Recognizing Algorithms Using Language Constructs, Software Metrics and Roles of Variables: An Experiment with Sorting Algorithms

AHMAD TAHERKHANI*, ARI KORHONEN, LAURI MALMI

Department of Computer Science and Engineering, Aalto University School of Science and Technology, PO Box 15400, FI-00076 AALTO, Finland
*Corresponding author: ahmad@cs.hut.fi

Program comprehension (PC) is a research field that has been extensively studied from different points of view, including human program understanding and mental models, automated program understanding, etc. In this paper, we discuss algorithm recognition (AR) as a subfield of PC and explain their relationship. We present a method for automatic AR from Java source code. The method is based on static analysis of program code including various statistics of language constructs, software metrics, as well as analysis of roles of variables in the target program. In the first phase of the method, a number of different implementations of the supported algorithms are analyzed and stored in the knowledge base of the system as learning data, and in the second phase, previously unseen algorithms are recognized using this information. We have developed a prototype and successfully applied the method for recognition of sorting algorithms. This process is explained in the paper along with the experiment we have conducted to evaluate the performance of the method. Although the method, at its current state, is still sensitive to changes made to target algorithms, the encouraging results of the experiment demonstrate that it can be further developed to be used as a PC method in various applications, as an example, in automatic assessment tools to check the algorithms used by students, the functionality that is currently missing from these tools.

Keywords: algorithm recognition; program comprehension; program understanding; static program analysis; roles of variables

Received 20 April 2010; revised 6 May 2010
Handling editor: Vladimiro Sassone

1. INTRODUCTION

From different perspectives, program comprehension (PC) can be divided into different subfields: human PC focuses on analyzing the human mental model, and discovering how humans understand programs and uses this information to formulate different models; automated PC concentrates on automatically understanding functionality of programs; program similarity evaluation research investigates programs from a structural point of view and so forth. Algorithm recognition (AR) is a research field, where the problem is to recognize and classify algorithms. Recognizing algorithms implies understanding their functionality, and thus AR is closely related to automatic PC. Although PC has been studied extensively from various perspectives, these studies rarely focus on AR. There still seems to be a lack of an adequate and efficient technique that is able to tackle the AR problem.

Our main goal in AR research is to extend the application of PC in automatic assessment tools. Automatic assessment tools, such as Boss [1], CourseMaster [2] and WebCAT [3], employ many aspects of PC. They are capable of testing the correctness of programs, analyzing the structure and style of programs, evaluating the efficiency of programs, etc. (see the survey by Ala-Mutka [4] for an overview of the field). For their functionalities, these tools are widely used by teachers to reduce their workload, specially in the first and second programming courses, which are usually large. However, these tools are not
able to examine what algorithm is used to solve the given problem. As an example, it is difficult to automatically assess a programming assignment like ‘Write a program that sorts an array using Quicksort that switches to Insertion sort when the sorted area is less than 10 items’. The correct program sorts the given unsorted numbers and outputs the result, but the output does not tell us anything about the applied algorithm. A simple approach to discover the used algorithm would be to check some intermediate states, but this is clumsy and unreliable as students may very well implement the basic algorithm in slightly different ways, for example, by taking the pivot item from the left or right end in Quicksort. This is the AR perspective of PC that can be added to automatic assessment tools. The goal of our research is to address this shortage by developing methods that could automatically recognize algorithms from source code.

Different aspects of PC are widely used in other applications as well. These applications can apply the methods and techniques developed to solve the AR problem as a complement or enhancement of their functionalities. First, AR can be used for source code optimization purposes, which include the task of tuning existing algorithms or replacing them with more efficient ones. This is a challenging problem that arises in connection with developing compilers for parallel processing machines. The main problem is how to identify algorithms that can be parallelized, and how to replace them with new parallel algorithms that compute the same results. See, for example, Metzger and Wen [5] that present a good overview of this area. Second, the problem of maintaining and further developing large legacy codes has been an issue in many businesses and organizations, where the documentation is insufficient, outdated or simply non-existent. Reverse engineering tools, as related to PC methods, are used to partially solve this maintenance problem by analyzing source code, extracting information about structures and dependencies of source code, and presenting this information to maintainers in the form of various diagrams and graphs. By providing higher level information, these tools can help maintainers to do their work, but they are not able to automatically perform the task for the maintainers. Another slightly similar problem related to software maintenance that can use AR techniques is identifying clones in source code. Clones are pieces of code that implement the same thing [6, 7]. Identifying and removing clones are essential parts of code refactoring and software maintenance tasks, as they improve the quality of code remarkably and result in maintenance tasks being much easier. Furthermore, since recognizing and understanding abstractions is a central skill in building programming competences, it is beneficial to be able to identify clones from larger student programs as well. Finally, source-to-source translation methods can also employ AR techniques. Program translation via abstraction and reimplemention (PTAR) [8] is a source-to-source translation approach, which was introduced to address the weaknesses of another approach known as source-to-source translation by transliteration and refinement. The main idea in the PTAR approach is to perform the translation task by first gaining an abstract understanding of what the source program does, and then by reimplementing it in the target language based on the gained understanding. Obtaining an abstract understanding of source code is exactly the goal of AR.

In this paper, we present a method to recognize algorithms. The method is based on static analysis of program code including various statistics of language constructs, software metrics and roles of variables. Algorithms of the learning data are analyzed and converted into a number of different numerical and descriptive characteristics. These characteristics are algorithm specific and thus allow us to distinguish between different algorithms. This information is stored in the knowledge base of the system and is used in the testing phase to recognize previously unseen algorithms. Although we describe a prototype that is designed to recognize common sorting algorithms, the technique in general can be applied to other fields of algorithms as well. An earlier version of this work is reported in our previous work [9, 10]. This paper focuses on reporting the experiment conducted to evaluate the performance of the method by applying it to sorting algorithms. We explain the experiment including the process of data collection and data preparation, and we present the results along with their analysis.

The paper is structured as follows. Section 2 gives an overview of PC field, which is followed by a brief survey of previous work related to the field. This section also defines the problem of AR and outlines the relationship between AR and PC. In Section 3 we describe the method. Section 4 explains how the method has been applied in recognizing sorting algorithms. This is followed by presenting an experiment conducted on this application in Section 5. Section 6 wraps up the paper with a discussion and some conclusions.

2. PC AND AR

PC is a broad concept, which includes all those activities that are somehow related to understanding programs. In this context, understanding programs comprises different perspectives (e.g. human understanding vs. automatic understanding; understanding the functionality of a program vs. discovering the structure of the program; etc.). An essential feature that all the different perspectives share is abstraction. The main goal of the PC-related activities is to present a program in a higher level of abstraction to comprehend it more easily. In the following, we present a classification of PC research from these different perspectives: the different methods they use as well as the different abstractions, point of views and objectives they have. This discussion is intended to clarify the position of AR within PC research.

