Cpt S 515 Take-home Midterm

Please pay attention to the deadline. I will grade on the depth and completeness of your solutions. A perfect solution to the exam problems below would not be even possible. However, you are expected to present some creative and sound ideas that you want to explore in your solutions. Please also pay attention to your writing: it will include neccesary details.

Please type your solutions — handwritten solutions will not be accepted!

==========

A blackbox program is one whose source (**as well as assembly**) code is not available. However, you may run the blackbox and you can also observe the memory contents — both instruction memory and data memory along with registers – when it is run.

Some people believe that a decompiler can be used to reconstruct a high level code from the machine code of the blackbox such that the reconstructed code is, at some level, close to the source code of the blackbox. In this exam, you have to assume that such a decompiler is not possible and hence, I will not grade on any ideas that use a decompiler.

Some people also believe that, by reading machine code, one would figure out what's going on with the program. For instance, one could compare two machine coded programs and see if they are similar — you have to assume that, in this exam, this is not even correct. Syntactically comparing machine codes is meaningless! I will not grade on any ideas that try to do machine code matching.

Now, here is the problem. Consider a blackbox program P that

- takes k integer inputs (the k is known) $x_1, \cdots, x_k$;
- correctly outputs an integer value $f(x_1, \cdots, x_k) = b + \sum_{1 \leq i \leq k} a_i x_i$, where the values of the integer constants $b, a_1, \cdots, a_k$ are unknown. Hence, P always correctly computes the unknown linear function f;
- we do not know how P is implemented – it may use a call stack, it may use linked lists, it may even connect to the Internet to search wiki: who knows what P would do. However, we do know that P declares a global variable A that is a two-dimensional array A encoding a k-by-k matrix of integers and does a lot of read/write operations on the array;
- when P runs, it executes an instruction in the machine code per step;

- you can feed any integer inputs $x_1, \cdots, x_k$ to the blackbox P and then, you can observe its output, you can observe which instruction in the machine code (stored in the instruction memory) is run at each step, you can also observe all the values in A at each step of the run;

- above are the only things that you can observe on the run.

We call such a blackbox a *515-blackbox*.

Now, you are given two 515-blackboxes P_1 and P_2. Notice that the functions f_1 and f_2 computed by P_1 and P_2 respectively may not be the same. Design an algorithm to provide a similarity metric between P_1 and P_2.

Mid-Term

Sheryl Mathew (11627236)

October 15, 2018

1. Algorithm:

Let similar and not_similar be 2 global variables which contains the count of how many times the 2 blackboxes P_1 and P_2 are similar or not.

- The algorithm contains different functions which checks whether P_1 and P_2 are similar or not. For every function either variables similar or not_similar will be updated. After all the functions have been executed the count of the variables similar and not_similar will be compared. If the count of similar > count of not_similar then P_1 and P_2 are similar else they are not similar.

- **Function 1**: Run both the blackboxes P_1 and P_2 and notice the output of the black boxes which consists of the instruction sets which have been executed at each step of the code. Each blackbox output is considered as an individual sequence of instruction sets. Therefore we get two sequences I_{11} and I_{21} for the same input x_1. Compress the sequences I_{11} and I_{21} using Lempel-Ziv compression algorithm and calculate the compression rate for I_{11} and I_{21}. Run the blackboxes P_1 and P_2 for another input x_2. Compress the sequences I_{12} and I_{22} and calculate their corresponding compression rates. Run the blackboxes P_1 and P_2 for all inputs till x_n. Compress the sequences and calculate their corresponding compression rates. Find the average of I_{11} to I_{1N} to get N_1 and the average of I_{21} to I_{2N} to get N_2. Normalize N_1 and N_2 to the range [0, 1]. If the normalized value is 0 then update similar by 1 else update not_similar by 1. We use LZ algorithm to approximate Kolmogorov complexity that is the execution sequence (run-time behavior of source code) is compressed.
- **Function 2**: Run both the blackboxes P_1 and P_2 and notice the output of the black boxes which consists of the instruction sets which have been executed at each step of the code. Each blackbox output is considered as an individual sequence of instruction sets. Therefore we get two sequences I_{11} and I_{21} for the same input x_1. We can also find the instantaneous bit rate of every I_{11} to I_{1N} and I_{21} to I_{2N} is obtained and we find the corresponding averages A_1 and A_2. Then we perform time series transformation on A_1 and A_2 to find the bit rate signals for both. Plotting the signals we can compare their bit rate signals and if the 2 plots are similar then update similar by 1 else update not_similar by 1. The problem that we face is that there is no ideal method to find the bit rate signal and how to process the signal to get meaningful data

