Control Flow Graph based Framework for effective Evaluation of Source Code

Sandeep Jain, Raju Pal, Anindya Srivastava
Department of Computer Science & Engineering
Jaypee Institute of Information Technology University, Noida

ABSTRACT
A novel framework to evaluate computer programming labs is proposed. The primary focus is fair and effective evaluation of programming labs. The proposed framework includes a user friendly interface for instructor to detect cheating and evaluate submitted programs. Every submitted program is first checked against predefined test cases. If the program passes through test cases, then it is checked for plagiarism. Finally, the program is checked for performance. An efficient and novel approach to evaluate performance of submitted programs is proposed. The computational complexity of the submitted programs is evaluated using control flow graph of the submitted program.

General Terms
Computational complexity, Algorithms, static code analysis

Keywords
Student Evaluation, Plagiarism, Program Comprehension, Static Code.

1. INTRODUCTION
Almost all undergraduate and postgraduate technical courses contain one or more computer programming labs. Also, from time to time various Computer Programming Competitions take place on the global scale [12, 13, 14 and 15] with the objective being to seek individuals with excellent programming and algorithm skills.

Various universities and programming contests around the world have been using automated evaluation of submitted programs. For instance, the ACM International Collegiate Programming Contest (ICPC) [11] makes use of the PC2: Programming Contest Control System [10]. The Stanford ACM Programming Contest [16] uses the Ultra Cool Programming Contest Control Centre tool [16]. It is worthy to note that Stanford University has launched a course "Introduction to Competitive Programming Contests" that makes use of the Peking Online Judge (POJ) [9].

Most of the present evaluation schemes test the submitted programs against a set of predefined test cases, which in turn are designed manually. Performing manual evaluation is one tedious task. This evaluation scheme is largely limited by the ability of the individual(s) designing the test cases. Sadly, this is the method employed in majority of the universities around the world. To add further, evaluating on the basis of run-time performance (time and memory) is not an ideal measure as it depends on server load, the programming language used, the compiling technology, the machine architecture and mainly the test cases. Also, according to the Computational Complexity Theory, Algorithms can be analysed in terms of their computational complexity i.e. the amount of resources (time and storage) they require in order to solve a given computational problem [2]. However, none of the current evaluation systems use the concept of computational complexity analysis. Computational complexity of the source codes is a necessary metric in order to achieve effective evaluation.

On a separate but related note, considering the humongous amount of code to be evaluated, source code plagiarism must also be taken into account. In the words of J. P. Gibson "At all stages of education and learning, students' reuse of other peoples' work and ideas is fundamental and it is to be encouraged. What is not encouraged is when the work of another person is presented as a student's own. This is plagiarism and must not be tolerated." [3].

In this paper, a novel evaluation framework is proposed. The aim is effective and fair evaluation of submitted programs. The framework checks for compiler errors, runs against test cases, checks for plagiarism and finally evaluates performance by analysing the source code for time complexity.

2. RELATED WORK
Currently, there exist two methods for evaluating source codes – manual and computer-aided. Computer-aided evaluation is more popular among Programming Contests where an ever large number of source codes are to be evaluated. PC2, the Programming Contest Control System in support of Computer Programming Contest activities, (used at ICPC [11] World Finals until 2008) [10] is one such popular tool. In fact, there even exist tools for designing/generating test cases, *e.g.*, HATE - Harness for Algorithm Testing and Evaluation [5].

However, neither PC2 nor any other tool provides any mechanism to evaluate the computational complexity of the source codes. Neither do they address the issue of plagiarism. Same is the case with other online systems such as TopCoder [14], CodeChef [15], etc.

The source codes written for computer programming assignments in universities and for programming contests problems are, by far, most prone to source code plagiarism [4]. There exist a number of tools to address this issue: MOSS – Measure of Software Similarity [6], JPlag [7], etc.

3. FRAMEWORK OVERVIEW
This paper proposes a three-phase framework for effective and fair evaluation of source codes (Fig. 1).

Phase 1: Check for compiler errors, then check for correctness with the help of predefined test cases.

Phase 2: Check for plagiarism.

Phase 3: Check the computational complexity with the help of predefined computational model.

