EECS 478 Fall 2014 Project No. 1



### The University of Michigan, EECS Department EECS 478: Logic Synthesis and Optimization, Fall 2014

John P. Hayes Shaobo Liu

Project 1: Binary Decision Diagrams

INSTRUCTIONS:

Distribution Date: Wednesday September 24*th*, 2014 Due Date: Friday October 17*th*, 2014 at 23:55

Submit Online on CTools *→* Assignments *→* Project 1

* You must abide by the Engineering Honor Code. All work submitted should be work done by

you and you alone. If you are unsure about the level of collaboration, please consult Hayes or

Liu.

* Make sure your code is indented properly, readable and well commented.
* Your code portion should not output anything at the display, unless explicitly men- tioned. Verbose codes will be penalized.
* When you are finished, collect the following two deliverables:
  + project1.cpp
  + README *{*uniquename*}* where uniquename is your uniquename.
* Create a tar ball with *only* these files. To do this:
  + Create an empty directory called EECS478P1 *{*uniquename*}* , where *{*uniquename*}*

is your uniquename.

* + Copy all your deliverables into this directory.
  + Type tar -cvf EECS478P1\_*{*uniquename*}*.tar EECS478P1\_*{*uniquename*}*.
  + Type gzip EECS478P1 *{*uniquename*}*.tar .
* There are 100 (+10 bonus) points possible for this project.

# Introduction

The goal of this assignment is to strengthen your understanding of binary decision diagrams (BDDs). After finishing, you should have an intuitive sense of how BDDs represent Boolean functions, and how operations are performed on them.

You will implement several operations on BDDs in C++. This includes the *apply* function, taking negative and positive cofactors, Boolean difference,probability calculation, and checking P- equivalence.

# Provided Files

The provided tar ball has the following files.

* + bdd node.h and bdd node.cpp
  + smart pointer.h
  + bdd tables.h and bdd tables.cpp
  + operation.h and operation.cpp
  + Bool expr.h and Bool expr.cpp
  + Bool expr parser.h and Bool expr parser.cpp
  + main.cpp
  + project1.h and project1.cpp
  + Makefile

The *only* file that you will be modifying is project1.cpp.

## bdd\_node.h and bdd\_node.cpp

The bdd node class provides the building block for BDDs, along with some useful operations. The definition of bdd node is provided in bdd node.h, and the implementation of the member functions in bdd node.cpp.

Each bdd node object has a variable that it splits on, and a pointer to two other bdd nodes, the positive and negative cofactors with respect to its variable. There are two static (exist once for the entire class) bdd nodes that represent the two terminal nodes **1** and **0**. Every BDD will have these two nodes at the bottom of its structure. You can tell whether or not a given node is one or zero using the is one() and is zero() member functions.

Generally, when working with a BDD, you have a pointer to a bdd node that is the head of a binary tree of bdd nodes that represent a certain function. Since BDDs provide a canonical representation of Boolean functions, no other structure will exist in another in the same or different

structure to represent that same function. Each child in turn represents another function, the positive and negative cofactors with respect to the variable that the node splits on. So, if you were to have two

distinct BDDs, one representing the function *F*1 = *a ∧ b ∧ c*, and the other representing the function

*F*2 = *b∧c*, the second BDD would simply be a pointer to the positive cofactor of the first BDD.

Each bdd node has an internal variable probability that holds the probability of the Boolean

function being represented. The probability of a Boolean function is defined as the number of 1s in its truth-table divided by the total number of truth-table rows. For instance the probability *p*(*F*1) of

the Boolean function *F*1 = *a∧b∧c* is 1 , because it has a truth-table of size 8, with a single 1 in it.

8

## smart\_pointer.h

Since there may be many pointers to bdd nodes within a hierarchy, each representing different functions, keeping track of who is in charge of eventually deleting a bdd node is tricky. For example, suppose you are done working with a BDD, can you destroy the entire BDD by navigating the entire tree and deleting each node? What if another pointer exists that is pointing to a function represented by a subtree of the original BDD? Or what if it is the subtree of another function?

To handle such difficulties, all pointers to bdd nodes are handled using smart pointers. Smart pointers work by associating a reference count with the object that they point to. Upon creation, they increment the reference count of the object. Upon their destruction or assignment to another object, rather than deleting the object, they decrement the reference count of their object, and only if the count is zero do they delete it. If other smart pointers are pointing to the same object, the reference count will still be above zero, and the object will persist. In this way, the clean up of the dynamically allocated objects is handled automatically, so you do not have to worry about memory leaks (objects that are never destroyed, and lost forever) or segmentation faults (by calling delete on the same pointer more than once).