With respect to the method, PC research can be divided into two broad categories: dynamic methods and static methods. Dynamic methods involve executing the program using some predefined input and examining the output in order to
understand the behavior of the program. Because the program’s output and its behavior depend on the input, different inputs must be used to elaborate the understanding of the behavior. Although the output for a particular input is always exact, outlining a comprehensive behavior of an arbitrary program cannot be guaranteed by using dynamic methods. This follows from the fact that the input determines which path of the program will be executed, and thus finding a set of inputs that executes all possible paths is difficult (the number of paths grows exponentially as the number of decisions increases) or even impossible (infeasible paths exist). Despite these disadvantages, the role of dynamic methods in ensuring that programs work correctly is highly important. Because of this, dynamic methods are widely used in automatic assessment tools to grade students’ work. The correctness of students’ submissions is evaluated by running them with some predefined inputs and comparing the outputs with the expected values (see, for example, [1, 2, 11]). Dynamic analysis can either be used directly for PC purposes, or be applied along with static analysis to test the correctness of programs.

Static methods do not include program execution, but are based on the overall examination of code. In static methods, the structure of the program is analyzed in a general manner, that is, in a sense of using all the possible inputs. Static methods include many different techniques that can be used depending on the focus of the analysis. These techniques include analyzing different features of the code such as control and data flow, the complexity of the program in terms of different metrics and so forth. Because of the thorough and comprehensive analysis that can be achieved by static methods, they are well suited for PC research.

PC research can be divided into subfields based on different perspectives including program functionality, program structure, implementation styles, as well as human mental models in the understanding process. Different subfields provide different outcomes: knowledge about what the program does, how different parts of the program are related to and depend on each other, how the program is implemented and how it is understood by a human. All these outcomes upgrade low level information to a higher level so that it can be used in different applications and be further analyzed for different purposes.

The relationships among PC research and other research fields have been illustrated in Fig. 1. The octagon with a gray background depicts PC as a concept and the rounded rectangles with white background represent research fields somehow related to PC. The arrows in the figure illustrate some of the important relationships among the entities as described in the following.

2.1. Research methods

We present a brief survey on PC-related research techniques, which covers some of the entities shown in Fig. 1. For a more detailed discussion see [10].

**FIGURE 1.** A way to classify different research fields related to PC research and their interrelationship.

*Human PC* aims to discover how humans understand computer programs and build mental models of them. This perspective of PC focuses on explaining the patterns followed by humans in the understanding process. These patterns can be used in computing education in different ways. They can guide teachers in designing programming courses so that it can support students’ understanding of programs and help them in the process of building their mental model (see, e.g. [12]). They can also help learners to enhance their learning process. Moreover, these patterns can be used to build suitable tools to support the process of understanding programs both in software maintaining tasks as well as from the pedagogical point of view (see for example [13]). In Fig. 1, these are reflected by the arrows between Human PC and the neighboring entities.

*Knowledge-based techniques* correspond to Automatic PC in Fig. 1. These techniques focus on automatically understanding the functionality of programs and employ the human understanding models to automatize the PC process. To perform the task, they use their knowledge base, which comprises predefined plans. Plans are frequently used stereotype schemas into which programs that are supported by a knowledge-based PC tool are divided. The process of understanding is converted into the process of matching the unknown program code against the plans. Since the functionality of the plans of the knowledge base is known, if a match between the plans and the unknown program is found, the functionality of the unknown program is identified.

On the basis of the different models of human understanding of programs, knowledge-based techniques are divided into *bottom-up, top-down* and *hybrid* techniques. Knowledge-based techniques mostly work bottom-up (see, e.g. [14]), in which recognition of the target program starts with matching the lower-level, i.e. basic, plans first. An overall understanding can be achieved by connecting the meanings of already recognized
basic plans. In top-down techniques, the recognition process starts with higher-level plans. By knowing the domain of the problem, the right plans that solve the problem can be selected from the library and be compared with the code. These techniques require the specification of the target problem, since the domain must be known (see, for example, [15]). Hybrid techniques use both techniques (see, e.g. [16]).

Because understanding each piece of code requires the existence of the corresponding plan in the knowledge base, a more comprehensive system needs more plans. On the other hand, more plans implies a less efficient system, since the process of searching and matching the plans is costly. Knowledge-based techniques have thus been criticized for being capable of understanding small programs only. To address this issue of scalability and inefficiency, various improvements to these techniques have been suggested including fuzzy reasoning [17].

Reverse engineering techniques are closely related to PC, since being able to reverse engineer programs requires that they be understood first. These techniques are used to recover a program’s high-level design plans, create high-level documentation and diagrams for it, rebuild it, extend its functionality, fix its faults, enhance its functions, among many others. Therefore, the output of reverse engineering techniques include information about both the structure of programs, as well as their functionality. Both forms of this output greatly benefit software maintainers in understanding complex programs. However, reverse engineering techniques have been criticized for only generating abstract specifications and documentations that can help maintainers to understand programs better, rather than providing direct and concise information about what those programs do [18]. From this point of view, these techniques can be considered as methods for analyzing the structure of programs more than as methods for automatically understanding their functionality.

Software visualization (SV) is a research field in software engineering that uses graphics and animation to illustrate different aspects of software. SV systems can be utilized in program development, research and teaching to help programmers and learners understand the structure, abstract and concrete execution as well as the evolution of software. One of the key elements in human PC is the interaction between the user and the system. Visualizations can help create mental models by means of suitable abstractions depicting the target system. In addition, SV tools can provide functionality that enables the user to explore and debug the target system in terms of direct manipulation. This kind of interaction can be utilized, for example, in learning environments in which the idea is to simulate algorithms and this way deepen the understanding of the algorithms [19].

Program similarity evaluation techniques, also called plagiarism detection techniques, are used to discover similarities between programs. These techniques are mainly developed and used at universities to inspect students’ submissions for plagiarism and to prevent it. Their focus is thus on the structure and style of programs, rather than their functionality.

Program similarity evaluation techniques use two different approaches to perform the task: they may use attribute-counting approaches [20, 21] or structure-based approaches [22]. The former approaches are based on computing distinguishing characteristics and various software metrics from the target programs, whereas the latter approaches analyze the structure of the programs. Structure-based approaches can be further divided into string matching-based systems and tree matching-based systems. Structure-based approaches are generally considered more accurate and reliable than attribute-counting approaches, as simple textual modification of two similar programs may cause attribute-counting approaches to fail in detecting their similarities. Structure-based approaches are claimed to be much more tolerant to different types of modifications [23].

