# Introduction

A binary decision tree is a binary tree used to represent a Boolean function. In this assignment, each non-leaf node contains a variable of the Boolean function denoted **Xi** = {**X1**, **X2**, **X3**, …, **Xn**}, and each leaf node contain Boolean function output of value 0 or 1. The left path of a node represents value 0 of the variable in the node and the right path of a node represents value 1 of the variable in the node. When compacted, a binary decision tree becomes a compact binary decision tree. A compacting process is used to eliminate redundant information from the binary tree. The process is done by checking whether it exists duplicate information in the data structure and deleting such information.

# Algorithms

In the first example, compacting a BDT is done by deleting repetition of two adjacent leaf nodes under the same lowest level variable node(i.e. **Xi**, if the variables are in order), which might result in the tree not being fully compacted, where the leaf nodes have values “0101” from left to right after the first compacting process is done, and the variables from the root of the tree down are **X1** then **X2**. In this case, **X1** and **X2** have to be reversed to further compact the BDT, then the leaf nodes become “0011” from left to right and can be compacted again.

Dependency of the Boolean function output on the variables is an important concept. When compacting, provided that a Boolean function gives the same output no matter what value an instance of **Xi** has(namely, the Boolean function’s output doesn’t depend on this instance of the variable), the process is essentially deleting the node that contains that instance of the variable **Xi** and connecting the incoming node(parent node) to one of the said node’s outgoing nodes(child nodes). In a full binary decision tree, the first variable **X1** of a Boolean function has only one instance, but other variables have two or more instances.