In bdd node.h, there is a typedef defining a bdd ptr type:

typedef smart pointer<bdd node> bdd ptr;

In each function you implement, you will be passed a bdd ptr to the head of whatever func- tion(s) you need to operate on. You can use them just like normal pointers, only you will not need to worry about deleting them. In fact, since each bdd node should only be created through the unique table with the function create and add to unique table(), you will never need to call new or delete to create or destroy bdd nodes. There are more details on the unique table below.

## bdd\_tables.h and bdd\_tables.cpp

Two tables help maintain the BDD universe. First, the *unique table* keeps track of every unique node, and handles the creation of new ones. Given a node *n* with respect to a variable and two child nodes, the function find in unique table() can tell you whether or not *n* exists in the *unique table*. If *n* does not exist, then the function create and add to unique table() can be used to create and add *n* to the *unique table*. The role of the *unique table* is crucial in maintaining the canonical nature of the BDD universe, so it is important that you first check to see if a certain node exists before creating a new one.

Second, the *computed table* keeps track of previous computations. Each time you find a non- trivial result (i.e., a result that requires recursive calls to apply) of an operation on two BDDs, you should enter in the result into computed table using insert in computed table(). Con- versely, before making recursive calls to apply, check the computed table first using

find in computed table() to make sure the result has not been computed before. While the *computed table* is not crucial to the correctness of the BDD universe like the *unique table*, it helps make apply more efficient.

The bdd tables class manages these two tables, and provides an interface to find in or insert into each. bdd tables is maintained as a singleton, meaning only one instance of it can be created.

To get access to the tables, use

bdd tables::getInstance();

The one and only bdd tables object will be created automatically on the first call, while subsequent calls will return a reference to it. If you are unfamiliar with the idea of a singleton, it is easiest to think of it as a global variable with the guarantee that only one instance can be created.

## operation.h and operation.cpp

The operation class allows you to perform binary operations on bdd nodes when the result is apparent from the nodes alone without looking at their children. operation is a function object, meaning that an operation object can be called like a function, taking two bdd node pointers as its arguments and returning a resulting bdd node pointer. If the result cannot be determined from the nodes alone, a NULL pointer is returned indicating the result could not be determined. For example, when applying the operation AND to two bdd ptrs *bdd1* and *bdd2*, they must either be the same node, or one of them must be the node representing **0** or **1**, the result is not known immediately without examining further down the tree, and a recursive call is necessary.

Each operation object maintains internally what operation it will perform when it is called, and provides an interface to change that operation. Currently, only the AND, OR and XOR operations are supported.

The following code fragment demonstrates basic usage of the operation class.

bdd\_ptr bb1, bdd2;

...

operation op; op.set\_operation("or");

bdd\_ptr = result = op(bdd1, bdd2); if (result == 0)

{

// operation failed, not a terminal case

}

else

{

// result is valid, can be used as the "or" of bdd1 and bdd2

}

## Bool\_expr.h, Bool\_expr.cpp, Bool\_expr\_parser.h, and Bool\_expr\_parser.cpp

The files Bool expr.h, Bool expr.cpp, Bool expr parser.h, and Bool expr parser.cpp make up a binary that facilitates the parsing and creation of Boolean expressions. The main class is Bool expr. A Boolean expression is made up of a tree structure of Bool expr nodes. Each Bool expr object is of type VALUE, AND, OR, or NOT. If it is of type VALUE, then it is a leaf node

that contains the literal. If it is of type NOT, then its *right* member variable points to a Bool expr node that is to be inverted. If it is of type AND or OR, then its *left* and *right* member variables point to the nodes that are ANDed or ORed together.

You will not have to work directly with the classes defined in these files for this assignment.

## main.cpp

The main function is provided as well as functions that parse the input, and build BDDs. The main function calls build bdd from input, which parses the user input using the Bool expr parser class, and then calls bdd from expr. The bdd from expr function traverses the Bool expr object passed to it and builds a BDD using the apply function.

## project1.h and project1.cpp

These two files contain the tasks that you will be implementing. Again, you should only be modifying

project1.cpp.

# Getting Started

Download the tar ball eecs478p1.tar.gz from CTools. To decompress the tar ball, type:

tar -xvf eecs478p1.tar.gz

into a directory of your choice. This will create a directory called eecs478p1, and will include the necessary files for your program. In that directory, compile the program by typing make. To run the program, type ./project1.

