Evalcompactbdt

This function takes two arguments, the first is the root of the tree, second is a string for input to evaluate.

In the binary decision tree, except for the leaf which will either be "1" or "0", the rest of node is in the form of "x*num*", where *num* is a index for a char at position (num-1). When evaluate the tree, if the string[num-1] is "0", it goes to left branch, otherwise, goes to right branch.

For example, continues example above, when evaluate string is "0100"($x_1x_2x_3x_4$), correspond to internal minterm 0010 ($x_4x_3x_2x_1$). Check it is not leaf node("1" or "0"), remove first char of string "x1", get 1, evaluate char of input string at index 0, which is 0, evaluate left node.
Check it is not leaf node, remove first char of string "x2", get 2, evaluate char of input string at index 1, which is 1, evaluate right node.
Check it is leaf node, output the content, returns "1".

Internal representation of minterm in C++

For flexibility of the code, all the algorithm and function implemented below uses same abstract data structure called “term”, which is treated as a vector consist of multiple word (integer) of constant length. All mask, minterm, implicant are implemented by using term.

The reason for use word (64 bit integer) is because of efficiency. Another way to implement such data structure is using std::vector<bool>, which although occupies same space (stores multiple bool in same word), but it requires mask when read and write and it does not support read and write multiple bits at a time, such as XOR of two 64-bit word.

Bitwise operator (~,|,^,&) had been overloaded, to allow readable and more maintainable code.

```
term operator~(const term& v1);
term operator&(const term& v1,const term& v2);
term operator|(const term& v1,const term& v2);
term operator^(const term& v1,const term& v2);
```

Below are how current implementation of term is defined. Each word is a 64 bit unsigned integer (unsigned long long), to form a vector of any length required by the fvalues.

```
#define wdlength 64
#define wdMax 0xFFFFFFFFFFFFFFFF
typedef uint64_t wd;
typedef std::vector<wd> term;
```

Impletion of Buildcompactbdt in C++

Generate minterm

This step is performed by function call genMinterm(fvalues,minterms) ,where fvalues is a vector of string of same length and minterms is vector of minterm, both pass by reference.

The algorithm can be represented by C++ like pseudo code below:

```
int numwd=fvalues[0].size() / wdlength;
int lastlen=fvalues[0].size() % wdlength;
For( str : fvalues){
        int p=fvalues[0].size()-1;
        term minterm(numwd+1,0); //init minterm to (numwd+1) word consist of 0

        //Most significant word of the term
        wd word=0
        Repeat lastlen times{
                if (str[p]==1){ word = word *2+1 } else { word = word *2};
                p--;
        }
        minterm[numwd]=word

        //All other less significant word of the term
        While(p>0){
                word=0;
                Repeat wdlength times{
                if (str[p]==1){ word = word *2+1 } else { word = word *2};
                p--;
                }
                minterm[(p+1) / wdlength]=word
        }
}
Minterms.push_back(minterm)
```

Find prime implicant[9]

Quine–McCluskey algorithm is used in this step, also by taking inspiration from (Jadhav Vitthal, 2012)[10], a structure is made to recursively compute this. The structure is called intermittent.

At the start the structure intermittent is populated by minterms with same mask, normalised by doing a bitwise AND with the mask and then separated into smaller group according to number of 1's to contain, the mask has same length as minterm. For example, $\mathrm{X}_1\overline{X_4}$ and $\mathrm{X}_1\mathrm{X}_4$ will be populated to same instance of the structure, as they both have mask of 0110, but they will be speared to 2 group with minterm represented by 0001(one 1's) and 1001(two 1's) respectively.

The structure intermittent had a function called compareAll, which will compare neighbour groups of 1s iteratively by Quine–McCluskey algorithm, and create new instances of the structure by using "new" on HEAP and append it to a map with the common mask as key. For the part that did not use to create new instance, it is push backed to a vector of another structure implicant, as it is prime implicant.

Structure implicant only consist of two data field with no functions, first is the mask and the second is minterm, for example $\mathrm{X}_3\overline{X_4}$ will have mask 0011 represent for "don't care" of term $\mathrm{X}_1\mathrm{X}_2$ and minterm 1000, where least significant 2 bit is forced to 0 due to the mask.

Pseudo code is not provided for this section, as it involved with multiple struct with their public data and function. A flowchart is used to given a overview of how these struct linked together to execute efficient Quine–McCluskey algorithm.