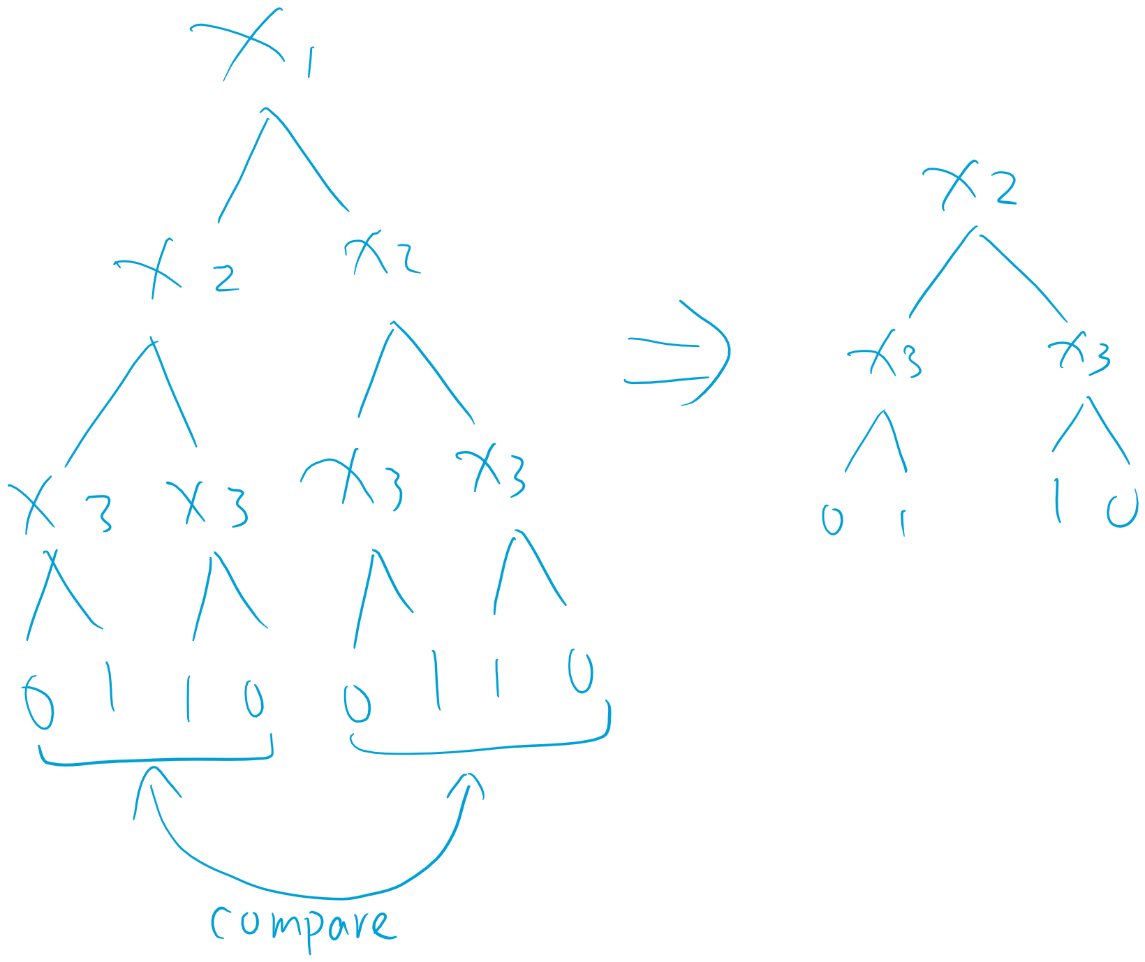
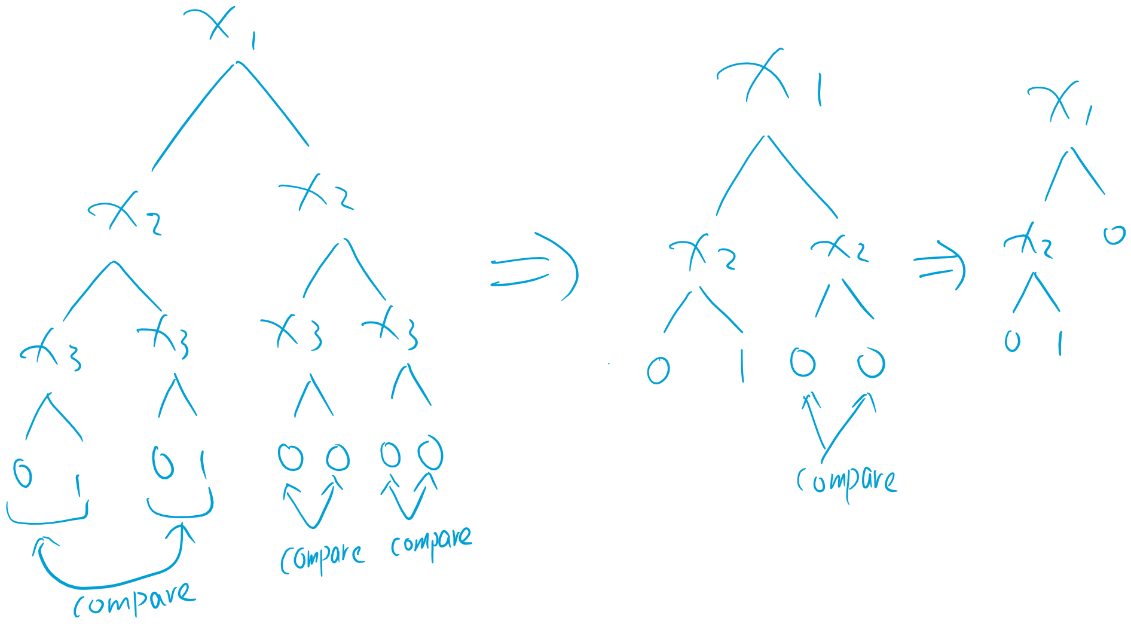
With the dependency concept, we can make sure a binary tree is compacted without omitting the case where Boolean function outputs do not depend on a variable that is higher up, i.e. not easily seen by comparing just two adjacent leaf nodes.

After deleting repeating subtrees in the tree, the order of the nodes can then be swapped to further compact the tree. An example will be shown below in the node swapping algorithm part.

## Delete repeated subtrees

Two ways of checking dependency(deleting repeated subtrees) have been considered. Both only need to be run once to delete all the repeated trees as subtrees with root nodes at each level are compared, and the comparisons are done from the bottom up with a level traversal.

Below are two examples of how the following two algorithms work in a high level abstraction:



Figure

Figure

### Comparing subtrees

The first one is comparing subtrees, if two subtrees of a node are the same, delete the node and the right subtree, connect the parent node to the left subtree. After repeated operation on all subtrees of the given BDT, the instances of variables left are variables that have an effect on the outputs of the Boolean function. This method requires traversal of a BDT and all of its subtrees(including leaf nodes). If a BDT has **n** variables, the number of subtrees and leaf nodes in total to traverse is 2n + 2n -2. And each subtree(other than leaf nodes) has more than one nodes. When comparing subtrees, the number of nodes to access(ignoring the operations needed to get to those nodes) is:

2n + [2n-1 \* (2+1)] + [2n-2 \* (2 \* (2+1) +1)] + [2n-3 \* (2\*(2 \* (2+1) +1))] + … + [21 \* (2n)]

=2n + [2n-1 \* (22 – 1)] + [2n-2 \* (23 -1)] + [2n-3 \* (24 – 1)] + … + [21 \* (2n - 1)]

=(n-1) \* 2n+1 + 2n – (2n-1 + 2n-2 + … + 21)

=(n-1) \* 2n+1 + 1

And if there are m subtrees to delete, the number of nodes to access(including the operations needed to get to those subtrees, but excluding the actual deletion of the nodes in the subtrees because it is the same for the two algorithms) is:

m + m = 2m

Because for each subtree to delete, we need to access the node with two repeating subtrees and the parent node of it, for when we want to connect its parent node to its left subtree.