You are allowed to do your development on any platform. However, your code will be tested and evaluated on the CAEN linux machines. If you are unfamiliar with Linux, you can get some help by visiting <http://www.engin.umich.edu/caen/faqs/linux> or by looking up other tutorials available online. You are encouraged to test your program with a rigorous set of test cases that exercise all of the features of the program. You are allowed to trade test cases with other students, as long as you name

all the people with whom you traded the test cases in your README file.

# Tasks (100 points)

Your task is to implement the following tasks/functions: (1) apply, (2) negative cofactor,

(3) positive cofactor, (4) boolean difference, (5) handling node probabilities, and (6)

check P equivalence.

## Task 1: The apply Function (40 points)

Given a basic operation that can operate on terminal cases, apply can perform that operation on two BDDs of arbitrary complexity. There are many different ways to implement this function, so it is important that you understand each portion of your own implementation.

## Task 2: Positive and Negative Cofactors and Boolean Difference (20 points)

To implement the cofactor functions, again, think recursively. First, what are the terminal cases? When do you know what the cofactor of a bdd node is without seeing its children? For the non- terminal case, you will need to make recursive calls, and construct a new node. Keep in mind that the *unique table* must be used to create new bdd nodes.

The Boolean difference of a Boolean function *f* (*x*1*, x*2*, · · · , xn*) with respect to variable *xi* is de-

noted as *d f*

*dx*

*i*

and is defined

*d f*

*dxi*

= *f* (*x*1*, x*2*, · · · , xi−*1*,* 0*, xi*+1*,··· , xn*) *⊕ f* (*x*1*, x*2*, · · · , xi−*1*,* 1*, xi*+1*,··· , xn*)

or in other words, the nagtive cofactor XORed with the positive cofactor.

## Task 3: Handling Probabilities (20 points)

As mentioned earlier, each bdd node contains a variable probability that carries the probability of the function being represented. The probability value of each node must be calculated and assigned recursively as the node is being created in the apply function. While your first task was to ensure that the apply function creates nodes properly, in this task you need to make sure that each node gets the correct probability as well.

## Task 4: Checking P-equivalence (10 points)

Two Boolean functions *F* and *G* are *P-equivalent* if they have the same probability. For example, the functions *F*(*a, b, c*) = *acl* + *bc* and *G*(*a, b, c*) = *ab* + *ac* + *bc* are P-equivalent, because they both have

fours 1s in their truth-table. In this task, you implement the check P equivalence function that checks if two BDD nodes are P-equivalent.

## Task 5: README file (10 points)

Write a README *{*uniquename*}* file, where *{*uniquename*}* is your uniquename, as your project report and include it in your submission. Write one to two paragraphs *total* about the functionalities

for each task. In the case where you could not finish a task, please mention it the in README file as well. Give the names of any students (if any) with whom you have traded test cases.

# Bonus Task (10 points)

Two Boolean functions *F*(*x*1*,··· , xn*) and *G*(*x*1*,··· , xn*) are *SC-equivalent* with respect to (wrt) vari- ables *x*1*, · · · , xm*, if for all possible values of *x*1*, · · · , xm*, the corresponding cofactor of *F* is P-equivalent

to the corresponding cofactor of *G*.

For example, the functions *F*(*a, b, c*) = *acl* + *bc* and *G*(*a, b, c*) = *ab* + *ac* + *bc* are SC-equivalent

wrt *a, b* because:

*Falbl* = 0 is P-equivalent to *Galbl* = 0 and

*Falb* = *c* is P-equivalent to *Galb* = *c* and

*Fabl* = *cl* is P-equivalent to *Gabl* = *c* and

*Fab* = 1 is P-equivalent to *Gab* = 1.

Your task is to implement the function check SC equivalence, which gets two bdd nodes, and returns true if the two BDD nodes are SC-equivalent wrt their top *m* variables (defined by the parameter number of variables). For this task (and only this task) you can assume that the two BDD nodes have the same top *m* variables (i.e., the top *m* variables appear on both nodes).

# Deliverables

In this project you will only change the file project1.cpp. You will also generate a README file. These are the only files you have to deliver:

* 1. project1.cpp
  2. README *{*uniquename*}* where *{*uniquename*}* is your uniquename. Create a tar ball with *only* these files. To do this:

1. Create an empty directory called EECS478P1\_*{*uniquename*}*
2. Copy both deliverables into this directory
3. Type tar -cvf EECS478P1\_ *{*uniquename*}*.tar EECS478P1\_*{*uniquename*}*
4. Type gzip EECS478P1\_*{*uniquename*}*.tar

Submit the compressed file through the online form at CTools *→* Assignments *→* Project 1. No printed copy is required. The submission deadline is October 17*th* 2014, 23:55.