Function Similarity Evaluation Tool

CSEE E6861y - Computer-Aided Design of Digital Systems

Kshitij Bhardwaj
kb2673@columbia.edu

Paolo Mantovani
pm2613@columbia.edu

ABSTRACT
This tool evaluates the similarity of two functions, which have the same support, given two covers representing the functions. The tool kernel is a divide and conquer algorithm which recursively splits the covers with respect to a variable by producing positive and negative cofactors, until a termination rules applies.

The program is written in `C++`, which compiles to fast native executable and the code consists of a main function, which calls the basic components of the program:

- **The parser**: takes two PLA files as input, which describe the two covers, each of which has the same number of input variables and a single output;
- **Function Similarity**: is the heart of the program, containing the recursive function. The latter takes the parsed covers and performs three basic steps:
    1. Chek if a termination rule applies; It is worth to notice that the rules are checked in order of complexity, so that the faster rules will match first. Also, most of the rules have running time linear in the number of literals or in the number of cubes in the covers. We implemented alaso a few rules with quadratic complexity, however, these are not enabled by default.
    2. If not, then split the covers and do two recursive calls on the positive and nefative cofactor;
    3. If yes, then evaluate the similarity of the current cofactors and return.
    4. During the recursion, if the `verbose mode` is enabled, all the intermediate PLA files are saved to disk. The local variables and the arguments passed to the recursive function are small enough not to incur in a memory fault.
- **The printer**: is responsible for invoking the methods for printing the recursion tree and the PLA files if `verbose` is active.

1. FUNCTION SIMILARITY
The similarity between two functions represents the percentage of input patterns, among the complete set of possible input vectors, for which the functions give the same output. For this tool we assume that the functions are completely speciefied (i.e. the don't care set, or DC-set, is empty). A brute force approach to this problem, involves the generation of the truth tables, starting from the covers, which has exponential complexity. A classic solution is to follow the "Divide et Impera" technique, that means to split the problem into smaller subproblems until a trivial and fast termination step can be applies.

1.1 Shannon Decomposition
When no termination rule matches the current covers, the program produces cofactors to split the problem. The following equation shows how we can recover the similarity of the original functions, given the similarity of the cofactors:

$$FSim(F,G) = \frac{FSim(F_x, G_x) + FSim(F_{x'}, G_{x'})}{2} \qquad (1)$$

Notice that the function $FSim$ takes covers as input, but returns the similarity of the functions they represent. Variable x is a generic splitting variable. This simple rule is always correct, because when we split the cover, with respect to a variable, we obtain two cofactors which no longer depend on the splitting variable and contain informations about one half of the original cover.

1.2 Termination Rules
A handful set of termination rules is key to speed up the program and reduce the number of recursive calls. However, too complex rules have been discarded, because the cost for checking if they match is often times higher than the cost of an extra recursive call.

- **B1**: Both covers are empty. This is the simplest rule and we just return 1.
- **B2**: Both covers are a tautology. Again we return 1, however, tautology checking can be quite expensive, therefore, we chose not to implement the complete recursive algorithm. Instead, we just try to mathc tautology checking termination rules and let our main recursion continue. In fact, if one of the covers is truly a tautology, this rule should probably match anyway within a few recursion, while if none of the cofactors ends up being a tautology, we waste lot of time doing recursive cofactoring on covers, just to prove that they are not a tautology and we cannot terminate the main recursion.

- **B3**: One cover is empty. In this case we just need to return the size of the OFF-set of the non empty cover, divided by the total number of possible input patterns. To get the OFF-set cardinality, we actually count the number of elements in the ON-set, because the given covers contain cubes which cover the ON-set. This operation is not trivial and in the most general case requires to apply again the divide and conquer technique on the non empty cover. This time doing a recursion to calculate the ON-set cardinality is worth the effort, because this rule terminates the main recursion.

- **B4**: One cover is a tautology. This rule is complimentary to rule B3 and we need to return the ON-set cardinality of the cover which is not a tautology, divided by the number of possible imput vectors.

- **B5**: One cover is empty and the other is a tautology. This rule gives a trivial answer, which is that the similarity to be returned is 0.

- **B6**: Both covers have a single cube. In this case we can easily calculate the cardinality of the ON-sets of the two covers, by simply looking at the number of don't cares in the input part of the cubes. Moreover, we can quickly calculate the intersection of the two cubes, thus deriving in linear time both common zeros and common ones. The sum of common zeros and common ones, divided by total number of minterms gives the similarity of the covers.

