Experimental Evaluation

All the test is performed on Win10 Ubuntu, with average 13.5%±0.5% CPU (i7-7700K) with DDR3 RAM. The time shown is the time taken to execute buildcompactbdt() function.

In all of test below, unless stated otherwise, is done by taking integer set[12] as input, treated as minterm, generate fvalues and run buildcompactbdt(), and finally testing correctness by iterate all the possible combination and compare if "1" only get returned when it is a member of fvalues.

Is it worth to implement functions to find essential prime implicants?

I used my code without step 3 mentioned in Description of Buildcompactbdt algorithm to shown number of node produced by tree with only prime implicants as control group.
Used online resources[13] to generate the essential prime implicant, inject to step 4, to see the performance with essential prime implicants, as experimental group.

The first set I tested is with minterm used in previous work through 0,1,2,5,6,7,8,9,10,14 which forms a fvalues with size 10 and each fvalue string is made by 4 chars.
The result from control group, without reduce term to find essential prime implicants is 17 nodes. There is only one form of essential prime implicant, which is $\overline{x_2}\,\overline{x_3} + \overline{x_1}x_2 + \overline{x_4}x_3x_1$, by inject to step 4, creating a tree with 13 node. Both tree are drawn in the table below:

Use prime implicants	Use essential prime implicants
0 x1 1 x2 x2 2 x3 1 x3 x3 3 1 0 1 x4 0 x4 4 1 0 1 0	0 x1 1 x2 x3 2 x3 1 x2 x4 3 1 0 1 0 1 0 4
17 nodes	13 nodes

The second set I tested is with minterm[14] 40,51,37,38,27,50,14,5,42,53,61,20,21,16,15,22, 63,30,25,47,6,19,17,23,13,56,26,9,52,45,24,12,1,7,62,41,28,32,58,8,0,29,39,35,43,60,57,59,1 1,31,3,44,54,46,36,4,48,2,34,18, which forms a fvalues with size 59 and each fvalue string is made by 6 chars.
The result from control group, without reduce term to find essential prime implicants is 29 nodes, the result from experimental group with all 12 different form of simplification measured, it had max of 35 nodes, minimum of 31 nodes, median of 35 nodes.
Interestingly for this case, by find essential prime implicants increases the number of node required.

The last set I tested is with minterm0,2,4,8,9,10,11,12,16,21,22,23,24,25,26,27,28,29,30, 31,32,39,40,44,46,49,51,52,61,63which forms a fvalues with size 29 and each fvalue string is made by 6 chars.
The result from control group, without reduce term to find essential prime implicants is 61 nodes, the result from experimental group with all 2 different form of simplification is 57 nodes and 63 nodes.

As shown, from 3 set of data above, step 3 does not always lead to reduced to node and due to time constraint, it is not implemented.

---

[12] All testing data are generated randomly from random.org by using integer set generator.
[13] Frédéric Carpon (2012) *quine-mccluskey-frederic-carpon-implementation* http://quine-mccluskey-frederic-carpon-implementation.e-geii.eu/ [29th/04/2018]
[14] Randomly generated from random.org https://www.random.org/integer-sets/

Memory and time evualtion and optimisation

Before I optimise the step 2 of Buildcompactbdt (Quine–McCluskey), the runtime of step 3 is heavily depends on the size of the input fvaules, it will take significant part of HEAP after fvalues size reaches 64 and above, also the runtime is often greater than 1 minutes. Hence, I decide to debug the find out how can I improve it.

Here is the performance before optimise, with input of minterms from 0 to 31, generate a fvaules of size 32 and each fvalue string have 5 chars.
Before the optimisation, the Buildcompactbdt function will take 15.34 seconds to run, uses 34.4MB on HEAP maximum, but there is no memory leak, as all struct created by new are deleted before pointer go out of scope.
After the optimisation, with the same input, Buildcompactbdt functions only takes 104.1 ms to construct the tree, with 101kB on HEAP at maximum, with 99.3% reduction on time and 99.7% reduction on HEAP.
See second row of table below for original output, first 5 lines are my debug code in main(), after that are information provided by valgrind.

I start this evaluation by identity the part where takes most of time and memory happened at "later pass" described in flowchart Figure 1. By using "cout" to show intermediate process in this section, I found the problem is due to duplicate entry to each of intermittent struct tempList.
To solve this problem, I changed the code in add function of struct intermittent to perform additional check of duplicate, shown on third row of table below.

