Unate Recursive Complement Algorithm

Out: March 28th, 2016; Due: April 10th, 2016

I. Motivation

1. To give you experience in implementing the Unate Recursive Paradigm (URP).
2. To show you an important application of the computational Boolean algebra: obtain the complement of a Boolean function.

II. Problem Description

In the lectures, we talked about how to use the Unate Recursive Paradigm (URP) idea to determine tautology for a Boolean function represented as a cube list using the Positional Cube Notation (PCN). It turns out that many common Boolean computations can be done using the URP ideas. In this assignment, we will extend these ides to do **unate recursive complement**. This means: we give you a file representing a Boolean function F as a PCN cube list, and you will complement it, and return $\bar{F}$ as a PCN cube list. You will write a program that performs the unate recursive complement.

1. Algorithm Overview

The overall skeleton for URP complement is very similar to the one for URP tautology. The biggest difference is that instead of just a yes/no answer from each recursive call to the algorithm, URP complement actually returns a Boolean function represented as a PCN cube list. We use "cubeList" as a data type in the pseudocode below. A simple version of the algorithm is below:

```
1. cubeList Complement( cubeList F ) {
2.     // check if F is simple enough to complement it directly and quit
3.     if ( F is simple and we can complement it directly )
4.         return( directly computed complement of F )
5.     else {
6.         // do recursion
7.         let x = most binate variable for splitting
8.         cubeList P = Complement( positiveCofactor( F, x ) )
9.         cubeList N = Complement( negativeCofactor( F, x ) )
10.        P = AND( x, P )
11.        N = AND( x̄, N )
12.        return( OR( P, N ) )
13.    } // end recursion
14. } // end function
```

There are a few new ideas here, but they are mechanically straightforward.

2. Termination Conditions

Lines 2, 3, 4 in the algorithm are the termination conditions for the recursion--the cases where we can just compute the solution directly. There are only 3 cases:

1. **Empty cube list**: If the cube list F is empty, and has no cubes in it, then this represents the Boolean function "0". The complement is clearly "1", which is represented as a single cube with all its variable slots set to don't cares. For example, if our variables are x, y, z, w, then the cube representing the Boolean function "1" would be: [11 11 11 11].
2. **Cube list contains All-Don't-Cares Cube**: If the cube list F contains the all-don't-care cube [11 11 ... 11], then clearly $F = 1$. Note, there might be other cubes in this list, but if you have $F = (\text{stuff} + 1)$, it is still true that $F = 1$, and $\bar{F} = 0$. In this case, the right result is to return an empty cube list.
3. **Cube list contains just one cube**: If the cube list F contains just one cube and it is not the all-don't-cares cube, you can complement it directly using the DeMorgan Laws. For example, if we have the cube [11 01 10 01] which is $y\bar{z}w$, the complement is:
   $$(\bar{y} + z + \bar{w}) = \{[11\ 10\ 11\ 11], [11\ 11\ 01\ 11], [11\ 11\ 11\ 10]\}$$
   which is easy to compute. You get one new cube for each non-don't-care slot in the F cube. Each new cube has don't cares in all slots but one, and that one variable is the complement of the value in the F cube.

3. Selection Criteria

Lines 6, 7, 8, 9 are just like the tautology algorithm from class, and work exactly the same. However, let us be more precise about how to actually implement this. Our goal is to specify this

so carefully that, if you implement it correct, you will get exactly the same answer as our course test code. This makes it possible for us to grade things more fairly, since everybody is "aiming" at the same function.

Here are the rules for picking the splitting variable, in order of priority:

1. Select the most binate variable. This means, look at variables that are not unate (so, they appear in both polarities across the cubes), and select the variable that appears in true or complemented form in the most cubes.
2. If there is a tie for this "most binate variable" and more than one variable appears in true or complemented form in the same number of cubes, break the tie in this manner: let T be the number of cubes where this variable appears in true form (i.e., as an x). Let C be the number of cubes where this variable appears in complement form (i.e., as an $\bar{x}$). Choose the variable that has the smallest value of $|T - C|$. This way, you pick a variable that has a roughly equal amount of work on each side of the recursion tree.
3. If there is a still a tie, and several variables have the same smallest value of $|T - C|$, then choose the variable with the lowest index. For simplicity, in this assignment, we will assume that the input variables are numbered like this: x_1, x_2, x_3, x_4, etc. If there is a tie at this point (variables appear in same number of cubes, with identical minimum $|T - C|$ value) then just pick the first variable in this list. For example, if you get to this point, and the variables are x_3, x_5, x_7, x_{11} – you pick x_3, since 3 is the smallest index.
4. What if there is no binate variable at all? Then select the unate variable that appears in the most cubes in your cube list.
5. What if there is a tie, and there are many such unate variables, that each appear in the same number of cubes? Then, again select the one with the lowest index.