In addition to the aforementioned techniques, there are several other PC-related techniques for automatic or semi-automatic program understanding, such as Clone detection techniques [7, 24], Program understanding based on constraints satisfaction [25, 26], Task-oriented program understanding [27], Data-centered program understanding [28] and Understanding source code evolution [29].

2.2. Algorithm recognition

PC can also be considered as the problem of AR (see [9, 10]). Being able to recognize and classify algorithms implies understanding a program. Therefore, finding out what family of algorithms a given algorithm belongs to or what kind of algorithms it resembles, involves PC tasks. The applications of such AR tools include source code optimization, helping programmers and maintainers in their work, examining and grading students’ submissions, and so on. We elaborate on this perspective of PC in the following.

We define AR as activity to outline algorithms from source code. We limit the goal to the automatic source code analysis to identify pieces of code that can be matched against a set of known algorithms. Thus, the aim is to abstract the purpose of the code and recognize, for example, those parts that have potential for improvement. By the given definition, AR is closely related to PC, specially from the functionality point of view. Recognizing an algorithm implies discovering its functionality. On the other hand, by understanding the functionality of an algorithm, we can determine what algorithm is in question; that is, through examining the functionality of, for example, a given sorting algorithm by matching it against the plans in the knowledge base of the system, the algorithm can be recognized to be, as an example, a Bubble sort algorithm (see [14]).

There are different algorithms that can perform the same computational task, such as sorting an array or finding the minimum spanning tree of a graph. For instance, the sorting problem can be solved by using many algorithms.
Although PC has been studied extensively from its different perspective discussed in Section 2, there are other issues that make the AR problem even more challenging. In real-world programs, algorithms are not ‘pure algorithm code’ as in textbook examples. They include calls to other functions, processing of application data and other activities related to the domain, which greatly increases the complexity of the recognition process. The implementation may include calls to other methods or the other functionalities may be included within the code.

In addition to the complexity that results from the aforementioned variations, there are other issues that make the AR problem even more challenging. In real-world programs, algorithms are not ‘pure algorithm code’ as in textbook examples. They include calls to other functions, processing of application data and other activities related to the domain, which greatly increases the complexity of the recognition process. The implementation may include calls to other methods or the other functionalities may be included within the code.

With respect to computational complexity, AR can be regarded to be comparable to many undecidable problems. For example, it is similar to the problem of deciding the equivalency of syntactical definitions of programming languages, which is also known as the equivalency problem of context-free grammars, and is proved undecidable [30]. As will be described in Section 3, we tackle the problem by converting it into the problem of examining the characteristics of algorithms and matching characteristic vectors created from algorithms. In addition, we limit the scope of our work to include a particular group of algorithms. Furthermore, we are not looking for a perfect matching, but aim at developing a method that provides statistically reasonable results.

3. METHOD

Although PC has been studied extensively from its different perspective discussed in Section 2, these studies rarely focus on the AR point of view. In this section, we introduce a new method of recognizing algorithms. The method is based on static analysis of source code including various statistics of language, software complexity metrics, as well as roles of variables. Various characteristics related to these statistics and metrics are identified and extracted from algorithms. These characteristics are selected and designed so that they can describe different algorithms individually. Algorithms are then converted into vectors of characteristics and the problem of AR is transformed into the problem of finding the similarity between the vectors as described in what follows.

The recognition process consists of two phases: the learning phase and the testing phase. In the learning phase, the system is trained by the algorithms randomly collected as the training samples. The training samples are converted into characteristic vectors, each vector labeled by the correct type of the corresponding algorithm. These vectors are stored in the database. In the testing phase, a set of previously unseen algorithms including both those algorithms of the same type as in the training set, as well as other types of algorithms or irrelevant code are also converted into characteristic vectors. These unknown vectors are then compared with those from the knowledge base of the system. If a match is found, the unknown vector is labeled with the type of the algorithm from the learning data to which it matches, otherwise, it is labeled with an Unknown type. Figure 2 shows a simplified model of the method. In the figure, the learning and testing phases are illustrated by rectangles and the inputs and outputs of the process are represented by ellipses. If an algorithm is falsely labeled by the system as Unknown, for example, due to having a smaller or larger number of some characteristic, a teacher can manually assign the correct type to the algorithm in the database, extending the knowledge base of the system. We discuss the possibilities for involvement of a teacher in more detail later in this section when we elaborate on the method.

There are two types of characteristics: the numerical characteristics and the descriptive characteristics. The numerical characteristics are those statistics and metrics that can be expressed as numerical values. Descriptive characteristics describe some distinctive properties of the algorithm in question and are not numerical information in nature. The numerical and descriptive characteristics used in the method are listed in Table 1. The abbreviations in the table are used to refer to the corresponding characteristic in the next sections of the paper including Table 3 explained in Section 4 (which shows the
### TABLE 1. The numerical and descriptive characteristics computed to convert algorithms into characteristic vectors.

<table>
<thead>
<tr>
<th>Numerical characteristics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS</td>
<td>The number of assignment statements in the algorithm.</td>
</tr>
<tr>
<td>LoC</td>
<td>The lines of code of the algorithm.</td>
</tr>
<tr>
<td>MCC</td>
<td>The cyclomatic complexity, i.e. McCabe complexity of the algorithm (see [32]).</td>
</tr>
<tr>
<td>( N_1 )</td>
<td>The total number of operators in the algorithm.</td>
</tr>
<tr>
<td>( N_2 )</td>
<td>The total number of operands in the algorithm.</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>The number of unique operators in the algorithm.</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>The number of unique operands in the algorithm.</td>
</tr>
<tr>
<td>( N )</td>
<td>Program length (( N = N_1 + N_2 )).</td>
</tr>
<tr>
<td>( n )</td>
<td>Program vocabulary (( n = n_1 + n_2 )).</td>
</tr>
<tr>
<td>NoV</td>
<td>The number of variables in the algorithm.</td>
</tr>
<tr>
<td>NoB</td>
<td>The number of blocks in the algorithm. A block refers to a sequence of statements wrapped in curly braces, e.g. a method or a control structure (loops and conditionals).</td>
</tr>
<tr>
<td>NoL</td>
<td>The number of loops. Supported loops are for loop, while loop and do while loop.</td>
</tr>
<tr>
<td>NoNL</td>
<td>The number of nested loops in the algorithm.</td>
</tr>
<tr>
<td>Descriptive characteristics</td>
<td>Description</td>
</tr>
<tr>
<td>Recursive</td>
<td>Is the algorithm recursive or non-recursive.</td>
</tr>
<tr>
<td>Tail recursive</td>
<td>Is the algorithm tail recursive or not.</td>
</tr>
<tr>
<td>Auxiliary array</td>
<td>Does the algorithm use an auxiliary array.</td>
</tr>
<tr>
<td>Roles of variables</td>
<td>The roles of the variables of the algorithm.</td>
</tr>
</tbody>
</table>

