# SINGLE CUBE EXTRACTION

CSEE 6861: MIDTERM PROJECT REPORT - Part 1

Paolo Mantovani Kshitij Bhardwaj

## 1. Introduction

The tool targets a critical step in multi-level logic optimization: SINGLE-CUBE EXTRACTION. Single-cube extraction involves the following five steps:

- **Step 1: Read Logic Network File**: In this step, the parser of the tool reads all the equations in the given logic network and stores the cubes and variables present in each of the given function. Each function is stored in the form of function IDs.
- **Step 2: Construct Cube-Variable Matrix**: Based on the reading in De-Micheli, a cube-variable matrix is constructed next. This is a binary matrix with value 1 if a variable (represented by columns) is present in a particular cube (represented by rows), otherwise the value is zero.
- **Step 3: Compute Prime Rectangles**: This step computes all the rectangles which are not contained by any other rectangles. All the prime rectangles are of atleast two rows but can have one or more columns.
- **Step 4: Evaluate Prime Rectangles**: Prime rectangles generated in step 3 are evaluated here. First, the prime rectangles that contribute to only one function and the prime rectangles with only one column are removed from the list generated in step 3. Next, a value is assigned to each prime rectangle in the new list. The "value" of a rectangle is given by: (#literals in logic network before single-cube extraction) - (#literals in logic network after single-cube extraction). We call the new list of prime rectangles as candidate prime rectangles.
- **Step 5: Find Best Prime Rectangle**: This step finds the best prime rectangle from the list of the candidate prime rectangles. The prime rectangle with highest value is chosen as the most optimal prime rectangle for single-cube extraction.

The tool has been implemented using C++ programming language. It has three source files: main.cpp, test_cube.hpp and test_cube.cpp. "test_cube.hpp" is the header file which contains a class called "sing_cube". This class contains all the private variables that are used to store the

variables and the cubes for each functions among other important parameters such as the number of cubes, number of functions, function IDs etc. "test_cube.cpp" contains all the implementation of functions declared in the header file. The implementation uses three functions:

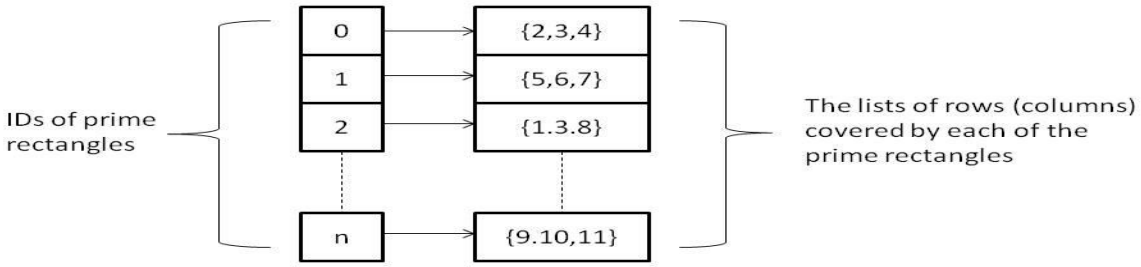
- **parse_file()**: This function parses the input logic network file and stores the different parameters required for single-cube extraction in the variables declared in the class header. This function is responsible for step 1 and step 2 and the generated output files: vars_list.txt (list of variables and column numbers); cubes_list.txt (list of cubes, row numbers and cube IDs) and matrix.txt (cube-variable matrix).
- **print_rectangles()**: This function computes all the prime rectangles in the cube-variable matrix (step 3) and also finds best prime rectangle according to the value returned by another function "get_literals_after_ext()". "print_rectangles()" is responsible for step 3 and step 5 and the generated output file: restricted.txt which contains the list of candidate prime rectangles with their values and the list of most optimal prime rectangles with corresponding values.
- **get_literals_after_ext()**: This function calculates the value of the candidate prime rectangles and returns it to the function "print_rectangles()". This is responsible for step 4.