- **B7**: One cover has a single cube and g has non intersecting cubes. Assuming we can check that a cover has only non intersecting cubes, then this rules extends rule B6: the size of the ON-set of the cover with multiple cubes is obtained by summing up the number of minterms covered by each of its cubes, while the intersection between the two covers is calculated by intersecting the single cube of the first cover, with every cube of the other. Notice that if the cubes in the second cover had intersections, then complexity would be exponential. Unfortunately, the problem of checking wether the cubes intersect or not has quadratic complexity, thus we chose to let this rule as an option, which can be enabled by adding the flag `-single_disjoint` to the command line when starting the tool.

- **B8**: Both covers have multiple non intersecting cubes. This rule extends further rules B6 and B7. Intuitively this is the most powerful of the three and it could save many recursive calls at the cost of two quadratic checks. Again we chose to leave it as an option, enabled by the flag `-multi_disjoint`. When this rule is enabled, rule B7 is enabled as well.

- **B9**: Both covers show single input dependence with respect to the same variable (same value). If this rule matches it means that the covers are not tautology therfore the two covers look exactly the same, but for the number of rows. Notice that there is no guarantee that the initial covers or the cofactors don't have duplicated or redundant cubes. In this case we obviousely return 1.

- **B9**: Both covers show single input dependence with respect to the same variable (but different value). This is complementary to rool B9 and we just return 0.

- **B11**: Both covers show single input dependence, but with respect to different variables. In this case it does not matter whether the variable is shown in its positive or complemented form. The similarity of the two functions is exactly 0.5, because in this special condition the two covers have half of in ON-set in common and half of the OFF-set in common.

- **B12**: One cover shows single input dependence. This rule is not trivial, but it allows early termination, by performing one last cofactoring with respect to the variable on which one cover has single input dependence. In fact, we obtain one empty cofactor and a tautology, thus rules B3 and B4 apply on the cofactors.

A wise ordering of the rules checking can speed up the algorithm in most cases, because rules with a fast check and, even more important, which allow to terminate quickly, will match first. Also, we combine the checking of similar rules, to avoid repeated calculations and we chose the following ordering:

1. Check rules from B1 to B5;
2. Check rules from B9 to B12;
3. Check rules from B6 to B8 (depends on which are enabled);
4. Check unateness conditions for termination or pruning. This set of rules is presented in the next section.

1.3 Unateness Conditions

For function similarity, having simply a condition of unateness, does not help to improve the speed of the algorithm, however, there are a few special cases, which allows to prune the recursion tree.

- **U13**: Both covers are positive/negative unate on the same variable with no DCs. In other words, this means that both covers have either all 1's or all 0's on the same culumn. If we chose the variable bound to that column, then rule B1 applies to a pair of cofactors and we can prune one branch of the recursionon.

- **U14**: Both covers are unate on the same variable with no DCs, but one is positive unate, while the other is negative unate. This is a termination rule because it implies that the covers do not share any ON-set minterm; thus we just need to calculate the cardinality of the two ON-sets to obtain the number of common zeros, which divided by the total number of combinations gives the similarity of the functions.

- **U15**: One cover is positive/negative unate in a variable Xi with no Dcs. If we pick Xi as splitting variable then rule B3 applies to a pair of cofactor and we can therefore prune one branch of the recursion.

Again the ordering is important, therefore we check first if rules U14 applies, because it terminated the recursion.

1.4 Splitting Variable

When none of the rules above applies, we pick a binate variable as splitting variable, to try keeping the tree balanced. This choice aims to favour rule B6, which is very quick to check and requires a simple calculation: getting the similarity of two cubes.

2. REGRESSION TEST

To test our program we implemented a script in the `Makefile` that executes the program with a bunch of input PLAs and checks the result against a golden output, which we calculated manually for most of the inputs. The test suite includes also a larger example, with 20 input variables, which we couldn't test manually.

Other trivial input files, not included in the test suite, have been used to test the rules on by one, right after we implemented it.

The regression script allowed us to build the program in an incremental way, checking that any additional feature we added did not compromise the previous functionality.

2.1 Covers with a large support

Our program can easily handle large input files, with covers that have many cubes. However, a major constraint is given with the number of variables. The total number of input patterns, in fact, is exponential in the number of variables, but the integer type, for instance, is encoded with 32 bits, thus we can'r represent numbers larger than $2^{32} - 1$. A second issue that could occur when we have a large number of input variables is that the similarity could be too close to 1 or 0, thus it can't be represented even using double precision floating point numbers.

3. CONCLUSIONS AND POSSIBLE IMPROVEMENTS

The tool we propose performs a fast Function Similarity Evaluation starting from two given covers in PLA format which share the same support. As a further improvement to the tool we envision to implement a functio that performs SCC over the covers. This step could be expensive, but removing all redundant cubes could dramatically reduce the total recursive calls, thus speeding up the algorithm. Given the complexity of SCC, the user should be able to set a threashold on the size of the covers, so that the tools performs it only when the cover is small enough. This is a semplification ruele (**M1**) which was not a requirement for this version of the tool.