The value of the numerical characteristics computed from the sorting algorithms of the learning data. The characteristics illustrated by \( N_1, N_2, n_1, n_2, N \) and \( n \) are the Halstead metrics [31] that were introduced for measuring software complexity. Introduced by McCabe, cyclomatic complexity, also known as McCabe complexity, is also a metric used to measure the complexity of a program, which describe the complexity as a graph using control flow of the program (see [32]). The number of assignment statements and lines of code along with the other numerical characteristics shown in Table 1 are widely used in program similarity evaluation techniques.

The descriptive characteristics listed in Table 1 are self-explanatory. Roles of variables are explained in more detail at the end of this section.

#### 3.1. Other characteristics

In connection with aforementioned characteristics, the following characteristics are computed as well.

1. The information related to each of the blocks and loops of a program, such as their starting and ending lines, their length, as well as interconnection between them. This provides the information about the number of blocks and loops and how they are positioned in relation to each other.

2. The pattern in which the value of loop counters in the program are initialized and updated. This information determines the incrementing and decrementing loops in the program. If the loop counter increases after each iteration, the loop is said to be an incrementing loop and if the value decreases, the loop is considered to be a decrementing one.

3. Both direct and indirect dependencies between each variable that appears in the program and all the other variables. We used the following common definition of direct and indirect dependencies: variable \( i \) is said to be directly dependent on variable \( j \), if \( i \) gets its value directly from \( j \). Moreover, if there is a third variable \( k \) on which \( j \) is dependent (either directly or indirectly), then \( i \) also becomes indirectly dependent on \( k \). A variable can be both directly and indirectly dependent on another one.

As we will discuss in Section 4, these characteristics are used to identify different patterns that can facilitate the recognition process.

#### 3.2. Recognition process

As Table 1 shows, there are a total of 17 characteristics computed for algorithms. Therefore, algorithms are processed and stored in the database as 17-dimensional characteristic vectors. The algorithms supported by the system, that is, the algorithms that exist in the knowledge base of the system, constitute a vector space and previously unseen algorithms are recognized by identifying their position in this vector space. This approach allows us to convert the complex task of AR to the simpler task of pattern recognition, which basically consists of converting algorithms of learning data into characteristic vectors.
To guide the recognition process, we build a decision tree, which includes a test on the characteristics in each node. The tests comprise both examination of the frequency of occurrences of the numerical characteristics in an algorithm, as well as investigation of the descriptive characteristics of the algorithm.

To set up the Analyzer (the implemented tool that performs the is the training phase. Once it has been decided what kind of algorithms the Analyzer is desired to recognize, many different implementations of these algorithms will be analyzed with respect to the aforementioned characteristics. The resulting characteristic vectors labeled with the correct type are stored in the database. Thereafter, the Analyzer has the following information about each algorithm: the type of the algorithm, the descriptive characteristics of the algorithm and the range of values each numerical characteristic can have (within the learning material). The second phase is the testing phase, where the Analyzer is required to identify new unseen algorithms based on what it has learned in the training phase. In this phase, when the Analyzer encounters a previously unknown test example (test case), it first computes its numerical and descriptive characteristics. After this, the Analyzer retrieves the information of already known algorithms of the learning data from the database. From this information, the minimum and maximum limits of the numerical characteristics are counted. If the value of some numerical characteristics of the test case does not fit into the range of the corresponding minimum and maximum values of the learning data in the database, the test case is classified to be *Unknown*. Moreover, a message is outputted including the information about the numerical characteristic that does not fit into the permitted range, as well as whether its value is smaller or greater than the corresponding numerical characteristic of the learning data. Finally, the information of the test case is stored in the database and the recognition process for this test case is terminated. Otherwise, if the test case passes this stage, the examination process continues with examining the descriptive characteristics according to the steps given by the decision tree up to a leaf node. Depending on the results of this examination, either the process ends with labeling the test case with the type of the algorithm that is indicated by the leaf of the decision tree, or alternatively, if the characteristics of the test case do not conform to those of algorithms from the learning data, it is assigned an Unknown type. The information of the algorithm is stored in the database in both cases. These different phases of the recognition process are illustrated with Fig. 2.

It is possible that an algorithm of the testing data is implemented using such programming style that, for example, the value of some of its numerical characteristics is smaller or greater than those of the corresponding algorithms of the learning data. This will result in rejecting the algorithm, although it is a positive one. In these cases, a teacher can manually inspect the algorithms classified as *Unknown* to verify whether they are indeed true negative algorithms or incorrectly labeled *Unknown*. If a rejected algorithm turns out to be positive, the teacher can correct its type in the database. This is illustrated in Fig. 2 with interaction between the teacher and the knowledge base of the system. The knowledge of the Analyzer can be extended in this way. Next time, the algorithm implemented using the same programming style will be recognized as the correct type.

Roles of variables are part of the descriptive characteristics used in the process of recognizing algorithms. Due to their importance in the process, we briefly explain roles of variables in the following. More information on roles can be found from [33].

### 3.3. Roles of variables

The concept of roles of variables is based on the idea that each variable used in a program plays a particular role that is related to the way its value is initialized and updated. Roles of variables are specific patterns how variables are used in source code. As an example, a variable that is used to store the most desirable value found so far from a set of elements has a most-wanted holder role. As Sajaniemi [33] argues, roles of variables are part of programming knowledge that have remained tacit. Experts and experienced programmers have always been aware of existing variable roles and have used them, although the concept has never been articulated. Giving an explicit meaning to the concept can make it a valuable tool that can be used, for example, in teaching programming to novices. Roles of variables can enhance novices’ understanding of the different ways in which variables can be used in a program. The concept can also offer an effective and unique tool to analyze a program with different purposes. In this work, we have extended the application of roles of variables by applying them in the problem of AR.

Roles are cognitive concepts, which implies that human inspectors may have a different interpretation of the role of a single variable. However, roles can be analyzed automatically using data flow analysis and machine learning techniques [34, 35].