We can say this more concisely like this:

if (there are binate variables) {
    pick the binate variable in the most cubes, and if necessary,
    break ties with the smallest $|T - C|$, and then with the smallest variable index;
}
else { // *there are no binate variables*
    pick the unate variable in the most cubes, and if necessary,
    break ties the smallest variable index;
}

4. Working with Returned Cube Lists

Lines 10, 11, 12 are new. The tautology code just returned yes/no answers and combined them logically. The complement code actually computes a new Boolean function, using the complement version of the Shannon expansion:

$$\bar{F} = x \cdot (\bar{F})_x + \bar{x} \cdot (\bar{F})_{\bar{x}} = x \cdot \overline{(F_x)} + \bar{x} \cdot \overline{(F_{\bar{x}})} = \mathrm{OR}(\mathrm{AND}(x, P), \mathrm{AND}(\bar{x}, N))$$

where P is the cube list representing the complement of the positive cofactor of F and N is the cube list representing the complement of the negative cofactor of F.

The $\mathrm{AND}(\mathrm{variable}, \mathrm{cubeList})$ operation is simple. Remember that in this application, you are ANDing in a variable into a cube list that lacks that variable, i.e., if you do $\mathrm{AND}(x, P)$, we know that P has **no** x variables in it. To do AND, you just insert the variable back into the right slot in each cube of the cube list. For example, $\mathrm{AND}(x, yz + z\bar{w}) = xyz + xz\bar{w}$ mechanically becomes:

$$\mathrm{AND}(x, \{[11\ 01\ 01\ 11], [11\ 11\ 01\ 10]\}) = \{[01\ 01\ 01\ 11], [01\ 11\ 01\ 10]\}$$

The $\mathrm{OR}(P, N)$ operation is equally simple. Remember that "OR" on two cube lists just means putting all the cubes in the same list. Thus, this just concatenates the two cube lists into one single cube list. Note that $\mathrm{OR}(P, N)$ puts the cube list P first and the cube list N the next.

There are a few other tricks people do in real versions of this algorithm (e.g., using the idea of unate functions more intelligently). However, we will ignore them. Note that the cube list results you get back may not be minimal, and may have some redundant cubes in them, but this is fine.

III. Input/Output File Format

We are using a very simple text file format for this program. Your code will read a Boolean function specified in this format, complement this function, and then write to **the standard output** in exactly this same format. The file format looks like this:

- The first line of the file is a number N, which specifies the number of variables in the Boolean function. It is a positive integer. We number the variables starting with index 1. For example, if this number is 6, the variables of the function are $x_1, x_2, x_3, x_4, x_5, x_6$.
- Each of the remaining lines of your file describes one cube. Those cubes together form the entire Boolean function. Each of these lines contains exactly N characters, where N is specified in the first line of the file. There are three choices for each character: "`0`", "`1`", and "`-`". If the character in a slot is "`0`", then the variable for that slot appears in the complement form in the cube; if the character in a slot is "`1`", then the variable for that slot appears in the true form in the cube; if the character in a slot is "`-`", then the cube does not depend on the variable for that slot. For example, a line "`1-0-1`" represents the cube $x_1\bar{x}_3x_5$.

That's it. This is really very simple. Suppose we have this function as input:

$$F(x_1, x_2, x_3, x_4, x_5, x_6) = x_2 x_4 \bar{x}_5 + \bar{x}_2 \bar{x}_4 x_6 + x_1 x_2 \bar{x}_3 \bar{x}_4 + x_5 x_6$$

Then the input file format would look like this:

```
6
-1-10-
-0-0-1
1100--
----11
```

Your program should also handle some boundary conditions. This means that we may give you the special Boolean function $F = 0$, or $F = 1$, as inputs. Just to be clear, if you detect the input Boolean function is $F = 0$, then you should output the cube list for $F = 1$, which has just one cube in it, but with only don't cares in this cube. For example, the **output** cube list for $F = 1$ with 6 input variables, i.e., $F(x_1, x_2, x_3, x_4, x_5, x_6) = 1$, should be:

```
6
------
```

If you detect the input Boolean function is $F = 1$, then you should output the cube list for $F = 0$, which is just an empty list with no cubes in it. For example, the **output** cube list for $F = 0$ with 6 input variables, i.e., $F(x_1, x_2, x_3, x_4, x_5, x_6) = 0$, should be:

```
6
```

Note: we ask you to write your result to **the standard output**, not a file. The output is in exactly the same format.