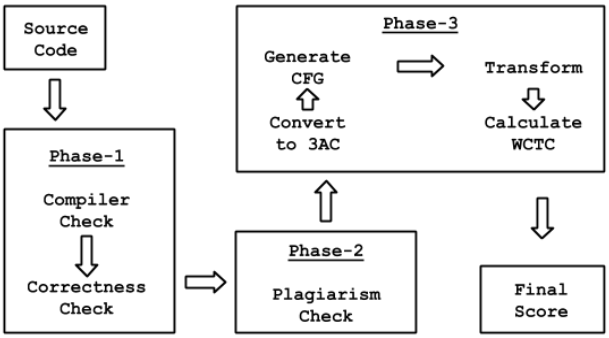

Fig. 1: Framework Architecture

4. PROPOSED FRAMEWORK

The proposed framework is described as follows:

Phase 1: Check for correctness

The program is first checked for compiler errors, then for correctness with the help of predefined test cases. By far, the most practical method of evaluation is to check the run-time performance against a set of predefined test cases. This initial approach is essential in order to check the validity of the code. At this point, the code is expected to meet certain time and memory thresholds so as to consider it for further analysis.

Apart from the limitations imposed by the hardware (machine architecture) and the software (server load, programming language used, compiling technology used), this method is further limited by the design of the test cases. Designing test cases is a tedious task. It is impractical to manually design them. As a consequence, test scripts are written to automate this process, but these test scripts themselves are prone to error, thereby compromising the validity of the original source code which is to be tested.

Phase 2: Check for plagiarism

Assuming that the source code qualified Phase *1*, the next and equally important task is to check for plagiarism. Detecting plagiarism in source codes manually is going to require individuals with excellent memory and experience. To overcome this issue the tools like MOSS [6] and JPlag [7] are widely used. Depending upon the programming language an appropriate tool should be used.

Phase 3: Check for Computational Complexity

Clearing the first two phases brings to the final and the most important task: determining the computational complexity of the source code. This is done because computational complexity, by definition, is a necessary metric for evaluating source codes. Now the complexity metric obtained is to be tested against a best computational complexity metric of a model source code written by the evaluator. Surprisingly, little work has been done in this field. Perhaps the reason behind this is the non-deterministic nature of computational complexity. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for *all* possible program-input pairs cannot exist [1].

However, this paper shows that within a limit, it is possible to determine the worst case time complexity. This paper proposes an efficient and novel approach for calculating the worst case time complexity (**WCTC**) of source codes written in any structural programming language.

The first step is to write down the equivalent Three Address Code [8] (3AC) to split the code into atomic instructions. Doing this also ensures the evaluation to be machine independent. At this point a safe assumption is made, that these atomic instructions take a constant time to execute, i.e. O(1) time.

Using the obtained 3AC, the Control Flow Graph [8] (CFG) for the source code is constructed. After the construction, the graph *G* is divided into sub-graphs (*Gin* 's) such that the *Gi*'s either have no loop or have a single first-degree loop. This process is repeated indefinitely for all the *Gin* 's hence obtained with the first degree loops; until there is no sub-graph left with a nested loop. Note that while splitting the graph, the obtained sub-graphs are assigned an incremental degree-value at every repetition. This value will come in handy at a later stage of the calculation. It is denoted by the superscript *n* whereas the subscript *i* is used to identify different sub-graphs with same degree-value.

Observe that now the sub-graphs (*Gin* 's) can be divided into two types - one without loops, and other having single loop. The WCTC of loop-less sub-graphs will be O(1). This is true because such a sub-graph consists of an arbitrary but constant number of atomic instructions each having O(1) time complexity.

Next consider the other class of sub-graphs (*Gin* 's), those having a single loop. Determining the WCTC of these graphs means estimating the maximum number of times control can return in this loop. This requires a closer look into the atomic instructions included in the sub-graph. It is checked whether these instructions can safely be rewritten in the normal-form depicted in Fig.2, while causing no harm to the original logic implemented.

```
NORMAL FORM
…
<Loop>:
Test [Iterator <= Limit]
True:
…
/*loop-body*/
…
Iterator = Growth-Function(Iterator)
goto <Loop>
False:
…
```

Fig. 2: Normal Form

This transformation is key to this calculation. By determining the *Growth_Function* working on the *Iterator*, the WCTC of the sub-graph can be determined. For example, let the *Growth_Function* be:

1. Iterator = Iterator + C (Constant); // (C>=1) WCTC = O (n)
2. Iterator = Iterator * C (Constant); // (C>=2) WCTC = O (logC n)
3. Iterator = Iterator ^ C (Constant); // (C>=2) WCTC = O (logC logC n)
4. Iterator = Iterator + 1 / (Iterator ^ Limit); WCTC = O (a^n) where a=constant.