As described in [33], 99% of all variables in novice-level procedural programs can be covered using only nine roles. Currently, there are 11 roles recognized that cover all variables in novice-level programs in object-oriented, procedural and functional programming. Table 2 shows these 11 roles along with their descriptions.

To further clarify the concept of roles of variables, we present the simple example of Fig. 3, which shows a typical implementation of a Selection sort algorithm in Java. The algorithm uses five variables with the following roles. In the figure, variables *i* and *j* are loop counters, i.e. the integer-type variables that are used to control the iterations of the loops.
TABLE 2. The roles of variables and their descriptions.

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stepper</td>
<td>A variable that systematically goes through a succession of values, e.g. values stored in an array.</td>
</tr>
<tr>
<td>Temporary</td>
<td>A variable that holds a value for a short period of time appears in temporary role.</td>
</tr>
<tr>
<td>Organizer</td>
<td>A data structure holding values that can be rearranged is a typical example of the organizer role.</td>
</tr>
<tr>
<td>Fixed value</td>
<td>A variable that keeps its value throughout the program. The fixed value role can be thought as a final variable in Java which is immutable once it has been assigned a value.</td>
</tr>
<tr>
<td>Most-wanted holder</td>
<td>A variable that holds a most desirable value that is found so far.</td>
</tr>
<tr>
<td>Most-recent holder</td>
<td>A variable that holds the latest value from a set of values that is being gone through, and a variable that holds the latest input value.</td>
</tr>
<tr>
<td>One-way flag</td>
<td>A variable that can have only two values and once its value has been changed, it cannot get its previous value back again.</td>
</tr>
<tr>
<td>Follower</td>
<td>A variable that always gets its value from another variable, i.e. its new values are determined by the old values of another variable.</td>
</tr>
<tr>
<td>Gatherer</td>
<td>A variable that collects the values of other variables. A typical example is a variable that holds the sum of other variables in a loop, and thus its value changes after each execution of the loop.</td>
</tr>
<tr>
<td>Container</td>
<td>A data structure into which elements can be added or from which elements can be removed. For example, all Java data structures that implement Collection interface.</td>
</tr>
<tr>
<td>Walker</td>
<td>Can be assigned to a variable that is used for going through or traversing a data structure.</td>
</tr>
</tbody>
</table>

```java
// i and j: steppers, min: most-wanted holder
// temp: temporary, numbers: organizer
for (int i = 0; i < numbers.length-1; i++){
    int min = i;
    for (int j = i+1; j < numbers.length; j++){
        if (numbers[j] < numbers[min]){
            min = j;
        }
    }
    int temp = numbers[min];
    numbers[min] = numbers[i];
    numbers[i] = temp;
}
```

FIGURE 3. An example of stepper, most-wanted holder, temporary and organizer roles in a typical Java implementation of Selection sort algorithm.

These variables have the stepper role, as all variables used as a loop counter typically do. Variable `min` stores the position of the smallest element found so far from the array, and thus has the most-wanted holder role. Variable `temp` demonstrates the temporary role, as it is used in the swap operation to store the smallest element so far found in the array for a short period of time. Finally, `numbers` has the organizer role, as it holds the values to be sorted.

4. RECOGNIZING COMMON SORTING ALGORITHMS

We applied our method to the five commonly used sorting algorithms: Insertion sort, Bubble sort, Selection sort, Quicksort and Mergesort, in order to illustrate the feasibility and performance of the method. To restrict the scope of our work, we selected sorting algorithms owing to several reasons. Sorting algorithms are a widely discussed topic in computing education. They are used as examples in programming courses as well as in courses on data structure and algorithms. Consequently, a large number of sorting algorithms are easily accessible from textbooks, the Web, students’ work as well as in handouts of the related courses. Sorting algorithms are also easy to analyze as they are not too complex. Moreover, there are many different sorting algorithms that perform the same computational task. This feature makes discriminating sorting algorithms a challenging task. Finally and most importantly, as we shall discuss later, sorting algorithms include both very similar and yet very different algorithms with respect to their characteristics. As an example, Insertion sort and Bubble sort algorithms are very similar, whereas Mergesort and Bubble sort algorithms are quite different. This is a very beneficial feature of sorting algorithms from the perspective of our method, since it results in the performance of the method to be tested more thoroughly.

At the beginning of the study, we manually analyzed many different versions of the aforementioned sorting algorithms. On the basis of the results of these analyses, we posited a hypothesis that the numerical and descriptive characteristics mentioned in Table 1 along with the other data discussed in Section 3 are sufficient to describe the sorting algorithms and could be used to discriminate between them. The problem that we were trying to solve was whether a new unknown algorithm from the testing data could be reliably enough identified by comparing its characteristics with the corresponding information of the algorithms from the learning data.
TABLE 3. The numerical characteristics of the five sorting algorithms of the learning data. The first number indicates the minimum and the second number the maximum of the value of the corresponding characteristic (see Table 1 for explanation on the abbreviations).

<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>N1</th>
<th>N2</th>
<th>n1</th>
<th>n2</th>
<th>N</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>4/6</td>
<td>2/2</td>
<td>4/5</td>
<td>8/11</td>
<td>13/21</td>
<td>4/6</td>
<td>40/57</td>
<td>47/58</td>
<td>3/6</td>
<td>2/4</td>
<td>87/115</td>
<td>5/10</td>
</tr>
<tr>
<td>Bubble</td>
<td>5/6</td>
<td>2/2</td>
<td>4/5</td>
<td>8/11</td>
<td>15/21</td>
<td>4/5</td>
<td>46/55</td>
<td>49/57</td>
<td>4/6</td>
<td>2/4</td>
<td>95/112</td>
<td>6/10</td>
</tr>
<tr>
<td>Selection</td>
<td>5/6</td>
<td>2/2</td>
<td>5/6</td>
<td>10/12</td>
<td>16/25</td>
<td>4/5</td>
<td>47/61</td>
<td>51/64</td>
<td>4/6</td>
<td>2/5</td>
<td>98/125</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>161/210</td>
<td>11/24</td>
</tr>
<tr>
<td>Mergesort</td>
<td>7/13</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>11/16</td>
<td>5/10</td>
<td>190/279</td>
<td>16/26</td>
</tr>
</tbody>
</table>

Next, we developed a prototype Analyzer that is able to automatically compute all the data and convert algorithms into characteristic vectors. The Analyzer is implemented in Java and the current version is able to process source code written in Java. It parses the code, counts all its numerical and descriptive characteristics shown in Table 1 and analyzes all the related data (see Section 3). The Analyzer stores the data into a database consisting of the following four tables: Algorithm, Block, Variable and Dependency. We also integrated a software into the Analyzer for automatic detection of roles of variables (see [34]).