This process includes some repeated access of values stored in memory, as there are only 2n+1 – 1 nodes in a full binary decision tree with n variables. But it allows us to save pointer values of the nodes that need to be deleted, thus eliminating the need to access those nodes from the top of the tree every time. The number of nodes to traverse is the same in best, worst and average cases given any n.

Extension: The deletion operation accesses the repeating subtrees from the lowest level of the BDT and works its way up. If we can get it work its way down it could be even faster since deleting a subtree from the top will likely delete some subtrees from the bottom, thus resulting in less access of the roots of subtrees. But the segmentation fault thrown when accessing the address of the stored roots of a subtree that is already deleted needs to be handled.

### Comparing leaf nodes

The second one is comparing the leaf nodes. This is similar to comparing subtrees but it doesn’t require comparing the variable nodes, which is not necessary considering we are only compacting once and the tree given at the start is a full tree. If we compact more than once, we need to deal with a partially processed tree after the first compacting, which is not full. If the tree given at the start is not full, then the leaf nodes can’t be compared straight away as some of them are at different level under different variables.

This method is different to the method used in the given example in a way that it not only compares one leaf node with its adjacent leaf node under the same lowest level variable **Xi**, but it also compares two leaf nodes with their adjacent two leaf nodes under the same variable **X(i-1)**, then four leaf nodes with their adjacent four leaf nodes under **X(i-2)** and so on, with the **number of nodes compared** going from 20 to 2n-1 for a BDT with n variables. The comparison process firstly traverses the BDT once to access all leaf nodes and store them all in a vector. With each **number of nodes compared**, every element in the vector(values in leaf nodes) is accessed once, which means the elements in the vector are compared with no overlapping. For example, if the **number of nodes compared** is 2, then in the vector, leaf node values in indexes 0 and 1 in the vector will be compared to values in indexes 2 and 3 respectively, in the next comparison, values in indexes 4 and 5 will be compared to indexes 6 and 7 respectively.

The locations of the instances of variables that are redundant are extracted using the path to reach from the root node. For example, if a leaf node value has an index of 5 in the leaf node vector of size 16(i.e. 4 variables BDT), then the path to reach to the variable that is directly above it will be “010”. For a bigger **number of nodes compared**, we will have to reach the relevant variable that is higher up the tree hierarchy(closer to root), so we only need to truncate the path to reach that variable. To determine “0” or “1”, we first add one to the index, so it goes from 1 to 2n. Then the size of the vector is divided by 2, if the index is bigger than the value then we obtain “1”, if the index is smaller than the value then we obtain “0”.

In total, there are **n** different **number of nodes compared** for a tree of height **n**, and for each of them, 2n elements are accessed in the vector. So, it requires n \* 2n operations for the comparisons. And in the worst-case scenario, Boolean function outputs are not dependent on any of the variables of a Boolean function. All variables will need to be deleted, the number of nodes to access(ignoring the operations needed to reach these nodes) is:

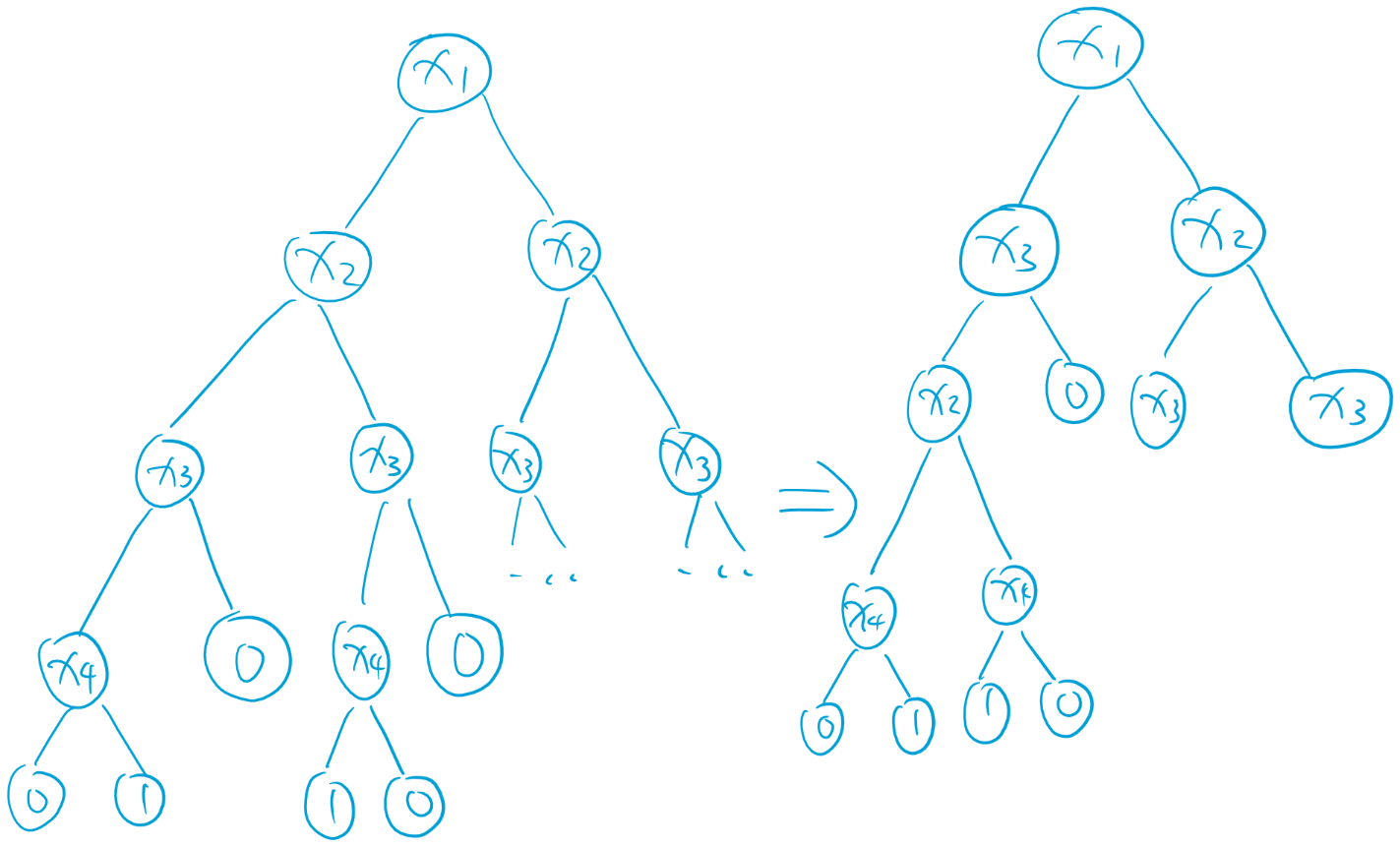
2n-1 + 2n-2 + … + 21 + 20

= 2n – 1

If there are m subtrees to delete, including the operations required to reach these subtrees but excluding the number of nodes to delete, the number of nodes accessed is bigger than the Comparing Tree algorithm because the root nodes of the subtrees we want to delete is always accessed from the top of the whole tree which requires significantly more operations.

I also notice that the above method accesses the variable nodes that are at the lowest level first and moves its way up. Going from top down theoretically saves time but is not worthwhile. Because if we do so, the binary tree is changed in a way that renders the stored paths to reach the variables useless. Unless we do something about it, such as not deleting the non-desirable variables but instead changing their values to “-1” and deleting them only after all paths have been used. However, this is not worthwhile.

## Node swapping

After the first compacting is done, the order of instances of variables can then be swapped to further compact the tree. One example is shown below, the order of the two **X3** nodes on the left can be swapped with the instance of **X2** node on the left. It is detected by first comparing either the left or(exclusive or) the right subtrees of the right and left child nodes of a node, if they are the same, the variable in the child nodes can be swapped with the variable in the node itself.

Note that the left and right subtrees of the right and left child nodes can’t both be identical after the first compacting process because they would have been compacted in the first place. This detection and swap operation swap the variables between two adjacent levels, when applied repeatedly, the order of the variables can be swapped into place to cut down the tree even more. Swapping of nodes from two levels that are further apart is accomplished by swapping between two levels multiple times.