Having determined the WCTC of both the classes of sub-graphs (*Gin* 's) completes first part of the calculation. Next, the WCTCs of the sub-graphs (*Gin* 's) with the highest degree are used to determine the WCTC of their parent super-graph. This is done by multiplying the WCTC of the parent sub-graph with the maximum WCTC value from the set of WCTC values of its child sub-graphs. Repeating this process *n* times

gives the WCTC of the complete graph (*G=Gn=0*). This can be formulated as follows:

*WCTC of Parent Super-graph (Gik)(k<n) = WCTC of Parent Super-graph (Gik)(k<n) calculated from Loop-Analysis * MAX { WCTCs of its Child Sub-graphs (Gik+1) }*

Thus, for a specific class of source codes whose loops can be transformed into the special format mentioned in this paper, this paper devises a novel way of computing the Worst Case Time Complexity.

5. COMPLETE EXAMPLE

Consider the sample C code-snippet Fig 3. Here is a general iterative function with both multiple and multi-level loops, as well as multiple branches. This example is used to depict the simplicity of the method adopted in this paper. [Note: This example focuses on Phase-3 of the Framework.]

```
int foo(int n)/*Function=O(n^2)*/
{
    …
    while(i<=n)/*Loop=O(n)*/
    {
        …
        i=i+1;
    }
    …
    while(i<=n)/*Loop=O(nlogn)*/
    {
        …
        while(j<=n)/*Loop=O(logn)*/
        {
            …
            j=j*2;
        }
        i=i+1;
    }
    …
    /*3-way Branch=O(n^2)*/
    if(test-1)/*Branch1=O(log logn)*/
    {
        …
        while(i<=n)/*Loop=O(loglogn)*/
        {
            …
            i=i*i;
        }
        …
    }
    else if(test-2)/*Branch2=O(n)*/
    {
        …
        while(i<=n)/*Loop=O(n)*/
        {
            …
            i=i+1;
        }
        …
    }
    else /*Branch3=O(n^2)*/
    {
        …
        while(i<=n)/*Loop=O(n^2)*/
        {
            …
            while(j<=n)/*Loop=O(n)*/
            {
                …
                j=j+1;
            }
            i=i+1;
        }
        …
    }
    …
} /*End-of-foo(n)*/
```

Fig. 3: Sample Code

Step 1: Convert the Program into equivalent 3AC

This step is necessary in order to simplify the complex high-level code and to partition it into basic blocks. A basic block is a maximal sequence of instructions that can be entered only at the first instruction in the sequence and exited only at the last instruction in the sequence [8]. Considering that the 3ACs have atomic instructions it can be safely assumed that each basic block corresponds to a time-complexity of O(1).

Step 2: Generate the Control Flow Graph

A Control Flow Graph is, basically, a graphical representation of all the paths that might be traversed during the program's execution [8]. In general use, the nodes of a CFG are shown as decision points from where the program control can choose between two paths depending upon the Boolean state of the test-condition (See Fig. 4).

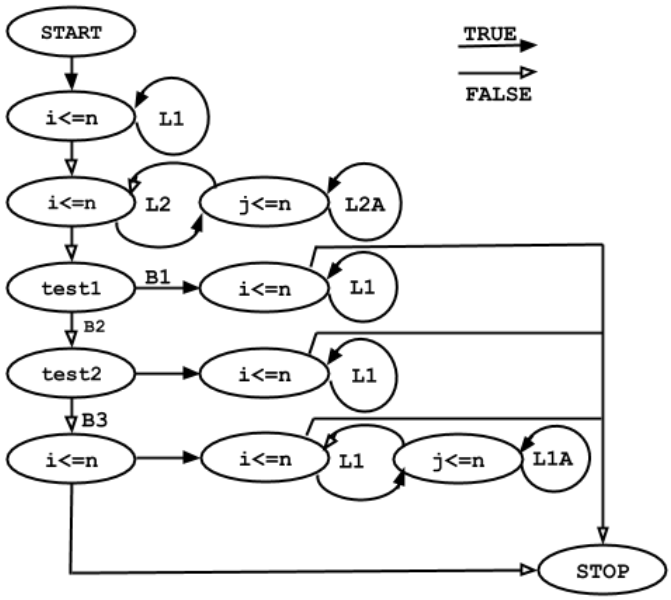

Fig. 4: Control Flow Graph

These decision-points/nodes are basic blocks, really. This means every node has O(1) time-complexity, which is obvious.

