Algorithm Recognition by Static Analysis and Its Application in Students’ Submissions Assessment

Ahmad Taherkhani  
Department of Computer Science and Engineering  
Helsinki University of Technology  
Finland  
ahmad@cs.hut.fi

Lauri Malmi  
Department of Computer Science and Engineering  
Helsinki University of Technology  
Finland  
lma@cs.hut.fi

Ari Korhonen  
Department of Computer Science and Engineering  
Helsinki University of Technology  
Finland  
archie@cs.hut.fi

ABSTRACT

Automatic program comprehension (PC) has been extensively studied for decades. It has been studied mainly from two different points of view: understanding the functionality of a program and understanding program structure. In this paper, we address the problem of automatic algorithm recognition and introduce a method based on static analysis to recognize algorithms. We discuss the applications of the method in the context of automatic assessment to widen the scope of programming assignments that can be checked automatically.

Keywords

program analysis, algorithm recognition, automatic grading

1. INTRODUCTION

Current automatic assessment systems for programming exercises, such as CourseMarker [9], BOSS [12] or Web-Cat [4] provide many advantages for large programming courses. These systems have versatile methods to check the submitted programs, among which checking correctness and functionality (tested against teacher’s test data) and program structure are perhaps the most important ones. Other features available in some systems include checking programming style [5], the use of various language constructs [19], memory management [1], and run time efficiency [19].

Despite the multifunctional capabilities, it may be difficult to set up assignments that require the use of specific algorithms. For example, a data structures course could have assignments such as “Write a program that sorts the given array using merge sort”, or “Write a program that stores the given keys in a binary search tree and outputs the keys in preorder”. In this case, applying automatic assessment would require setting up tests for intermediate states of the algorithm instead of only the final state or output, yet the tests may not reveal usage of a different algorithm than requested.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Koli Calling ’08 November 13-16, 2008, Koli, Finland  
Copyright 2008 ACM 978-1-60558-385-3/08/11 ...$5.00.

In this paper, we present a method, based on research on PC, that supports automatic checking of and feedback on algorithms. The prototype implementation is based on applying static analysis of program code, including various statistics of language constructs and especially roles of variables. These allow the characteristics of various algorithms to be distinguished. The prototype has been built to distinguish between several common sorting algorithms, but the technique can be applied to other classes of algorithms as well. The first steps in our research on PC (in general) and a description of the new method is presented. Finally, some preliminary results and discussion follow.

2. PROGRAM COMPREHENSION

The problem of PC can roughly be classified into three categories. In the following, we present these categories and describe the most common applications of each.

Understanding functionality: Most of the studies in the field fall into this category. The PC problem is seen as the problem of understanding what the program does. The earlier studies were mainly motivated by the need of software developers, testers and maintainers to understand a system without having to read the code, which is a time-consuming and error-prone task (see for example [8, 14]). An automatic PC tool could be useful in software projects, for example, in verification and validation tasks.

Analysing structure and style: PC can be seen as examination of the source code, for example, to see how control structures are used and to investigate coding style. The objectives of these analyses could be to monitor students’ progress, to ensure that students’ submissions are in accordance with teachers’ instructions, and to get a rough idea about the efficiency of the code. Tools that perform these kinds of analyses are mostly used in computer science-related courses at universities and are often integrated into plagiarism detection systems [5, 17, 18].

Recognizing and classifying algorithms: The PC problem can also be viewed as the problem of algorithm recognition, i.e., being able to classify algorithms implies understanding a program. Therefore, the process of finding out what family of algorithms an algorithm belongs to or what kind of algorithm it resembles involves program comprehension tasks. The primary use of such an algorithm recognition tool is to examine submitted exercises and make sure that students have used the algorithm that they have been asked to. In this paper, the PC is discussed from this point of view.
Methods in the PC field can be divided into two categories: dynamic and static analysis. In dynamic analysis, a program is executed by some input, and the output is investigated in order to understand the functionality of the program. These methods are often used in automated assessment systems, where the correctness of students' submissions is tested by running their program using some predefined test input and comparing its output with the expected one (see for example [4, 9, 12]).

