CLASSIFYING SORTING ALGORITHMS

Pham Hong Long[1], Nguyen Hua Phung[1]

[1] *University of Technology, Hochiminh City, Vietnam*

Corresponding author*:* *phung@cse.hcmut.edu.vn*

Received August 16, 2011

ABSTRACTION

Classifying algorithms is a category in automatic computer program understanding problem. This paper introduces a new approach to solve the classifying algorithms problem. Based on this approach, we have built a system that can verify sorting algorithms written by students in order to help them improve their programming skills. The system can detect wrong implementations and even right implementations with some redundant actions.

Key words: program analysis, classifying algorithms, sorting algorithms, software engineering

1. INTRODUTION

Automatic computer program understanding has been an interesting subject for researchers in recent decades. This problem can be classified into three categories:

- Understanding functionality: answer the question of whether a program can perform specific functions or not. This is the main category in automatic computer program understanding problem.

- Classifying algorithms: indicate which algorithm in a group of algorithms that can solve the problem is implemented in the verified program.

- Analyzing structure and style: analyze how control structures are used in the program, therefore, know about programmer's style.

The main reason that makes this kind of problems interesting is its broad applications. Some of its most important applications are:

- Software verification: instead of having a group of experts to read and check program code, an automatic code recognition system can do similar work with less time, therefore can save a lot of cost in software engineering.

- Software maintenance: it is hard and tedious for a software maintenance expert to investigate a very poor documented software. It would be easier if there is a system that can read and document software automatically.

- Automatic assessment: in computer science, what takes a lot of time and effort of a lecturer is to assess programming exercises and assignments. A system that can understand programs automatically will help lecturers to assess students' programs quickly and accurately.

- Help to specify some properties of programs such as the complexity of algorithms, the use of system memory, etc. We can, therefore, know about the quality of written programs.

In this paper, we consider the problem of understanding a program from algorithm verification aspect, which can verify if a program is implemented according to a given algorithm.

For a problem, there are many algorithms to solve it. Each algorithm has its advantages and disadvantages so that it should only be applied in the appropriate context. For example, a bubble sort algorithm is simple to implement but less efficient in running while a quick sort is more complex to implement but more efficient.

This paper presents an approach to recognize an algorithm implemented in a program among existing ones. In particular, the approach is used to verify if an implementation is an in-place sorting algorithm. The approach is different from a similar previous work [1] in that we use dynamic analysis to detect the algorithm implemented.

The contributions of this paper are as follows:

- Introducing a new approach to solve algorithm verification problem and,
- Proposing a new operator to improve the expressiness of regular expression.

The remain of this paper is as follows: some related works are presented in Section 2; Section 3 presents our approach; Section 4 describes the experiments; at last, Section 5 concludes this paper.

2. RELATED WORKS

There have been some approaches that can be classified into two classes: static analysis and dynamic analysis. While static analysis approaches just analyze the source code to understand them, dynamic analysis approaches collect information when executing code.

2.1 Static analysis:

PROUST [2] is a system, built by Johnson and Soloway, used to debug and understand programs written by novice programmers. This system includes a set of sample programs that experts used to solve specific problems. When a new program is analyzed, its source code will be compared with the set of solutions stored in the system. If there is a point which doesn't match with the solution, that point may be a bug.

BUG-DOCTOR [3] is another system whose purpose is debugging and understanding programs. An analyzed program will be separated into smaller parts which do specific small tasks. These parts will be compared with some sample code used to solve specific problems, and then the functions of larger parts are found out. This process will be iterated until the function of entire program is determined.

Taherkhani [1] developed a static analysis tool to classify sorting algorithms. The tool analyses a sorting implementation and counts some attributes such as temporal variables, nested or sequential loop, etc. These values are compared to those analyzed from sample implementations of these sorting algorithms. The result is used to classify the sorting implementation.

2.2 Dynamic analysis

ASSYST [4] is an automatic assessment system which grades the students submissions based on five areas: correctness, efficiency, style, complexity and test data adequacy. To check the correctness, the submission is executed and the result is analyzed by a pattern matching tool which is developed using Unix Lex and Yacc.

Ceilidh [7], now called CourseMarker [6], is another automatic assessment system that also analyses the students' submissions dynamically. The system runs each submission with some test data, and then checks its output to determine whether the output satisfies the model output generated by model solution.

Scheme-robo [5] is a system that is used to analyze Scheme programs. It, like ASSYST and Ceilidh, executes students' program then compares the output. Moreover, it performs some static analysis to check whether some forbidden structures is used in the students' programs or not.