Intuitively, a CFG without any loop will correspond to a code with O(1) time-complexity. In order to not have a constant time-complexity, the CFG must have at least one loop. Moreover, even if the CFG has loops and the loops are independent of the input parameters (to the corresponding function), the function will correspond to a constant time-complexity yet again. So, it becomes clear that not only loops are necessary but, the dependency of these loops upon the input parameters is necessary as well.

For every loop whose iterations are dependent upon the input parameters, there exists a duplet D(loop-iterator, test-condition), which must be a function of the input parameters. This relationship between the duplet D and the input parameters can depicted in two ways-

a. test-condition/loop-bound is fixed, and different growth functions correspond to different time-complexities, e.g., *for(i=1;i<=n;i++):O(n)* and *for(i=1;i<=n;i*=2):O(log n)*.

b. growth-function is fixed, and different test-conditions/loop-bounds correspond to different time-complexities, e.g., *for(i=1;i<=n;i++):O(n)* and *for(i=1;i<=log n;i++):O(log n)*.

Clearly, these two formats are inter-convertible. The 'Normal Form' depicted in Fig. 2 corresponds to the former relationship format.

GCC, the GNU Compiler Collection [17] provides means [18, 19] for generating the equivalent 3AC and also for generating the CFG. Calling gcc with -fdump-tree-gimple option generates the 3AC and with -fdump-tree-cfg generates the CFG, e.g., "gcc -fdump-tree-gimple source.c" generates a source.c.004t.gimple file that contains the 3AC and "gcc -fdump-tree-cfg source.c" generates a source.c.013t.cfg file that contains the CFG.

Step 3: Transform the loops into Normal Form

The trickiest part is transforming the loops into the Normal Form depicted above, using the 3AC, basic-blocks and CFG obtained so far. First, consider an in-efficient but accurate compiler, i.e., a compiler that produces correct low-level implementation but performs almost no code-optimisation. Further, assume this compiler is also limited in terms of the machine instructions available, for example, it may implement only the elementary arithmetic operations – add, subtract, multiply, divide.

Such a compiler is bound to generate not only un-optimised 3ACs but also bound to implement the growth-rate of the loop-iterator using only the four elementary arithmetic operations. At this point, the transformation problem is reduced to determining the Growth-Function corresponding to the logic implemented using these four elementary arithmetic operations. A few examples of the same are mentioned earlier. With the knowledge of the corresponding basic-blocks and the CFG, loops having input-dependent time-complexities can be figured out and the above-mentioned transformation can be carried out.

Step 4: Calculate the Worst-case Time-Complexity

Consider Fig. 4, there are multiple and multi-level loops as well as multiple branches. Start by determining loops that wrap around a single basic-block – L1, L2A, B1-L1, B2-L1, B3-L1A.

Next determine the time-complexities of these loops by looking at the C version (see Code 1: Sample Code) written similar to the Normal Form discussed – L1:O(n), L2A:O(log n), B1-L1:O(log log n), B2-L1:O(n), B3-L1A:O(n).

Now determine the upper-level loops, i.e., loops wrapped around exactly one inner loop and calculate their time-complexities by checking the growth function of the iterator and multiplying it with the time-complexity of the inner loop – L2:O(nlogn)=O(n).O(logn), B3-L1:O(n^2).

Repeat this until the time-complexity of every loop within the current scope has been determined. Next, consider the fragment having branches, the total WCTC of this fragment will of course be the time-complexity of the branch having the maximum value. In this example it is B3-L1:O(n^2).

Finally, consider multiple loops, the WCTC will be the same as the time-complexity of the loop having the maximum value among them. In this example, among L1, L2 and B3-L1, B3-L1 has the maximum time-complexity and, hence, this is the WCTC of the entire function.

5. MEASUREMENTS AND EVALUATION

Table 1 and Table 2 below show the run-time iterations performed with respect to varying values of the input parameter – n, which correspond to the growth-function employed.