Static analysis, on the other hand, involves no execution of the code. These approaches analyze a program using structural analysis methods, which can be carried out in many different ways, focusing on different features of the code, for example, the control and data flow, the complexity of the program in terms of different metrics, etc. Most PC studies are based on static program analysis. We present the main approaches below.

2.1 Knowledge-based approaches

Knowledge-based techniques concentrate on discovering the functionality of a program. These approaches are based on a knowledge base that stores predefined plans. To understand a program, program code is matched against the plans. If there is a match, then we can say what the program does, since we know what the matched plans do. The plans can have other plans as their parts in a hierarchical manner. Depending on whether the recognition of the program starts with matching the higher-level or lower-level plans first, knowledge-based approaches can be further divided into three subcategories: bottom-up, top-down, and hybrid approaches.

Most knowledge-based approaches work bottom-up, in which we try to recognize and understand small pieces of code, i.e., basic plans first. After recognizing the basic plans, we can continue the process of recognizing and understanding higher-level plans by connecting the meanings of these already recognized basic plans and by reasoning what problem the combination of basic plans tries to solve. By continuing this process, we can finally try to conclude what the source code does as a whole. In top-down approaches, the idea is that by knowing the domain of our problem, we can select the right plans from the library that solve that particular problem and then compare the source code with these plans. If there is a match between source code and library plans, we can answer the question of what the program does. Since we have to know the domain, this approach requires the specification of the problem (see, for example, [10]). Hybrid approaches (see, e.g., [15]) use both techniques.

Knowledge-based approaches have been criticized for being able to process only toy programs. For each piece of code to be understood, there must be a plan in the plan library that recognizes it. This implies that the more comprehensive a PC tool is desired to be, the more plans must be added into the library. On the other hand, the more plans there are in the library, the more costly and inefficient the process of searching and matching will get. To address these issues of scalability and inefficiency, various improvements to these approaches have been suggested including fuzzy reasoning [3]. Instead of performing the exhaustive and costly task of comparing the code to all plans, fuzzy reasoning is used to select a set of more promising pieces of code, i.e., chunks, and carry out the matching process in more detail between those chunks and the corresponding plans.

2.2 Other approaches

The following approaches to PC can also be discerned.

Program similarity evaluation approaches: As the name suggests, program similarity evaluation techniques, i.e., plagiarism detection techniques are used to determine to what extent two given programs are the same. Therefore, these approaches focus on the structural analysis and the style of a program, rather than discovering its functionality. Based on how programs are analyzed, these approaches can be further divided into two subcategories: attribute-counting approaches [5, 17] and structure-based approaches [18]. In attribute-counting approaches, some distinguishing characteristics of the subject program code are counted and analyzed to find the similarity between the two programs, whereas in structure-based approaches the answer is sought by examining the structure of the code.

Reverse engineering approaches: Reverse engineering techniques are used to understand a system in order to recover its high-level design plans, create high-level documentation for it, rebuild it, extend its functionality, fix its faults, enhance its functions and so forth. By extracting the desired information out of complex systems, reverse engineering techniques provide software maintainers a way to understand complex systems, thus making maintenance tasks easier. Reverse engineering approaches have been criticized for the fact that they are not able to perform the task of PC and deriving abstract specifications from source code automatically, but they rather generate documentation that can help humans complete these tasks [16]. Since providing abstract specifications and creating documentation from source code are the main outcomes of reverse engineering techniques, these techniques can be regarded as analysis methods of system structure rather than understanding its functionality.

In addition to the aforementioned techniques, the following approaches to understand programs or discover similarities between them have also been presented: techniques used in clone detection methods [2], PC based on constraint satisfaction [21], task oriented PC [6] and data-centered PC [11]. Detailed discussion of these approaches is beyond the scope of this paper.

3. METHOD

Our approach in recognizing algorithms is based on examining the characteristics of them. By computing the distinguishing characteristics of an algorithm, we can compare these characteristics with those collected from already recognized algorithms and conclude if the algorithm falls into the same category.