On the basis of the characteristics discussed in Section 3, we also computed the following characteristics which are specific to sorting algorithms:

1. **In-place**: Does the algorithm need extra memory in order to carry out the sorting.
2. **OLIID** (outer loop incrementing inner decrementing): Whether the outer loop of the two nested loops used in the algorithm is incrementing and the inner is decrementing.
3. **ILIOL** (inner loop counter initialized to outer loop counter’s value): Whether the inner loop counter of the two nested loops used in the algorithm is initialized to the value of the outer loop counter.
4. **Temporary**: Whether the algorithm includes a variable playing the temporary role.
5. **Most-wanted holder**: Whether the algorithm includes a variable playing the most-wanted holder role.

The characteristics OLIID and ILIOL can easily be computed using the characteristic NoNL from Table 1 and the two characteristics numbered (2) and (3) in Subsection 3.1, respectively. Likewise, the in-place sorting algorithms can be identified by examining that these algorithms do not use an auxiliary array (Table 1). Finally, the existence of the roles Temporary and Most-wanted holder in an algorithm can simply be verified by going through all the roles that the role analyzer has detected for the variables of that algorithm.

As we will see later, these characteristics are good discriminators in the case of sorting algorithms. However, for other types of algorithms, we presumably need to use additional characteristics either derived from the characteristics described in Section 3, or computed separately. For example, we need to discover what roles of variables are typically used in other fields of algorithms, which can separate them from each other.

We randomly collected the learning data consisting of 70 different versions of the five sorting algorithms for the analysis. The process of data collection is discussed in the next section where we present the experiment. We ran the learning data through the Analyzer and stored their numerical and descriptive characteristics produced by the Analyzer in the database. Table 3 shows the resulted numerical characteristics. For each characteristic in the table, the first number shows the minimum value of the characteristic for the corresponding algorithm of the learning data, and the second number depicts the maximum value. A quick glance at the table reveals that these numerical characteristics divide the five sorting algorithms into two groups, a group with a smaller number of numerical characteristics consisting of the Insertion sort, Bubble sort and Selection sort algorithms, and a group with larger number of numerical characteristics comprising the Quicksort and Mergesort algorithms. Therefore, the recursive and non-recursive sorting algorithms can already be differentiated by their numerical characteristics alone.

In the next step, based on the learning data, we constructed the decision tree of Fig. 4 for classifying the sorting algorithms. The decision tree has 9 internal nodes (including the root) and 10 leaves. The internal nodes are represented by rectangles with white background and the leaves are depicted by rectangles with a gray background. Each internal node includes a test. The outcome of a test in an internal node determines which child of that node will be visited next. There are arcs from each internal node to its children (or to a leaf) annotated by the outcome of the test in the corresponding node. Each leaf is labeled by a type of the sorting algorithms, or by the word ‘Unknown’.

The characteristics which can best discriminate between the sorting algorithms are used in the root and the near nodes. Thus, the test ‘Recursive algorithm?’ in the root is the best discriminator. It divides the sorting algorithms into two groups—recursive and non-recursive—with the accuracy of 100%, as all the Quicksort and Mergesort algorithms of the learning data are recursive and all the Insertion sort,
Bubble sort and Selection sort algorithms are non-recursive. The tests in both children of the root examine whether the numerical characteristics of the recognizable algorithm are within those values retrieved from the database for the recursive or non-recursive algorithms. If the numerical characteristics are not within the permitted limit, the algorithm is labeled with ‘Unknown’ and the process of recognition is terminated. Otherwise, in the case of recursive algorithms, the next visited node is the one labeled with ‘Temporary role?’, where the performed test is whether the algorithm includes a variable with a temporary role. Quicksort algorithms typically include the temporary role, which appear in connection with swap operations to rearrange the elements of an array. However, Mergesort algorithms do not usually include a temporary role, since due to performing the merge operation, there is no need to perform swap operations. Nevertheless, analysis of the Mergesort algorithms revealed that some of them do use swap operations and thus include the temporary role. Therefore, if the outcome of the test is positive, the process is continued by visiting the left child of the node, which is labeled with ‘In-place?’. The test performed in this node is whether the recursive algorithm performs the sorting without making use of an auxiliary array. Again, using an auxiliary array is typical for Mergesort algorithms, but does not appear in Quicksort algorithms. However, some of the Mergesort algorithms work In-place and thus there is a need to perform another test. Next, the left child, which is the last node of this subtree and examines whether the recursive algorithm is tail recursive, is visited. All the analyzed Quicksort are tail recursive, which is not the case with the analyzed Mergesort algorithms.

On the right subtree of the root, the three non-recursive algorithms are examined. If the numerical characteristics of the recognizable non-recursive algorithm are within the permitted limit, the left child is visited with testing the existence of a variable with a most-wanted holder role. Analyzing the non-recursive sorting algorithms of the learning data showed that all the Selection sort algorithms include the most-wanted holder, whereas none of the Insertion sort and Bubble sort algorithms do. This is because Insertion sort and Bubble sort algorithms do not include selection of a min (or max) element from a...
list (see Table 2 for the definition of the roles). If the target non-recursive algorithm does not include the most-wanted role, the next visited node is the node labeled with ‘OLIID?’, which stands for ‘Outer Loop Incrementing Inner Decrementing’. All the Insertion sort and Bubble sort algorithms included two nested loops. In the analyzed Insertion sort algorithms, while stepping through the elements of an array, the value of the loop counter of the outer loop increases, whereas the value of the loop counter of the inner loop decreases. Although the Bubble sort algorithms mostly work so that the outer loop is decrementing and the inner loop is incrementing, some of them work like the Insertion sort algorithms. Therefore, further test is needed to discriminate between this part of the Bubble sort algorithms and the Insertion sort algorithms. With positive outcome of the test performed in this node, the next node to visit is the node labeled with ‘ILIOL?’, which is an abbreviation for ‘Inner Loop counter Initialized to Outer Loop counter’s value’. This is the case in the Insertion sort algorithms; but in Bubble sort algorithms, the inner loop counter is initialized whether to zero or to the length of the array.

Note that the node labeled with ‘Numeric characteristics within permitted limit?’ does not include a single characteristic of an algorithm, but rather the test for investigating all the numerical characteristics of the algorithm. It filters out those algorithms with less or more numerical characteristics, labels them with the Unknown type, stores them in the database and gives a message about the particular characteristic(s) that does not fit within the permitted limits. If these filtering nodes are not taken into account, the number of internal nodes that include a test using a single characteristic would be seven, and the maximum depth of the tree, which is five, would be reduced to four.