- **Function 3**: Considering the blackbox P_1 find the time taken to execute each line of code. If x_i is the input for P_1, then measure the time for the first output O_1. Let this time be t_1. Consider the time taken from the execution of O_1 to the execution of the second output O_2. Let this time be t_2. In the similar manner find all the execution times and find the average time taken. Plot the times as a histogram. Repeat the same for blackbox P_2. If the average time is same then update similar by 1 else update not_similar by 1. Also if the 2 histograms are similar then update similar by 1 else update not_similar by 1.
- **Function 4**: Assume the values of a & b before the beginning of execution. Keep all the input values in an array X. For all input values in X run the blackbox P_1 and find the output of the function f by evaluating the function based on the given a and b values. Store all the values in an array S_1. Similarly store all the values for P_2 in array S_2. Sequentially compare both the arrays ie we have to perform one to one mapping on S_1 and S_2 to check if the 2 arrays are approximately 95% same then update similar by 1 else update not_similar by 1
- **Function 5**: For P_1, consider the array X as input and array S_1 from method 4 as the output values. Split the data as 80% and 20%. Based on the 80% of data generate a linear regression equation E_1 based on the input and output values. Similarly generate a linear regression equation E_2 for P_2 considering X as input and array S_2 from method 4 as the output values for the same 80% of data. Considering E_1 for P_1 for the remaining 20% data predict the values and calculate the accuracy of the prediction AP_1. Similarly consider E_2 for P_2 for the remaining 20% of its data, predict the values and calculate the accuracy of the prediction AP_2. Then interchange the equations ie for P_1 consider E_2 for the remaining 20% data predict the values and calculate the accuracy of the prediction AP_3 and for P_2 consider E_1 for the remaining 20% data predict the values and calculate the accuracy of the prediction AP_4.. If AP_1 and AP_3 are almost the same as well as AP_2 and AP_4 then update similar by 1 else update not_similar by 1
- **Function 6**: For P_1, consider the array X as input and array S_1 from method 4 as the output values. Split the data as 80% and 20%. Based on the 80% of data generate a logistic regression equation E_1 based on the input and output values. Similarly generate a logistic regression equation E_2 for P_2 considering X as input and array S_2 from method 4 as the output values for the same 80% of data. Considering E_1 for P_1 for the remaining 20% data predict the values and calculate the accuracy of the prediction AP_1. Similarly consider E_2 for P_2 for the remaining 20% of its data, predict the values and calculate the accuracy of the prediction AP_2. Then interchange the equations ie for P_1 consider E_2 for the remaining 20% data predict the values and calculate the accuracy of the prediction AP_3 and for P_2 consider E_1 for the remaining 20% data predict the values and calculate the accuracy of the prediction AP_4. If AP_1 and AP_3 are almost the same as well as AP_2 and AP_4 then update similar by 1 else update not_similar by 1
- **Function 7**: Let the global variable for the first execution of P_1 be A_{11} and P_2 be A_{21} for the same input. IsoRank (which gives the similarity of the 2 matrixes) for A_{11} and A_{22} is calculated. Similarly the IsoRank is calculated for all the execution steps. The average is taken and normalized to the range [0, 1]. After normalization if the value is 1 then update similar by 1 else update not_similar by 1.

- **Function 8**: Let the global variable for the first execution of P_1 be A_{11} and P_2 be A_{21} for the same input. Sum of Absolute Differences for A_{11} and A_{22} is calculated. Similarly the Sum of Absolute Differences is calculated for all the execution steps. The average is taken and normalized to the range [0, 1]. After normalization if the value is 0 then update similar by 1 else update not_similar by 1.
- **Function 9**: Run both the blackbox for any one input x. Consider the global variable A value for the last execution of the blackbox. Convert the matrix to a graph where every node is labelled with i for row and j for a column and weight on the edge from I to j will be the element A[i,j]. We will get 2 such graphs, we can compare both the graphs using Similarity Flooding method. If both the graphs are similar then update similar by 1 else update not_similar by 1
- **Function 10:** Calculate the frequency of different operations that are performed by the blackbox ie the percentage of computation instruction and the data memory accesses which include load and store instructions. If the frequency is same for P_1 and P_2 then update similar by 1 else update not_similar by 1

- If count of similar > count of not_similar then P_1 and P_2 are similar else they are not similar.

  Here 2 program's semantics is compared rather than just the syntax. This is because 2 programs can have exactly the same semantics but different syntax.
  For example: Consider a C program to find 2 prime numbers it can be written using for loop or while loop when we compare the syntax the 2 programs will be different but their semantics is exactly the same.

  Source: Bit Rate of Programs [Cewei Cui, Zhe Dang, Thomas Fischer], Similarity in languages and programs [Cewei Cui, Zhe Dang, Thomas Fischer, Oscar Ibarra], Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching [Sergey Melnik, Hector Garcia-Molina, Erhard Rahm]