Although these systems assess the students' submissions based on model solutions but these models are not used to verify the implementation that conforms to the given specific algorithm. In next section, we introduce our approach that dynamically instruments students' submission to collect necessary information and then verify if the implementation follows the required algorithm.

3. OUR APPROACH

3.1 Mechanism

Figure 1 depicts the mechanism of our system. The verified program is instrumented to collect some information about the behavior of the program when it is executed. The information necessary to verify a sorting algorithm is activities that writes on the internal memory/array. The collected writing actions of the verified program is recognized by a finite automaton whose each accept state is corresponding to a specific sorting algorithm. When reading the writing actions of the verified program on a selected input, the automaton should reach an accept state regarding to the required sorting algorithm. Otherwise, the program is considered as a wrong implementation of the required algorithm. The input for the program is selected so that the writing sequences of sorting algorithms are different.

For example, let write(i,v) represent the action of writing value v to an element with index i. To sort an array of {13, 27, 6, 20}, an implementation of the bubble sort algorithm could generate the following writing actions:

write(1,6), write(2,27), write(2,20), write(3,27), write(0,6), write(1,13).

Meanwhile, an implementation of an insertion sort applying for the same input array could generate a different writing sequence as follows:

write(2,27), write(1,13), write(0,6), write(3,27), write(2,20).

By recognizing the writing sequence of a student's implementation according to a model of a given sorting algorithm, the system can verify if the implementation is of the specific one.

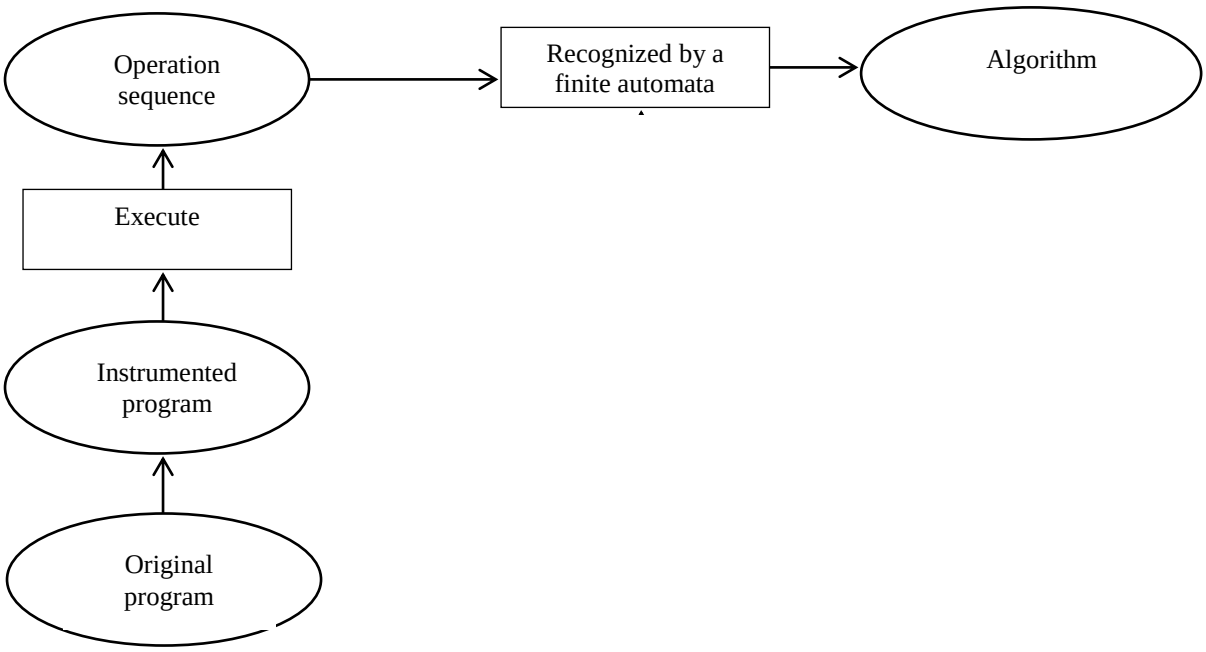

Figure 1. Mechanism of the system

3.2 Model of actions:

Although the finite automaton can be represented in form of a regular expression [8], one additional operator '!' should be introduced to improve the expressiveness. The additional operator is defined as follows:

((r)(q))! ::= (r)(q) | (q)(r) where r, q are regular expressions.

The new operator helps to represent some writing subsequences that can occur in any order. For example, when swapping two elements at index 1 and 2 of the array {13, 27, 6, 20}, two following writing sequences are possible:

write(1,6),write(2,27) or

write(2,27),write(1,6)

With the introduction of the new operator, (write(1,6) write(2,27))! can be used to recognize both above writing sequences. The new operator is especially efficient in the insertion sort to describe a moving process in which all elements can be moved in any order.