IV. Program Arguments

Your program takes a single command-line argument, which is the name of input file describing the cube list representation of the Boolean function. **Name your program as `compl`**. It should be invoked as

```
./compl <cube-list-file>
```

where `<cube-list-file>` gives the name of the input file.

V. Programming Language and Environment

We ask you to develop the code in Linux environment, using C or C++. Ubuntu Linux operating system is recommended. You can download it from http://www.ubuntu.com/. You can install it directly on your physical machine or on a virtual machine that lives on your physical machine. For the latter choice, you need to install a virtual machine first. For example, you can use VMware Player, which can be downloaded from http://www.vmware.com/. The installation is pretty simple and won't take you too much time.

Hint: to reduce the work load, you can use standard template library (STL).

VI. Compiling

If you write your code in C, you should compile your code using `gcc`; if you write your code in C++, you should compile your code using `g++`.

In order to let us test your code automatically, we ask you to provide us a `Makefile`. This is a special file used for compiling code in Linux environment. **You should name this file exactly as "`Makefile`"**. A `Makefile` consists of a set of **rules** of how to compile a final executable program. Each rule has the following format:

```
<Target>: <Dependency>
[Tab] <Command>
```

`<Target>` is the target you want to "make", which depends on a list of files shown in `<Dependency>`. `<Command>` is the command to "make" the `<Target>`. **Note**: There must be a **tab** before the `<Command>`; otherwise, it is a syntax error.

If this is the first time you write a `Makefile`, then you can write it in the simplest form, that is, a `Makefile` containing just one rule, which generates the final executable program. For example, suppose that you want to compile a program named `helloworld` from two `.cpp` files, `hello.cpp` and `world.cpp`, and two `.h` files, `hello.h` and `world.h`. The target is `helloworld`, which depends on four files: `hello.cpp`, `world.cpp`, `hello.h`, and `world.h`. The command for compiling the program is:

```
g++ -Wall -o helloworld hello.cpp world.cpp
```

Putting them together, you would write one rule in your `Makefile` like:

```
helloworld: hello.cpp world.cpp hello.h world.h
     g++ -Wall -o helloworld hello.cpp world.cpp
```

(You should note that there is a tab before the command!)

You should put your `Makefile` in your working directory. Once you have written your own `Makefile`, then you can **type "`make`"** in the terminal to compile the program.

A demo of `Makefile` is put in the Programming-Project-One-Related-Files.zip. Try it!

For more information about the Makefile, you can read some online tutorials. For example, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

VII. Testing

To help you debug your code, we have given you two test cases `test1.in` and `test2.in` and their correct outputs `test1.out` and `test2.out`. You can find these files in the Programming-Project-Two-Related-Files.zip. To see if your program runs correctly on `test1.in`, copy `test1.in` and `test1.out` to your working directory and execute the following commands in Linux:

```
./compl test1.in > mytest.out
diff mytest.out test1.out
```

This runs your program, taking input from the file `test1.in`, and placing output into the file `mytest.out` instead of the screen. (Here, ">" is the Linux output redirection facility, which redirects the output from the screen to the file `mytest.out`.) Then, the `diff` program compares your test output `mytest.out` with the correct output `test1.out`. If they are identical, your program passes that test case. If `diff` reports any differences at all, you have a bug somewhere. You can also use `test2.in` and `test2.out` in the same way to test your program.

We will test your code using these test cases, as well as a number of others. You should therefore definitely pass these test cases. However, you should also create a family of test cases that exercises your program with different inputs, since the test cases we have given you are not sufficient to catch all bugs.

VIII. Submitting and Due Date

You should submit your source code files together with your `Makefile` by email them to the instructor or the TA. The program generated by your `Makefile` should be named as `compl`. The due time is 11:59 pm on April. 10th, 2016.

IX. Grading

We will grade your assignment by running a variety of test cases against your program, checking your solution using our automatic testing program. Your final grade is the percentage of the test cases for which your program produces the same answer as ours.