Before optimsed	After optimesed
the tree has 1 nodes compare to tree in ass1 require 63 node reduced by 98% the buildcompactbdt function takes 15360.3ms ==414== ==414== HEAP SUMMARY: ==414== in use at exit: 72,704 bytes in 1 blocks ==414== total heap usage: 933 allocs, 932 frees, 34,356,724 bytes allocated ==414== ==414== LEAK SUMMARY: ==414== definitely lost: 0 bytes in 0 blocks ==414== indirectly lost: 0 bytes in 0 blocks ==414== possibly lost: 0 bytes in 0 blocks ==414== still reachable: 72,704 bytes in 1 blocks ==414== suppressed: 0 bytes in 0 blocks ==414== Rerun with --leak-check=full to see details of leaked memory ==414== ==414== For counts of detected and suppressed errors, rerun with: -v ==414== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)	the tree has 1 nodes compare to tree in ass1 require 63 node reduced by 98% the buildcompactbdt function takes 103.015ms ==420== ==420== HEAP SUMMARY: ==420== in use at exit: 72,704 bytes in 1 blocks ==420== total heap usage: 597 allocs, 596 frees, 100,756 bytes allocated ==420== ==420== LEAK SUMMARY: ==420== definitely lost: 0 bytes in 0 blocks ==420== indirectly lost: 0 bytes in 0 blocks ==420== possibly lost: 0 bytes in 0 blocks ==420== still reachable: 72,704 bytes in 1 blocks ==420== suppressed: 0 bytes in 0 blocks ==420== Rerun with --leak-check=full to see details of leaked memory ==420== ==420== For counts of detected and suppressed errors, rerun with: -v ==420== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0) ben@DESKTOP-6JA49PB: /ass/algdv/2$

minterm=minterm&baseNot;
int pcount=popcount(minterm);
 	
minterms[pcount].push_back(minterm);
used[pcount].push_back(false);
	minterm=minterm&baseNot;
	int pcount=popcount(minterm);
	
	if(notRepeated[15](minterms[pcount],minterm)){
	used[pcount].push_back(false);
	}

[15] notRepeated(A,B) function will return Boolean whether B is already in A, and push back B to A if not exist

Testing on case where can't be merged

One special case that I have experimented is when there are not term to merge to produce don't care, such that the only factor is going to make a difference is the order of the node chosen.

Below is the example of how node will be chosen for 00001, 00010 and 10000, 01000.

Term $x_1x_2x_3x_4x_5$	with condition i and ii[16]	All 3 condition i,ii,iii
00001 00010	0 x1 1 x2 0 2 x3 0 3 x4 0 4 x5 x5 5 0 1 1 0 13 nodes in total	0 x1 1 x2 0 2 x3 0 3 x4 0 4 x5 x5 5 0 1 1 0 13 nodes in total
10000 01000	0 x1 1 x2 x2 2 0 x3 x3 0 3 x4 0 x4 0 4 x5 0 x5 0 5 1 0 1 0 19 nodes in total	0 x3 1 x4 0 2 x5 0 3 x1 0 4 x2 x2 5 0 1 1 0 13 nodes in total

As shown from the table above, with condition ii, choice the node with most unbalanced 1 and 0 applies, it create minimum node consistently. Whereas if after condition i just choice numerically by condition iii, the number of nodes is heavily depend on the order of the input, leading to 46% increasement on number of nodes in this case from 13 nodes to 19 nodes.

Testing on large fvalues

Below is the result for 10,000[17] terms of fvalue. From in the range of 000000000000000 (minterm 0) to 1111111111111 (minterm 2^{15}-1).

Minterm Set reference number[18]	Number of nodes with condition i ii	Number of nodes with condition i ii iii	Number of nodes without simplification
Set 1	16861	16857	65535
Set 2	16659	16639	65535
Set 3	16815	16875	65535
Set 4	16991	16915	65535
Set 5	16817	16787	65535
Average Nodes	**16828.6**	**16814.6**	**65535**

As shown, there is a average of 74.3% of reduction by using my method compare to the node without simplification. By using condition ii, before choosing node numerically will reduce number of node further by 14 on average, although there are times where by using only condition i ii performs, but it is just luck.

[16] As condition described in step 4 of section Buildcompactbdt
[17] Max number of integers allowed to generate by random.org
[18] The Minterm Set used in the test bench can be find from end of appendix

