CS-472: Design Technologies for Integrated Systems

Programming Assignment 5 Due date: 15/12/2022, 23:59

1 Introduction

In this homework, we are implementing a very simple *Binary Decision Diagram* (BDD) package. To be more specific, a data structure representing *Reduced Ordered BDDs* (ROBDDs) and implementation of some basic operations of ROBDDs. In the following, we will refer to ROBDDs as simply BDDs.

To simplify the work, we will only use a fixed variable order with a fixed number of variables. The standard efficiency-enhancement and memory-reduction techniques, such as complemented edges, variable reordering, operation caching (computed table), reference count tracing, garbage collection, etc., are not employed either. If you are interested, feel free to reference to open-source BDD packages such as CUDD[1].

2 Data Structure and Algorithms

BDDs are directed acyclic graphs (DAGs) consisting of levelized nodes. A BDD `Node` has a corresponding variable `v` and two children nodes `T` (the THEN edge) and `E` (the ELSE edge). In our `BDD` package, variables are represented with `var_t` (which is actually just an unsigned integer) and nodes are represented with `index_t` (which is also an unsigned integer) indicating their position in the `nodes` array.

We use a fixed variable order: x_0 on the top and x_{n-1} at the bottom. For convenience, constant (terminal) nodes are at the n^{th} level and with indices 0 and 1. A `unique_table` is used to obey the reduction rule. A helper function `unique` is implemented for you. This function will look up in the unique table to see if a node identical to the requested one already exists. If so, it will return the index of the node; if not, a new node will be created and its index will be returned.

Operations with BDDs can be done recursively. In each recursive step, we identify the topmost variable x that we want to branch on and compute the operation result with the cofactors. Note that if a node is below x, it does not depend on x and hence its cofactor with respect to x is itself. In other words, we only branch on (take the children of) the node(s) at level x. (The pseudo-code on the lecture slides is the special case when the two nodes are at the same level. This is not complete!)

[1] `https://davidkebo.com/cudd`

3 Tasks

The implementation of the NOT and XOR operations is provided. Look into the code and try to understand it. Then, implement the AND operation similarly to XOR. Finally, implement the ITE operation, which is a little bit more challenging as it involves three operands. Search for `TODO` in `operations.hpp` to find where you should do your work.

You can use the `print` function to print the BDD for debugging. To read the printout, you may want to rotate your head 90 degrees counterclockwise (or rotate your screen clockwise) to view it more naturally as how we usually draw on paper. Note that the shared nodes will be printed more than once, but you can check their indices to identify them.

4 Compiling and Testing

With a command line interface, type `make` to compile the code. An executable named `hw5` will be produced if compilation is successful.

Type `./hw5` to test the code. Some simple test cases are provided in `main.cpp`. Feel free to define more test cases (for example, combining more operations, or using more variables).

The grading will be done on the *selsrv1* server, so please make sure your code compiles and runs well there. Please submit `operations.hpp` (without renaming it), compressed in a ZIP file, to Moodle. Note that other files will be replaced by the TA's version during grading.