4. EXPERIMENTS

We implemented the system and provided the models for 9 sorting algorithms: bubble sort, selection sort, insertion sort, shell sort, shaker sort, odd even sort, merge sort, heap sort and quick sort.

There are 63 students participating in the experiments. They have just finished the Data Structure and Algorithms course in which they just learned 7 sorting algorithms: bubble sort, selection sort,

insertion sort, shell sort, merge sort, heap sort and quick sort. Each student is required to implement a specific sorting algorithm and then the system will check the submission.

Table 1 describes the report of the system. Column *Right* indicates the number of accurate implementations, Column *Wrong* shows the number of wrong sorting implementations and Column *Wrong algorithm* presents the number of right sorting implementation but wrong according to the required algorithm. The three submissions of insertion sort in Column *Wrong algorithm* are implemented as gnome sort [9] which is similar to insertion sort except that moving an element to its proper place is performed by a series of swaps. The other submissions in Column *Wrong algorithm* are implemented in the right algorithm with some redundant writing actions.

Table I. Report of the system

	Number of imp.	*Report of the system*		
		Right	*Wrong*	*Wrong algorithm*
Bubble sort	*7*	*6*	*1*	*0*
Selection sort	*16*	*16*	*0*	*0*
Insertion sort	*11*	*8*	*0*	*3*
Shell sort	*8*	*1*	*2*	*5*
Heap sort	*8*	*2*	*1*	*5*
Merge sort	*6*	*3*	*0*	*3*
Quick sort	*7*	*7*	*0*	*0*

5. CONCLUSION

The paper presented a new approach that can help to verify a program. Some kinds of the program behavior are logged and checked with a model. The approach has been implemented and used to verify students' submissions if they conform to the specific sorting algorithm. The system can detect wrong implementations, wrong algorithm implementations and ones with some redundant actions.

REFERENCES

[1] Taherkhani A., Korhonen A., Malmi L. – Recognizing Algorithms Using Language Constructs, Software Metrics and Roles of Variables: An Experiment with Sorting Algorithms. The Computer Journal, (2010), doi: 10.1093/comjnl/bxq049.

[2] W.L. Johnson and Soloway E. – Proust: Knowledge-based program understanding, IEEE Transactions on Software Engineering, Volume SE-11, Issue 3, March 1985, pp 267–275.

[3] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat - A role for chunking and fuzzy reasoning in a program comprehension and debugging tool, 9th International Conference on Tools with Artificial Intelligence (ICTAI '97), 1997, pp 102–109.

[4] D. Jackson and M. Usher – Grading student programs using ASSYST, Proceedings of 28th ACM SIGCSE Symposium on Computer Science Education, 1997, pp 335–339.

[5] Riku Saikkonen, Lauri Malmi, and Ari Korhonen – Fully automatic assessment of programming exercises, Proceedings of The 6th Annual SIGCSE/SIGCUE conference on Innovation and

Technology in Computer Science Education, ITiCSE'01, Canterbury, UK, 2001. ACM Press, New York, pp 133–136.
[6] Colin Higgins, Pavlos Symeonidis, and Athanasios Tsintsifas - The marking system for CourseMaster, Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education, ACM Press, 2002, pages 46–50.
[7] S.D.Benford, E.K.Burke, and E.Foxley - Courseware to support the teaching of programming, Proceedings of the Conference on Developments in the Teaching of Computer Science, University of Kent, 1992, pages 158–166.
[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman – Introduction to Automata Theory, Languages, and Computation, Second edition, Addison-Wesley, ISBN: 0-201-44124-1, 2001, pages 83–89.
[9] Dictionary of Algorithms and Data Structure, National Institute of Standards and Technology, US, maintained by Paul E. Black, updated 2011, available at http://xlinux.nist.gov/dads/

APPENDIX

The regular expressions[1] of some sorting algorithms on input [3, 2, 1, 0]:

1. Bubble sort:

(((2,0)(3,1))! ((1,0)(2,2))! ((0,0)(1,3))! ((2,1)(3,2))! ((1,1)(2,3))! ((2,2)(3,3))!

| ((0,2)(1,3))! ((1,1)(2,3))! ((2,0)(3,3))! ((0,1)(1,2))! ((1,0)(2,2))! ((0,0)(1,1))!)

2. Insertion sort:

(((1,3)(0,2))! ((2,3)(1,2)(0,1))! ((3,3)(2,2)(1,1)(0,0))!

| ((2,0)(3,1))! ((1,0)(2,1)(3,2))! ((0,0)(1,1)(2,2)(3,3))!)

[1] For simplicity, we ignore the word "write" in the regular expression