# 2. Data Structures and Algorithms

The tool uses the following data structures and algorithms

## 2.1 Data Structures

The tool relies mostly on two data structures: vectors and vectors-of- vectors (2-D vectors). The prime rectangles to be generated at the end of step 3 have two components: a) list of rows covered by the rectangle, and b) list of columns covered by the rectangle. Two vector-of-vectors are used to store the list of rows and list of columns, respectively. The 2-D vector that stores the list of rows (or columns) for each prime rectangle has the following structure:




## 2.2 Algorithms

The tool uses the following two algorithms:

- **Algorithm 1: Compute-Prime-Rectangles**:
  - For each row, generate a set of columns where cube-variable matrix is 1 for this row. Store this set of columns in vector called "set_cols". A 2-D vector then stores all the "set_cols" and the corresponding rows. We call this 2-D vector as "vec_col_set".
  - Find intersections of each "set_cols" present in the "vec_col_set" with all the other "set_cols". If there is a valid intersection then push this intersection in another 2-D vector called "vec_vec_cols". The rows corresponding to the intersected "set_cols" are pushed in a 2-D vector called "vec_vec_rows". Both "vec_vec_cols" and "vec_vec_rows" have the structure given by the figure above.
  - There might be duplicate set of columns present in the "vec_vec_cols". Remove the duplicated set of columns and keep only the single copy in the 2-D vector. Before removing all the rows corresponding to the removed duplicated copies of the set of columns, they must be inserted into the set of rows corresponding to the single copy of the set of columns. This step generates new "vec_vec_cols" and "vec_vec_rows".
  - In order to generate the complete set of rows corresponding to every prime rectangle, intersect each set of columns ("vec_cols") in "vec_vec_cols" with the other sets of columns in "vec_vec_cols". There are two cases possible:

        - Case 1: There is a valid intersection and the result of the intersection is a set of columns ("inter_cols"), which is equal to the vector in discussion ("vec_cols"): add the set of rows corresponding to the set of columns, which intersected with "vec_cols" and produced "inter_cols" to the set of rows corresponding to "vec_cols".
        - Case 2: There is a valid intersection but "inter_cols" is not equal to "vec_cols". In this case, there are further two subcases:
            - Subcase 1: "inter_cols" is present in "vec_vec_cols" then do nothing.
            - Subcase 2: "inter_cols" is not present in "vec_vec_cols" then add "inter_cols" to "vec_vec_cols" and also add the set of rows covered by this prime rectangle ("inter_cols") into "vec_vec_rows".
    - Use Bubble sort to sort the list of rows and list of columns corresponding to each prime rectangle.
- **Algorithm 2: Algebraic Division**: This algorithm is used in the computation of the number of literals after the extraction of single-cube. After extraction, a new function "K" is created in the logic network. "K" corresponds to the prime rectangle that was extracted from other functions.
    - Each candidate prime rectangle is used to divide the functions from which it will be extracted.
    - The division uses a similar algorithm as the one presented in DeMicheli, Algorithm 8.3.1, pp. 362. The results of the division are the quotient and the remainder, which are then used in the calculation of the number of literals after extraction.
- **Optimizations**:
    - Fast in-place set intersections have been performed using the algorithms from Standard Template Library of C++. We have also used fast sort algorithms from STL libraries to sort the vectors.
    - Instead of using many nested for loops, multiple passes are done over "vec_vec_rows" and "vec_vec_cols" to compute the complete list of prime rectangles. This optimization is similar to the compiler optimization technique called 'loop fission' where a loop is broken into multiple loops over the same index range but each taking only a part of the loop's body.

## 3. Testing

The tool has been tested on many examples, listed as input1.txt to input17.txt in the tool's folder. It was tested on the benchmark provided in the handout as well as with other LNF files. We also used the problem 5 (single-cube extraction) of midterm homework to test the tool and verify the result. To test the robustness and timing complexity of the algorithm, the tool was also tested on an LNF that comprised of 20 functions, each function with 5 cubes and each cube comprising of 25 or 26 variables. In this case, our tool was able to generate the optimal prime rectangles in about 10 minutes approximately. But ofcourse, we were not able to verify the solution manually.

Many problems were fixed during the testing phase and the optimizations were added to speed up the executions. Some of the problems we fixed are as follows:

- **Literals computation after extraction**: Only the functions that were involved in extraction of cubes were included in the computation of number of literals after extraction. This was fixed to include the other functions also, which were not involved in single-cube extraction.
- **Nested for loops**: A first draft of algorithm to compute prime rectangles included many nested for loops and subcase 2 (in Algorithm 1) was not taken into consideration. We added subcase 2 in our algorithm, which helped to compute the correct list of prime rectangles. We also used loop fission optimization to speed up the execution of the tool.

## 4. Further Comments and Conclusion

The tool that we have developed as part of the midterm project can be used for single cube-extraction, an important step in multi-level optimization. A further extension of the tool is possible which can take more than 20 functions as an input and the variables are not in order and are skipped. The current version can still work correctly for more than 20 functions but the computation might take a while to generate the outputs as the number of prime rectangles will be very

large. We would also like to look at the changes that might be needed in the tool implementation so as to consider LNFs that also have complemented literals.
In the end, implementing a single-cube extraction tool was a very interesting and a good learning experience.