Characteristics like OLIID or ILIOL may appear very specific lacking the value of generalizability. They are indeed specific. They are, for example, not applicable to the Quicksort algorithms nor to the Mergesort algorithms, because these algorithms either do not have two nested loops or they cannot be separated by these characteristics. Moreover, the fact that these characteristics are located near the leaves reflects their specificity. These characteristics, however, provide a valuable means to separate the Insertion sort algorithms from the Bubble sort algorithms. Specially, the role of ILIOL is indispensable in this regard. In terms of the other characteristics, these two algorithms are so similar that it seems very difficult to otherwise differentiate between them (Table 3 and Figure 4). While these characteristics are specifically computed to be used in recognition of the sorting algorithms, their applicability as good discriminators for other fields of algorithms remains to be seen in future work.

As can be seen from Fig. 4, the roles of variables play an important and distinctive role in the process. Specially the most-wanted holder role that separates the Selection sort algorithms from the Insertion sort and Bubble sort algorithms is highly important, since the other characteristics of these algorithms are quite similar and thus cannot separate them reliably (Table 3).

5. EXPERIMENT

We evaluated the performance and accuracy of the method and the decision tree by empirical tests. We conducted an experiment on the five aforementioned sorting algorithms. In this section, we explain how the experiment was carried out, discuss the process of data collection and preparation, and present and analyze the results.

5.1. Data collection

We randomly collected a data set consisting of 287 algorithms in two phases. Part of these algorithms were used as the learning data and part of them as the testing data. The process of gathering these two data sets is discussed in the following.

5.1.1. Learning data

In the first phase, a total of 70 algorithms were collected as the learning data. The algorithms of learning data were gathered from various textbooks on data structures and algorithms, as well as from course material available on the Web. Some of the Insertion sort and Quicksort algorithms were from authentic students’ submissions. Algorithms in the learning data included only the five types of sorting algorithms mentioned above. All these algorithms were run by the Analyzer and the related numerical and descriptive characteristics were stored in the database. After verifying that each of these algorithms was actually of the claimed type, they were labeled by their type in the database.

The number and the percentage of each algorithm in the learning data is shown in the second column of Table 4. In the table, the first number indicates the number of the corresponding algorithm, and the second number in the parentheses shows the percentage of it from the total number of the algorithms of the learning data. When collecting the learning data, the exact number of each algorithm in comparison to other algorithms was not particularly planned. The criterion in mind was that all algorithms should have adequate number of representatives in the learning data. As shown in Table 4, the number of algorithms are more or less the same, and perhaps the difference between them reflects the fact that how commonly Java implementation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Learning data</th>
<th>Testing data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>17 (24%)</td>
<td>35 (16%)</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>11 (16%)</td>
<td>30 (14%)</td>
</tr>
<tr>
<td>Selection sort</td>
<td>14 (20%)</td>
<td>29 (13%)</td>
</tr>
<tr>
<td>Quicksort</td>
<td>16 (23%)</td>
<td>23 (11%)</td>
</tr>
<tr>
<td>Mergesort</td>
<td>12 (17%)</td>
<td>22 (10%)</td>
</tr>
<tr>
<td>Other</td>
<td>-</td>
<td>78 (36%)</td>
</tr>
<tr>
<td>Total</td>
<td>70</td>
<td>217</td>
</tr>
</tbody>
</table>
of each algorithm appears in the sources where they were collected from.

5.1.2. Testing data
In the second phase, 217 algorithms were collected as testing data. As was the case in the learning data, the testing data were also collected randomly from textbooks, from the Web and from the authentic students’ submissions (mostly Insertion sort and Quicksort). The testing data mainly consisted of the five sorting algorithms, but included other algorithms as well. These algorithms, which we refer to as Other, are discussed below.

The majority of Other algorithms were other sorting algorithms such as Heapsort, Shell sort, Radix sort and some other not so commonly used sorting algorithms (like Topological sorting, etc.). Moreover, Other algorithms included various enhanced types of the five sorting algorithms. These enhanced sorting algorithms include some additional code that changes the essence of the corresponding algorithms. As a result, the enhanced algorithms cannot be regarded as the basic form of the original algorithms. A typical example of these enhanced algorithms is the hybrid algorithm of Quicksort and Insertion sort, where sorting is carried out using Quicksort as long as the number of elements to be sorted is above some prefixed number (e.g. 10), and below this number Insertion sort is used. Another example is Cocktail sort, also known as Bidirectional Bubble sort, which is basically a Bubble sort that uses an additional loop and conditional statements to speed up sorting. In addition to these, Other algorithms contained some completely different algorithms as well, including binary search algorithms, or purely application data, such as small parts of interface code or code written to test how some algorithm works.

The third column of Table 4 shows the type and the number of the different algorithms collected as the testing data. Here again, the numbers in the parentheses indicate the percentage of the corresponding algorithm from the total number of the algorithms of the testing data. The sorting algorithms in the testing data were collected based on their availability, keeping in mind that to carry out the test, there should be a sufficient number of each type of algorithms. Similar to the learning data, these numbers of sorting algorithms perhaps indicates how frequently they are used as examples in the used sources.

The aforementioned method of separating learning and testing data is commonly known as Holdout Method (see [36]). In this method, the proportion of the data used as the training and testing set can be decided by the person who conducts the experiment. In order to avoid an overfitting problem and achieve the best generalization performance, machine learning and classification methods often follow a commonly used pattern to split the data set into a training and testing set. For example, the proportion of 50–50 or 70 percent for the training set and 30 percent for the testing set is widely used. We use a comprehensive testing data in order to evaluate the performance of the method as accurately as possible.

5.2. Data preparation
The collected algorithms, both the learning and testing data, were run and verified as being the type they were said to be. In order to run each algorithm and see its output, the Java main method was added to algorithms (if the algorithm did not already include one) along with a print statement. These print statements were removed from the algorithms before using them as learning and testing data.

Some algorithms, especially those collected from the Web, included some extra code, e.g. for testing the algorithm or calling methods from other classes. Since our method, in its present state, cannot process these kinds of application data, these extra codes were removed from the algorithms leaving the implementation of the algorithm itself untouched.