[9] ALL pseudo code from this sub-section onward, unless otherwise stated, treat term as a single integer, a simplification of how the real code is implemented.
[10] Corporate author (2012) *Modified Quine-McCluskey Method* https://pdfs.semanticscholar.org/650a/ae0c3e59bdd12e7b22158162166e4437949e.pdf [29th/04/2018]

The chart below shows how these structures linked together to find prime implicant.

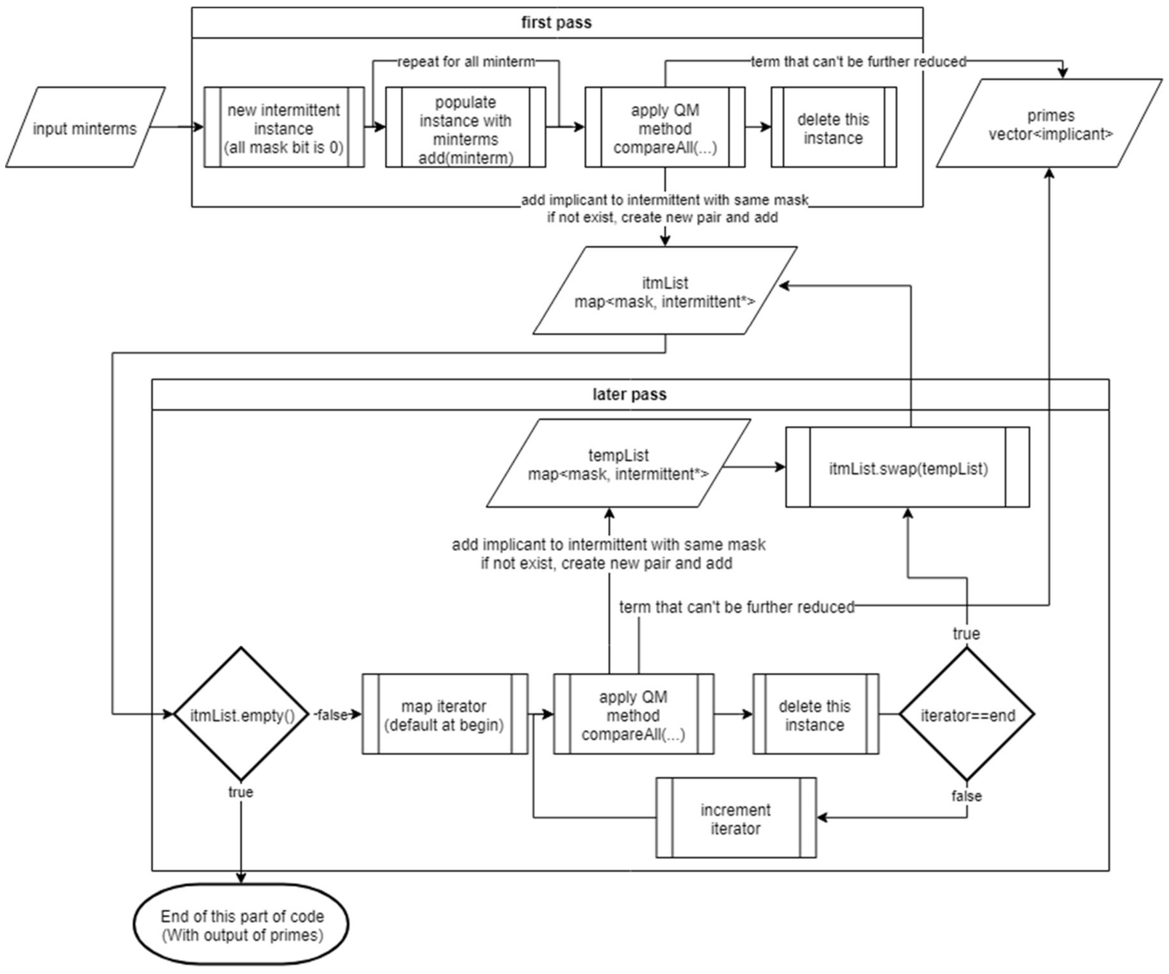
first pass
input minterms
new intermittent instance (all mask bit is 0)
repeat for all minterm
populate instance with minterms add(minterm)
apply QM method compareAll(...)
delete this instance
term that can't be further reduced
primes vector<implicant>
add implicant to intermittent with same mask if not exist, create new pair and add
itmList map<mask, intermittent*>
later pass
tempList map<mask, intermittent*>
itmList.swap(tempList)
add implicant to intermittent with same mask if not exist, create new pair and add
term that can't be further reduced
true
itmList.empty()
false
map iterator (default at begin)
apply QM method compareAll(...)
delete this instance
iterator==end
false
increment iterator
true
End of this part of code (With output of primes)

Figure 1

As shown from the chart, in the first pass, an intermittent is newed and constructed, then all the minterm produced by previous step is added to the struct, compareAll(...) function is called, produce an output of map itmList and primes for primes implicant. The current struct is than deleted to save HEAP space and avoid memory leak.

