trees, I did this for Trees with five, six and seven variables which all came back successful. Consequently we can be reasonably certain that the code works as expected across the desired range.

	Number of Variables						
Generation	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	**2**	**4.25**	**8.70312**	**21.1623**	**44.7562**	**92.1222**	**186.481**
2	**2**	**4.25**	**8.70312**	**18.0115**	**37.5476**	**77.6599**	**159.058**
3	**2**	**4.25**	**8.51562**	**17.5373**	**37.1351**	**77.6478**	**158.926**
4	**2**	**4.25**	**8.70312**	**17.3423**	**34.2495**	**67.7101**	**134.677**
5	**2**	**4.25**	**8.51562**	**16.9626**	**33.4723**	**66.1848**	**131.579**
Uncompressed size	**3**	**7**	**15**	**31**	**63**	**127**	**255**

Evaluation

To evaluate the code I increased the functionality of the algorithm described above so that it also counted the number of nodes using n_nodes and added them up and worked out the average. Based upon the number of times the code ran, for 1-4 variables the values are exact, for 5-7 variables the compression is estimated using a random sample of 1 million Trees for each number of variables. Performing this and plotting the results as percentages of the uncompressed size we get the results in the table above, resulting in the following graph:



To develop the optimal algorithm, 5 generations of code where developed. In section Algorithm, this section describes generation 5 (G5). The most similar algorithm to G5 is G4, which was the same but without the addition of the grouping affecting variable priority; this meant that in G4 variables sometimes branched off from each other when they didn't need to which led to unnecessary nodes being created. G1 and G2 are similar except that the whole Tree has the same order and it is not unique in different branches, this leads to a less optimal solution, the program also needed to check the solution to make sure it worked, as it was prone to conflicts above 3 variables making it slower, in G1 it over corrected leading to poor compression at high numbers of variables fixed in G2. G3 improved on G1 & G2 as it tried every order and worked out the one that would result in the least number of nodes; however, it still had a set order for the whole Tree, leading to a reduction in performance for the higher numbers of variables. G3 matched the performance of G5 up to 3 variables as different branch order only has an impact when we have 4 variables or more.

In the graph above you can see a trend in G5; as the number of variables increases the Tree gets more compressed compared to its uncompressed size. This is because as the number of variables increases so does the opportunity for compression on the average Tree, however as the variables increase further it seems to have less of an effect on the percentage compression and it tends towards the 50% value.