We divided the characteristics of a program into the following two types: numerical characteristics and descriptive characteristics. Numerical characteristics are those that can be expressed as positive integers, whereas descriptive characteristics cannot. The numerical characteristics examined in our method are: number of blocks (NoB), number of loops (NoL), number of variables (NoV), number of assignment statements (NAS), lines of code (LoC), McCabe complexity (MCC) [13], total operators (N₁), total operands (N₂), unique operands (n₁), unique operands (n₂), program length (N₁ + N₂) and program vocabulary (n = n₁ + n₂). The abbreviation after each characteristic is used to refer to it in Table 1, which is explained later in this section. From these characteristics, N₁, N₂, n₁, n₂, N and n are the Halstead metrics [7] that are widely used in program similar-
ity evaluation approaches. In addition to these, some other characteristics in connection with these numerical characteristics are computed such as variable dependencies (both direct and indirect), the information whether a loop is incrementing or decrementing, and the interconnections of blocks and loops. Descriptive characteristics comprise whether the algorithm is recursive or not, whether it is in-place or requires extra memory, and the roles of variables used in it.

Based on initial manual analysis of many different versions of common sorting algorithms, we posited a hypothesis that the information mentioned above could be used to differentiate many different algorithms from each other. In the prototype version, however, we decided to restrict the scope of the work to sorting algorithms only. We studied five well-known sorting algorithms: Quicksort, Mergesort, Insertion sort, Selection sort, and Bubble sort. The problem was whether new unknown code from, for example, student submissions could be identified reliably enough by comparing the information gathered from the submission with the information in a database. We developed an Analyzer that can count all these characteristics automatically. An unfortunate obstacle was, however, that the automatic role analyzer we got access to did not evaluate the roles of variables accurately enough. Due to time constraints, we could not replace it with another one, and some role analysis had to be carried out manually.

The recognition process is based on calculation of frequency of occurrence of the numerical characteristics in an algorithm on one hand, and investigation of the descriptive characteristics of that algorithm on the other hand. First, many different versions of the implementation of sorting algorithms are analyzed with regard to aforementioned characteristics and the results are stored in the database. Therefore, the Analyzer has the following information about each algorithm: the type of the algorithm, the descriptive characteristics of the algorithm and the minimum and maximum values of the numerical characteristics. When the Analyzer encounters a new submitted algorithm, it first counts its numerical characteristics and analyzes its descriptive characteristics. In the next step, the Analyzer compares this information with the corresponding information of algorithms retrieved from the database. If a match between the characteristics of the algorithm to be recognized and an algorithm from the database is found, the type of the latter algorithm is assigned to the recognized algorithm and its information is stored in the database. If no match is found, the algorithm and its information are stored in the database as the type "Unknown". An instructor can then examine the algorithms marked "Unknown" to ensure that they really do not belong to any type of algorithms. If an algorithm marked "Unknown" does belong to a type (a false negative case), the instructor can assign the correct type to that algorithm. This way, as new kinds of implementations of an algorithm are encountered, the allowed range of numerical characteristics of that algorithm can be adjusted in the database. Thus, the knowledge base of the Analyzer can be extended: next time, the same algorithm is accepted as that particular type.

The numerical characteristics are used in the earlier stage of the decision making process to see if the recognizable algorithm is within the allowed range. If it is not, the process is terminated and the algorithm is labeled "Unknown" without any further examination. In these cases, an informative error message about the numerical characteristics that are above or below the permitted limits is given to the user. If the algorithm passes through this stage, the process proceeds to investigate its descriptive characteristics.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>NoB</th>
<th>NoL</th>
<th>NoV</th>
<th>NAS</th>
<th>LoC</th>
<th>MCC</th>
<th>N_1</th>
<th>N_2</th>
<th>n_1</th>
<th>n_2</th>
<th>N</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>5/6</td>
<td>2/2</td>
<td>5/6</td>
<td>10/10</td>
<td>16/25</td>
<td>4/5</td>
<td>47/59</td>
<td>51/57</td>
<td>4/6</td>
<td>2/5</td>
<td>98/116</td>
<td>6/11</td>
</tr>
<tr>
<td>Quicksort</td>
<td>5/9</td>
<td>1/3</td>
<td>4/7</td>
<td>6/15</td>
<td>31/41</td>
<td>4/10</td>
<td>84/112</td>
<td>77/98</td>
<td>9/17</td>
<td>2/7</td>
<td>161/198</td>
<td>13/22</td>
</tr>
<tr>
<td>Mergesort</td>
<td>7/9</td>
<td>2/4</td>
<td>6/8</td>
<td>14/22</td>
<td>33/47</td>
<td>6/8</td>
<td>96/144</td>
<td>94/135</td>
<td>11/14</td>
<td>5/9</td>
<td>190/279</td>
<td>17/23</td>
</tr>
</tbody>
</table>