## Node number and time evaluation

I tested the number of nodes after compacting the tree with both the delete repeated subtrees and swapping algorithm in place, as well as with only the delete repeated subtrees algorithm working with random input strings. And the number of strings inputted is half of the height of the tree. With the swapping, the operation is slower but not by much, because swapping is only operated on the tree after deleting all the repeated subtrees, which is much smaller than the original full tree in most cases. It is shown that the swapping doesn’t increase the number of nodes compared to compact without swapping.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Height of tree(number of random strings inputted = height/2) | Number of nodes(delete repeated subtrees and swap) | Time taken(s) | Number of nodes(delete repeated subtrees only) | Time taken(s) |
| 3 | 7 | 1.24\*e^(-5) | 7 | 1.22\*e^(-5) |
| 5 | 9 | 3.33\*e^(-5) | 9 | 3.05\*e^(-5) |
| 8 | 39 | 1.55\*10^(-4) | 41 | 1.49\*10^(-4) |
| 10 | 55 | 6\*10^(-4) | 57 | 6\*10^(-4) |
| 15 | 133 | 0.0265 | 137 | 0.026 |
| 20 | 259 | 1.22 | 297 | 1.17 |
| 23 | 339 | 10.1 | 377 | 9.8 |
| 25 | 411 | 43 | 453 | 41.5 |

## Unexplained behaviour

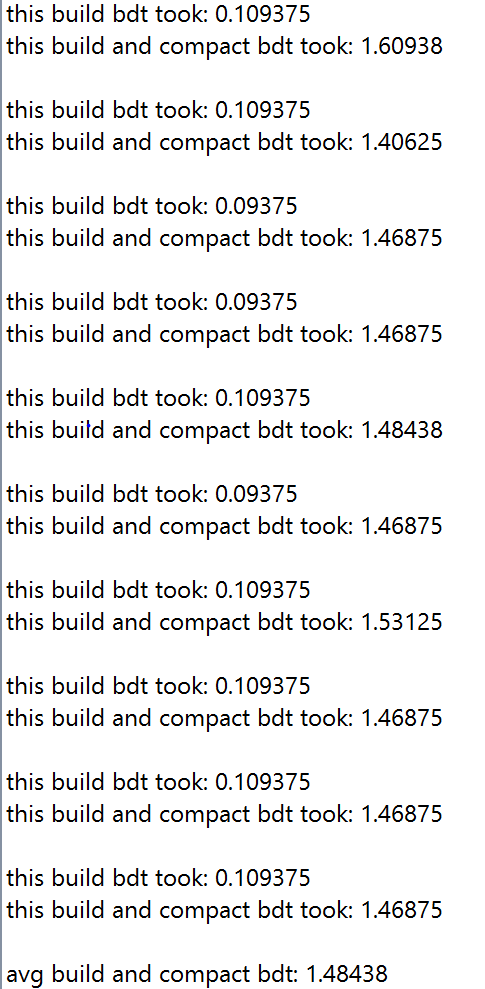
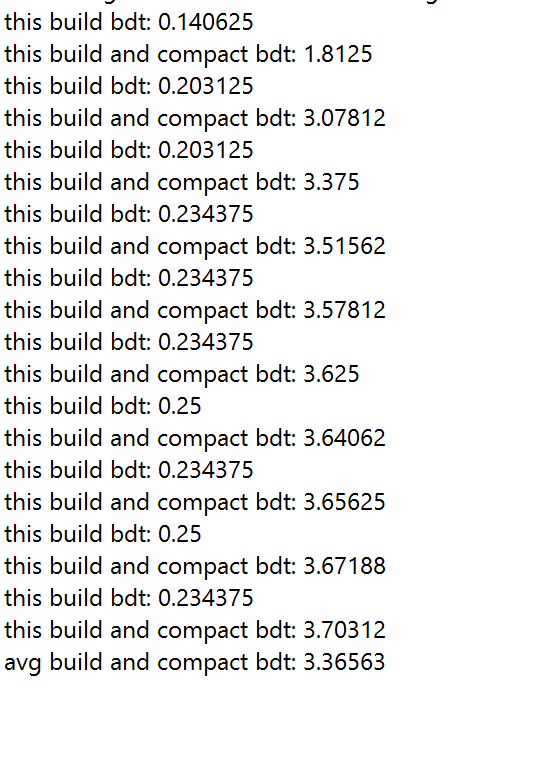
When testing my ***buildcompactbdt***, I realised the first iteration of ***buildcompactbdt*** was normal, but the iterations that follow become twice as long and stayed that way. There was a little change in the time spent for ***buildbdt***, but the little change in that seems to affect ***buildcompactbdt*** as well, which is not expected. The bdt built is the same for every iteration, why did ***buildcompactbdt*** also become longer? After attempts to debug this behaviour, I instead wrote new codes for ***buildbdt*** and didn’t change my ***buildcompactbdt***, and it works normally after that. Does it have anything to do with memory management of my computer?

Figure 3 Original buildbdt

Figure 4 New buildbdt