EECS 478 Fall 2020 Project No. 1

The University of Michigan, EECS Department

EECS 478: Logic Synthesis and Optimization

Professor John P. Hayes

Abhinav Sharma

Project 1: Binary Decision Diagrams

Distribution Date: Thursday September 24th, 2020
Due Date: Thursday October 15th, 2020 at 23:59 Submit Online via Canvas →Assignments →Project 1

INSTRUCTIONS:

- 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 Professor Hayes or GSI.
- Make sure your code is indented properly, readable and well commented.
- Your code should not output anything at the display, unless explicitly mentioned. Verbose codes will be penalized.
- When you are finished, collect the following two deliverables:
  - `project1.cpp`
  - `README_{uniquename}` where `{uniquename}` is your unique name.
- Create a tar bar with *only* these files. To do this:
  - Create an empty directory called `EECS478P1_{uniquename}`.
  - Copy all your deliverables into this directory.
  - Type `tar -cvf EECS478P1_{uniquename}.tar EECS478_{uniquename}`
  - Type `gzip EECS478P1_{uniquename}.tar`
- There are 100 points possible for this project.

1 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 and variable influence calculation.

2 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`.

2.1 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 $F_1 = a \wedge b \wedge c$, and the other representing the function $F_2 = b \wedge c$, 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 $F_1 = a \wedge b \wedge c$ is $\frac{1}{8}$, because it has a truth-table of size 8, with a single 1 in it.

2.2 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 function(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.

2.3 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()`. Conversely, 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.

2.4 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_ptr`s *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
}
```

2.5 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 `AND`ed or `OR`ed together.

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

2.6 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.

2.7 project1.h and project1.cpp

These two files contain the tasks that you will be implementing. Again, you should only be modifying `project1.cpp`.

3 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://caenfaq.engin.umich.edu 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.

4 Tasks (100 points)

Your task is to implement the following five functions: (1) `apply`, (2) `negative_cofactor`, (3) `positive_cofactor`, (4) `boolean_difference`,(5) `sort_by_influence`. and (6) `check_Probability_equivalence`

4.1 Task 1: The `apply` Function (30 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.

4.2 Task 2: Positive and Negative Cofactors and Boolean Difference (25 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 $f(x_1, x_2, \cdots, x_n)$ with respect to variable x_i is denoted as $\frac{df}{dx_i}$ and is defined

$$\frac{df}{dx_i} = f(x_1, x_2, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n) \oplus f(x_1, x_2, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_n)$$

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

4.3 Task 3: Probability and Variable Influence (25 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.

For a given Boolean function $f(x_1, x_2, \cdots, x_n)$, the influence of a variable x_i is defined as $influence(f, x_i) = p(\frac{df}{dx_i})$, i.e., the probability of the Boolean difference of f with respect to x_i. The influence of a variable is in range [0,1] and it indicates the dependency of the function to that variable. A variable with 0 influence is redundant, meaning that the function is independent of that variable.

Your task is to evaluate the influence of the variables of a given `bdd_node`, sort the variable by their influence and display them in descending order. For example, for the Boolean function $f(a,b,c) = ab + bc$, the `sort_by_influence` function must display a list of the variables with the following order and format

```
b,0.75
a,0.25
c,0.25
```

where the first column shows the variable names, and the second column, separated by a comma,

shows their influence. The order of the variables with the same influence does not matter. For simplicity, you can assume that the number of the variables is no greater than 20. **Note that this is the only place where your code should output to the display**.

4.4 Task 4: Checking Probability-equivalence (10 points)

Two Boolean functions F and G are Probability-equivalent if they have the same probability. For example, the functions F(x, y,z) = xz' + yz and G(x, y, z) = xy + xz + yz are Probability-equivalent, because they both have fours 1s in their truth-table. In this task, you implement the check_Probability_equivalence function that checks if two BDD nodes are Probability-equivalent.

4.5 Task 4: README file (5 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.

4.6 Task 5: Discussion (5 points)

Address the following question in your README file. Provide enough evidence (theoretical and experimental) to support your answer.
Question: "Can we reduce the size of BDDs by reordering the variables according to their influence value?"

5 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 Canvas → Assignments → Project 1. No printed copy is required.