Table 1: The minimum and the maximum of numerical characteristics of five sorting algorithms

Figure 1 shows a decision tree to determine the type of a sorting algorithm. At the top of the decision tree, we examine whether the algorithm is a recursive one and continue the investigation based on this. Highly distinguishing characteristic like this improve the efficiency, since we do not have to retrieve the information of all algorithms from the database, but only those that are recursive or that are non-recursive. In the next step, the numerical characteristics are used to filter out algorithms that are not within the permitted limits. As can be seen from Figure 1, the roles of variables play an important and distinguishing role in the process. All examined Quicksort algorithms included a variable with Temporary role, while none of the examined Mergesorts did. Since the Temporary role often appears in swap operations, this is somehow expected: Quicksort includes a swap operation, but in Mergesort there is no need for swapping because merging is performed. In the case of the three non-recursive algorithms that we examined, only Selection sort included a Most-wanted Holder. For the definition of different roles see [20]. The rest of the decision making process shown in Figure 1 is self-explanatory.
As an example of the numerical characteristics, we present the result of analyzing the numerical characteristics of the five algorithms in Table 1. We collected an initial database containing 51 different versions of the five sorting algorithms for the analysis. All algorithms were gathered from textbooks and course materials available on the WWW. Some of the Insertion sort and Quicksort algorithms were from authentic student submissions. For each characteristic in the table, the first and second number depict, respectively, the minimum and maximum value found from the different implementations of the corresponding algorithm. As can be seen from the table, the algorithms fall into two groups with regard to their numerical characteristics: the small group consists of Bubble sort, Insertion sort and Selection sort, and the big group comprises Quicksort and Mergesort.

4. DISCUSSION

The Analyzer is only capable of deciding which sorting algorithm a given algorithm seems to be. The correctness of the decision cannot be verified by using this method, since it is very difficult, if not impossible, to verify this using only static analysis. Dynamic methods should be used as well.

Our method assumes that algorithms are implemented using conventional and widely-accepted programming style. The method is not tolerant to the changes that result from using an algorithm in an application. Moreover, algorithms are expected to be implemented in a well-established way. As an example, although it is possible to implement Quick-sort in a non-recursive way, a recursive implementation is much more common. The same assumption is made by other PC approaches as well, e.g., knowledge-base approaches.

The most useful application of the Analyzer is perhaps verifying students’ submissions. There are many large size computer science courses lectured at universities where students are required to submit a number of exercises in order to complete a course. The Analyzer can be used to help instructors to verify the correctness of the type of the submissions. It is also possible to develop the Analyzer further to provide the students with detailed feedback about their submissions in different ways.

Although the method is examined only for sorting algorithms, it can presumably be applied to recognize other algorithms as well. Moreover, as we described previously, the roles of variables turn out to be a distinguishing factor that can be used to recognize sorting algorithms. This is, however, a topic well worth discussing further:

1. How well can the method be applied to recognize other algorithms?
2. What other factors could be used to characterize different algorithms?
3. Is there a minimum set of characteristics that is enough to solve the identification problem and how could it be found?
4. Roles of variables are cognitive concepts, thus a human analyzer may disagree with an automatic role analyzer. Is this causing serious problems?

5. REFERENCES