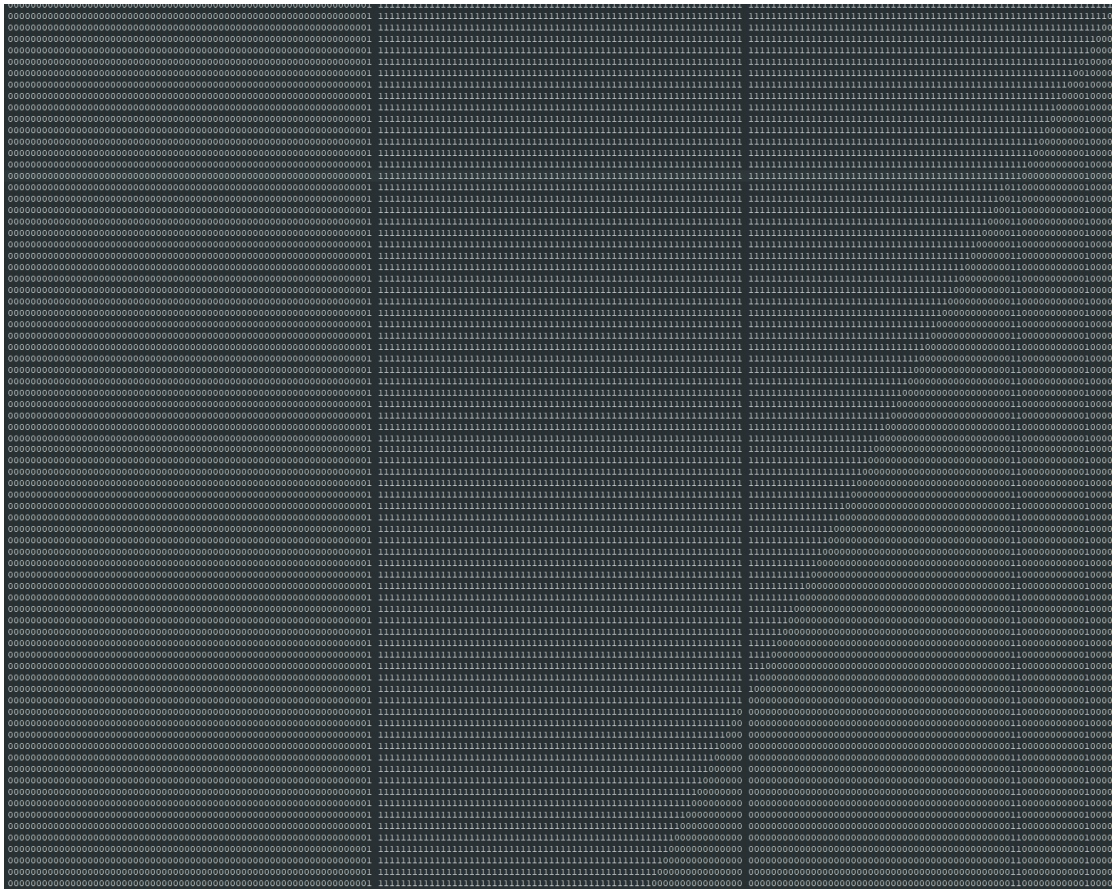
Manual test on fvalue with length greater than 64

Due to longest integer supported by c++11 is 64bit, for fvalue longer than 64 bit, automated test can't be easily performed. Hence manual test is done to confirm that it still functioning correctly.

The test contain 5 input fvalues each consist of 129 chars as shown below

```
10001010101000000000111111111111110000000000000000011111111111111111111111000000000000000011111111110000000000000100000000000000
10000010101000000000111111111111110000000000000000011111111111111111111111000000000000000011111111110000000000000100000000000000
10001010101000000010011111111111110000000000000000011111111111111111111111000000000000000011111111110000000000000100000000000000
10001010101000001000111111111111110000000000000000011111111111111111111111000000000000000011111111110000000000000100000000000000
10001010101000001100111111111111110000000000000000011111111111111111111111000000000000000011111111110000000000000100000000000000
```

To check if the tree is correct, all 5 string in fvalues are evaluated and confirmed to return 1. To check all other string will return 0, tree is printed on screen and verified the tree is the same as I expected.



The picture shown above is the terminal output of how node is considered, just as expected, in general node is picked from left to right (from x_0), where bit turn from 1 to 0 meaning it was in consideration. It's interesting to note that bit 5 and 17,18 had stayed 1, it is due to condition ii, due to x_5 and x_{17}, x_{18} can be merged while calculating the implicant.

Effect of speed on how Internal representation of minterm is implemented

For the cause of general purpose, the minterm is internal represented as vector in the end, to allow simplification of trees with each of fvalue string longer than 64 bit, which can not otherwise performed with a single word.

However, it creates overhead for strings that is shorter or equal to 64 bit, which leading to longer run time. To understanding the effect, a test is done, by generating fvalues string from 00000000000 (minterm 0) all to 1111111111111 (minterm 2^{13}-1), total of 2^{13} fvaules. With the result shown in the table below, is about 10 times slower, but consider the number of term access during the MQ algorithm growth exponentially (NP-hard) its affect will become more significant if number of term increase, such as 2^{32} terms.

Term as vector	Term as single 64-bit word