CENG 4120 Computer-Aided Design of Very Large Scale Integrated Circuits

Homework 2 (Part B)
Due data: March 18, 2024 11:59PM

1 Introduction

We have introduced in Lecture 3 that we can use Unate Recursive Paradigm (URP) to check if a Boolean function is a tautology. The Boolean function is described by a cube list using the Positional Cube Notation (PCN). URP can be utilized to realize many other Boolean operations for cube lists as well, and it serves as a basic building block of the two-level logic minimization tool ESPRESSO. The key idea of URP algorithms is to recursively divide a complex cube list into unate cube lists that can be processed efficiently. In this homework, we will extend URP tautology and make use of it to implement a part of the **irredundant** step of ESPRESSO, i.e., removing cubes that are redundant in a cube list.

2 Redundant Cube Removal

The objective of redundant cube removal is to remove some of the cubes in the given cube list F, such that the remaining cube list is still a valid representation (i.e., cover) of the original function. The cubes to be removed are found using the following two-stage method.

1. For each cube $c \in F$, check whether c is contained by (i.e., covered by, or implies) $F \setminus \{c\}$. If a cube c is not contained by $F \setminus \{c\}$, it means that some minterms in c are not in $F \setminus \{c\}$, and thus removing c will change the original function. Such non-contained cubes are called **relatively essential cubes** that must be kept in F. We denote the set of all relatively essential cubes as E.

2. Then, for each cube $c \in F \setminus E$, check whether c is contained by E. Since all the nodes of E are kept, a cube covered by E can be safely removed without affecting the original function. We denote such cubes as **totally redundant cubes**, and the set of all totally redundant cubes as R_t.

The final result of the reduced cube list is $F \setminus R_t$. [1]

2.1 Cube Containment

It can be seen that the redundant cube removal algorithm relies on a key subroutine: checking whether a cube is contained by a cube list. This, in turn, can be implemented by tautology checking.

Theorem 1. *A cube list F contains a cube c if and only if F_c (the cofactor of F w.r.t. c) is a tautology.*

The cofactor of F w.r.t. a cube means that taking the cofactor of F w.r.t. each literal in the product term of the cube. For example, if a cube c represents the product term $x'yz$, then $F_c = F_{x'yz}$.

[1] In fact, R_t does not necessarily include all the cubes that can be removed without affecting the function. In ESPRESSO, an additional step is performed to remove all the remaining redundant cubes, but we do not consider it in this homework.

3 URP Tautology

We revisit URP tautology in this section, including an additional unate reduction step which is not introduced in the lecture. For brevity, in the Positional Cube Notation (PCN), we use 1 for a positive polarity variable (01), 0 for a negative polarity variable (10), and 2 for a don't-care (DC) variable (11).

3.1 Termination Conditions

When at least one of the termination conditions is satisfied, we can immediately declare whether the cube list is a tautology or not. There are three such conditions:

1. If the cube list F contains an all-DC cube [2 2 ... 2], F is a tautology.
2. If the cube list F is unate and it does not contain an all-DC cube [2 2 ... 2], F is not a tautology.
3. If the cube list F has single var cube appearing in both polarities (i.e., $x + x' + \ldots$), F is a tautology.

3.2 Variable Selection

After the termination checking, if no early termination is triggered, we need to select a variable to split and then make recursive calls of tautology checking. Variable selection is an important factor affecting the efficiency of a URP algorithm. In this homework, we select the "most binate" variable. More specifically, we select the variable with the most product terms dependent on it. If a tie, select the variable with minimum $|T - C|$, where T (C) is the number of cubes the variable appears in positive (negative) polarity. If still a tie, select the variable with the smallest index. Note that only binate variables are considered as candidates.

4 I/O Format

Your program should read the input and write the result to a file containing a cube list respectively. The format of the two are the same. The first line is an integer m specifying the number of variables. The second line is an integer n specifying the number of cubes. For the remaining n lines, each line will contain m non-separated digits that describe a cube. If the digit at the i-th position of a cube is 1, the i-th variable appears in the cube with positive polarity (01); if it is 0, the variable appears in the cube with negative polarity (10); if it is 2, the variable does not appear (i.e., don't care) in the cube (11). For example, the cube list representing $F = ab + a\bar{c}d + \bar{b}c$ will be

```
4
3
1122
1201
2012
```

5 Test Cases

In this homework, we will provide you with two test cases for you to verify the correctness of your program. You are encouraged to generate more test cases by yourself. There will be 8 hidden cases as well. For all test cases, $1 \leq m \leq 32$ and $1 \leq n \leq 2000$. Your grade will depend on how many of the 10 test cases your program can pass within 20 seconds.

6 Additional Requirements

1. Your program should be implemented using C/C++ or Python and make sure it can be compiled (for C/C++) and executed on a Linux system. If you want to use other languages, consult the TA first.

2. Apart from the source code, you should also include a README file in your submission specifying how to compile (for C/C++) and run your program.

3. Suppose the name of your executable is "reduce" (for C/C++) or "reduce.py" (for Python), it should be correctly executed using the following command in the command-line environment:

   ```
   (for C/C++) ./reduce <in_file> <out_file>
   (for Python) python reduce.py <in_file> <out_file>
   ```

   <in_file> and <out_file> specify the paths of the input file and output file. For those who are not familiar with command-line argument processing, you can use the following code snippets.

   For C/C++:

   ```
   int main(int argc, char * argv[])
   {
       if (argc < 2 || argc > 3) {
           printf("usage: reduce <in_file> <out_file>");
           exit(1);
       }
       char * inFile = argv[1];
       char * outFile = argv[2];
   }
   ```

   For Python:

   ```
   import sys

   if len(sys.argv) < 2 or len(sys.argv) > 3:
       print("usage: reduce.py <in_file> <out_file>")
       exit(1)
   in_file, out_file = sys.argv[1], sys.argv[2]
   ```