EECS 478: Logic Synthesis and Optimization
Project 1 - Frequently Asked Questions
The University of Michigan - EECS Department
Fall 2020

This document is a collection of common questions anticipated during the course of Project 1. Going through this list of questions might help you get started faster.

Q: Don't I need to write the `*_cofactor` functions first, so that `apply` can use them?

A: No. If you had working `*_cofactor` functions, they would of course work when called from `apply`, but the full functionality of the `*_cofactor` functions isn't needed in `apply`; only some of the terminal cases.

After choosing a splitting variable, you should be able to figure out the necessary cofactors from the nodes alone (without considering their children).

Q: I'm confused about the `Operation` class. How do I use the `and_func`, `or_func` and `xor_func` that I see implemented in `Operation.cpp`?

A: The actual functions that implement the terminal operations: `and_func`, `or_func` and `xor_func`, are maintained internally by the `Operation` class. To change which function is being used, you use the `set_operation` function:

```
Operation op;
op.set_operation("or");
```

The above snippet leads to `op` internally changing its function to `or_func`. On subsequent calls to its `operator()` (using `op` like a function):

```
bdd_ptr result = op(bdd1, bdd2);
```

the internal call to `or_func` will be made. If you are new to function objects, they can appear tricky at first.

```
result = op(bdd1, bdd2);
```

is really the same thing as:

```
result = op.operator()(bdd1, bdd2);
```

Allowing classes to overload `operator()` in C++ gives you the ability to design objects that can be used like functions.

Q: How do I implement Cofactor Functions on BDDs?

A: In general, when implementing an algorithm recursively, you must first consider the terminal cases (when you know you are done). Next, if the result isn't immediately known, you need to figure out how to phrase the problem in terms of children of the existing node(s) and the operation you are implementing:

```
Bdd_node *pos_cf(Bdd_node *F, char var)
{
```

1. Is this a terminal case? If so return the result
2. Otherwise we must solve the problem by phrasing the problem in terms of `F`'s children using 1 or more calls to `pos_cf`.

```
}
```

The terminal cases are relatively straightforward since we dealt with them in `apply`: if the current node matches the var, or if we are on a terminal node or ... (one other terminal case).

For non-terminal cases, you are left with the problem of finding the cofactor of the function with respect to a variable that isn't at the current node (and may exist further down in the tree).

So, let's say the variable at the current node is 'x', and the variable that we are trying to cofactor with respect to (the argument var) is 'y. We are trying to find F_y, but all we have is a node with var x. At our current node, we have a representation of Booles expansion w.r.t x:

$$xF_x + x'F_{x\prime}$$

What happens if we take the positive cofactor w.r.t y on this?

$$(xF_x + xF_{x\prime})_y$$

Taking the above expression and manipulating it further yields a formula that represents how to break down the cofactor function into recursive calls.

Q: How do I choose a variable to split on?

A: Check out the `find_next_var()` function. You may find your answer there.