In all the later pass, itmList is firstly checked if it is empty. If true, means no more term need be further simplified and exist to next part, if false, iterate through the map for all the intermittent struct. For each intermittent struct, call compareAll(...) function but put output map to itmList, and delete the current instance and iterate to next struct in the itmList. After iteration finish, swap between tempList and itmList, ready for next pass. The pointer in itmList which swapped to tempList is no longer needed and delete automatedly as went out of scope.

The reason for the swap is feasible is due to implicant can only merge when mask is the same and its only different by one bit, but mask in tempList and itmList will never be, as tempList is like children of itmList, have 1 more bit of mask.

generate the binary decision tree

By using the algorithm introduced in Description of Buildcompactbdt from previous section. A recursive function can be used to implement this algorithm, with base cases of all don't cares and empty content.

Function recTreeConstructor(node, primes, nodeRemains) is used to implement the function. Where node is the root of the sub tree that is currently constructing, primes are the (essential) prime implicant produced by previous part, nodeRemains are the bit indicator of either bit had been used.
For example, for fvalue consist of 4 chars, if bit x_3, x_2 had been used, but x_4, x_1 haven't, the nodeRemains will be 1001 (by internal representation $x_4x_3x_2x_1$).

Structure bitCount is used, is has function of construct(...), which takes vector of prime implicant (primes), count number of set for each bit of the don't care (primes.mask) and count number of ones for each bit of the minterm (primes.minterm).
It has a function called getMask (digit), which returns the sum of set bit for that digit. And function called getTermLeast(digit), which returns the minimum between sum of ones and sum of zeros for that digit, such that the smaller the number returns, the more unbalanced ones and zeros it has.

Below is C++ like pseudo code used to describe this function.

```
If(primes.size()==0){
        //2 base case
        Node.lable="0" //first base case
        return;
}else{
        For(prime :primes){
                maxMaskCount=popcount(nodeRemains)
                If( popcount(prime.mask)== maxMaskCount){
                        Node.lable="1" //second base case
                        return;
                }
        }
        // heuristic method to decide which root to choose
        bitCount cnt;
        cnt.constrct(primes, ...);
        unsigned int  mincount= (int)-1 //use type casting to set mincount to highes possible

        for(digit in nodeRemains){
                if(cnt.getMask(digit)< mincount){
                        // condition i
                        mincount= cnt. getMask (digit)
                        optimalDigit = digit
                }elseif(cnt.getMask(digit)==mincount && cnt.getTermLeast(digit)<mostUnbalanced){
                        // condition ii
                        mostUnbalanced= cnt.getTermLeast(digit)
                        optimalDigit = digit
                }
        }
        Node.labe=”x”+ optimalNode

        // prepare for the recursive call
        For(prime :primes){
                If(prime.minterm[optimalDigit]==0){
                        // as described in algorithm description, have 2 possible cases
                        If(prime.mask[digit]!=0){
                                tempPrime=prime
                                tempPrime.mask= prime.mask & (~optimalMask[11])
                                leftPrimes.push_back(prime)
                                rightPrimes.push_back(prime)

                        }else{
                                rightPrimes.push_back(prime)
                        }
                }else{
                        leftPrimes.push_back(primes[i]);
                }
        }
                // start recursive call
        recTreeConstructor(node->left,leftPrimes, nodeRemains& (~optimalMask))
        recTreeConstructor(node->right,rightPrimes, nodeRemains& (~optimalMask))

}
```

[11] optimalMask is a mask where only optimalDigit bit is set to 1, rest is 0. ~ means bitwise NOT, & means bitwise AND, it is used to normalise the optimalDigit bit on mask to 0

Impletion of Evalcompactbdt in C++

From algorithm introduced in Description of Evalcompactbdt, below is the simple implementation.

```
While(node is not 0 or 1){
        If(str[node.val]!=0){
                node=node.right
        }else{
                node=node.left
        }
}
Return node.val
```

Where node.val is the content in the node, str[i] is the i^{th} char in string str indexing from 0.