Table 1: Growth-Function values for increasing values of input loop-limit – 'n'

n=input	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**
f(i):i=i+c	101	102	103	104	105	106	107	108	109
f(i):i=i*c	4	7	10	14	17	20	24	27	30
f(i):i=i^c	2	3	4	4	5	5	5	5	5

Table 2: Growth-Function values for increasing values of input loop-limit – 'n'

n=input	**2**	**3**	**4**	**5**	**6**	**7**	**8**
i=i+ 1/i^c	2	17	198	2593	39971	720565	14913020

Table 3 below shows the run-time iterations performed with respect to varying values of the input parameter – n, for the function discussed in the previous section (Fig. 3). Note that the resultant value never exceeds n^2, which corresponds to the WCTC of the function. In order to provide reliable results, the Boolean value of the test-conditions 'test-1' and 'test-2' were picked randomly. This ensured the function to traverse all the possible branches.

Table 3: Growth-Function value never exceeds the WCTC value

'n'	**'i'**	**'n'**	**'i'**	**'n'**	**'i'**	**'n'**	**'i'**
1	1	11	44	21	441	31	961
2	4	12	48	22	110	32	192
3	6	13	52	23	529	33	198
4	12	14	56	24	120	34	204
5	25	15	60	25	625	35	210
6	18	16	80	26	130	36	216
7	49	17	289	27	729	37	222
8	32	18	90	28	140	38	228
9	36	19	361	29	841	39	234
10	40	20	100	30	150	40	240

6. FUTURE WORK

There is wide scope of research and development in every phase of this framework. A phase-wise list depicting a few possibilities is presented below:

Phase 1: The system lacks automated generation of test cases. Currently, the system supports only Java.

Phase 2: The current plagiarism detection module needs to be enhanced with a more structure-oriented plagiarism detection approach.

Phase 3: This phase has the widest scope of research among all. The current system does not support the concept of recursion. Also, there is some ambiguity over classification of programs with loop structures that can be converted into the format specified in this paper. Complexity of library subroutines has not been taken into account.

7. REFERENCES

[1] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, (Ser. 2, Vol. 42, 1937).

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford (2001). Introduction to Algorithms. Chapter 1: Foundations (Second ed.). Cambridge, MA: MIT Press and McGraw-Hill. pp. 3–122. ISBN 0-262-03293-7.

[3] J. Paul Gibson (2009). Software Reuse and Plagiarism: A Code of Practice. Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education. pp. 55-59. ISBN: 978-1-60558-381-5.

[4] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, Sencun Zhu (2012). A first step towards algorithm plagiarism detection. Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 111-121. ISBN: 978-1-4503-1454-1.

[5] Patrick Courtney, Neil Thacker, Adrian F. Clark. Algorithmic modelling for performance evaluation. Machine Vision and Applications, (Vol. 9, Issue 5-6, 1997). Springer-Verlag New York, Inc. Secaucus, NJ, USA. pp. 219-228.

[6] Saul Schleimer, Daniel S. Wilkerson and Alex Aiken. Winnowing: local algorithms for document fingerprinting. Proceedings of the 2003 ACM SIGMOD international conference on Management of data. pp. 76-85. ISBN:1-58113-64-X.

[7] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, University of Karlsruhe, Department of Informatics, 2000.

[8] Aho, Ravi Sethi, Jeffrey Ullman. Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986. ISBN 0-201-10088-6.

[9] Ying Fuchen, Xu Pengcheng, Xie Di. Programming grid: a computer-aided education system for programming courses based on online judge. Proceedings of the 1st ACM Summit on Computing Education in China, (Article No. 10). ISBN: 978-1-60558-441-6.

[10] Steve Baber, David Hoelzeman, Becky Cunningham, Rick Massengale. Programming contest hosting: conference tutorial. Journal of Computing Sciences in Colleges, (Volume 26 Issue 5, May 2011) pp. 113-115.

[11] The ACM-ICPC International Collegiate Programming Contest. http://icpc.baylor.edu/

[12] Google Code Jam. http://www.google.com/codejam

[13] Facebook Hacker Cup. http://www.facebook.com/hackercup

[14] TopCoder, Inc. | Home of the world's largest development community. http://www.topcoder.com

[15] CodeChef. http://www.codechef.com. CodeChef is a global programming community.

[16] Stanford ACM Programming Contest - Stanford University. http://cs.stanford.edu/groups/acm/contest/help.php

[17] GCC, the GNU Compiler Collection. http://gcc.gnu.org/

[18] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In Proceedings of the GCC Developers Summit3, pages 171-180, May 25-27, 2003

[19] GIMPLE - GNU Compiler Collection (GCC) Internals. http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html