We integrated the role analyzer developed by Bishop and Johnson [34] into the Analyzer in order to detect roles of variables in the programs. This role analyzer requires that all variables of the program are annotated using special tags. If some variable is left untagged, the role analyzer simply ignores it. Along with the tags, the name of the variable must be provided. Mentioning the role that the user believes the variable is playing is optional. The idea behind the requirement of annotating variables and providing their roles is most probably the fact that the role analyzer is designed specially to be used in educational purposes: making the students tag variables and provide roles for them. When the role analyzer outputs a role for each annotated variable, it is easy to compare the role provided by the user with that outputted by the role analyzer. If the role outputted by the role analyzer differs from that provided by the user, the role analyzer explains its answer. We analyzed all variables in all algorithms and assigned a suitable role to them along with their name and the required tags. Although the role analyzer does not require the roles of the variables to be provided by the user, we provided the roles of all variables along with their name as well in order to be able to analyze and find the exact reason if our tool fails to recognize some algorithm correctly. Since failure in recognizing the roles of variables in our work would result in failure in recognizing specially Selection sort and Quicksort (Fig. 4), this feature of the role analyzer enables us to follow carefully whether the roles provided by it corresponds to those annotated by us, and if not, how the role analyzer justifies it. If necessary, it is straightforward to make the annotating process automatic.

5.3. Testing strategy
When testing the method, we distinguish the following cases. True positive indicates a case where the analyzer correctly recognizes an algorithm that belongs to the target set of the five sorting algorithms. True negative correspondingly indicates rejecting an algorithm that is not among the target set. False positive denotes that an algorithm not belonging to the target set is incorrectly recognized as one belonging to the set, and
false negative correspondingly an algorithm belonging to the set, which is not recognized as such or is recognized as another member of the set. True positive and true negative cases indicate successful passing of tests and accurate performance of the Analyzer, whereas false positive and false negative cases mean failure in tests and poor performance of the Analyzer. In the following, we further illustrate these cases with examples and outputs of the Analyzer in each case.

An example of true positive is a Quicksort recognized as such. In these cases, the Analyzer simply outputs ‘The algorithm is a Quicksort’. If a binary search algorithm, for example, is recognized as Unknown, a true negative case is in question. In this case, the message would be something like the following: ‘The algorithm seems to be a recursive algorithm, that has the following characteristics out of the permitted limit: Number of operators is below the permitted limit, Number of operands is below the permitted limit, Assignment statement is below the permitted limit, Line of code is below the permitted limit, Program length is below the permitted limit’. If, on the other hand, a Heapsort is falsely recognized as a Mergesort, we have the false positive case with the Analyzer providing the following message: ‘The algorithm is a Mergesort’. As we shall discuss below, the false positive cases are very rare. Finally, an example of false negative is when a Quicksort is falsely recognized as a Mergesort, for example, due to the failure of the role analyzer to recognize a temporary role (Fig. 4). The output of the Analyzer in this case would be ‘The algorithm is a Mergesort’. Another typical example of the false negative cases is when, for example, a Selection algorithm is not recognized as such (i.e. recognized as Unknown) because some of its numerical characteristics are below or above those Selection sorts known to the Analyzer, i.e. those which exist in the learning data.

5.4. Results

In the experiment, a total of 217 different algorithms including the five sorting algorithms, other sorting algorithms, as well as other types of algorithms were tested. Almost 86% of the tests were passed successfully (either the true positive or the true negative cases) and 14% of them failed (the false positive and the false negative cases). The overall test results are shown in Fig. 5, where the portion of each true positive, true negative, false positive, and false negative cases from the overall results.

![Overall test results](image)

Fig. 5. Portion of the true positive, true negative, false positive and false negative cases from the overall results.

was 29 (the false negative cases, 13%), whereas the number of those algorithms not belonging to the set but incorrectly recognized as belonging to the set was 2 (the false positive cases, 1%). The accuracy of the technique when viewed in the context of helping teachers in assessing students’ work—which is the main application of the method—appears reasonable: 86% of the students’ submissions do not need to be dealt with manually. This means a significant reduction in workload, especially in large programming courses. In the following, we further elaborate on the results.

False negative cases are the most common error the Analyzer makes. Simply by adding some extra and irrelevant code, for example, a swap operation or two, the Analyzer can be made to believe that the algorithm is not the type that it actually is. These cases, however, usually result from idiomatic code and, as we shall discuss in Section 6, our method is currently not tolerant to irrelevant changes that are intentionally made to the code, and expects the target algorithms to be implemented in a well-established way.

The interesting observation to make here is the false positive cases, which were only two cases out of the 217 test cases. The first of these two cases was a Heapsort algorithm recognized as a Mergesort and the second was a Quicksort algorithm recognized as a Quicksort (there were several different implementations of Heapsort algorithms in the testing data, but just one Quicksort algorithm). This very low number of false positive cases, which is less than one percent of the testing data, suggests that the numerical and descriptive characteristics along with the roles of variables provide such specific and detailed information about an algorithm that makes it very rare for other algorithms to falsely be recognized as this one. In order to occur, a false positive case requires that the recognizable algorithm passes the steps shown in the decision
FIGURE 6. The test results (the true positive and the false negative cases only) for the five sorting algorithms.

tree (Fig. 4), which cannot happen so often compared with false negative cases. For example, in order to recognize an algorithm falsely as a Selection sort, the algorithm has to have the right number of numerical characteristics and, additionally, include at least one variable playing the most-wanted holder role. This is not as likely as a true Selection sort being labeled as Unknown due to reasons like using extra variables or failure in producing a correct role (this is further discussed in Section 6).

The test results for the five sorting algorithms, separated based on the true positive and the false negative cases, are shown in Fig. 6. For each sorting algorithm, the first column in blue color shows the total number of the corresponding algorithm, the second column in green shows the number of the true positive cases and the third column in red shows the number of the false negative cases.

As Fig. 6 shows, the non-recursive sorting algorithms are generally recognized better than the recursive ones (91.4%, 83.3% and 82.8% true positive cases for the Insertion, Bubble and Selection sort algorithms, respectively, as opposed to 65.2% for the Quicksort and 72.7% for the Mergesort algorithms). These percentages are computed in the end of this section and are shown in Table 5). This has occurred barely by chance. The reason for this is that the more complex the algorithms are, the more their implementations are likely to vary. Quicksort and Mergesort are more complex than the other three non-recursive sorting algorithms. As a result, they can be implemented in more different ways using different strategies in lower level details. For example, as discussed in Section 2, there are several different strategies to carry out pivot selection and partitioning in Quicksort. Existence of different options in implementing algorithms results in a higher possibility for different implementations of those algorithms to differ from each other, especially in their numerical characteristics. For example, as will be discussed in the following, all those 27% false negative cases for the Mergesort algorithms are due to the difference between the algorithms in their numerical characteristics. This will pose a challenge to us in the future when we further develop our method to cover other algorithms. We will have to come up with a more sophisticated method that performs better with more complex algorithms.

Another observation to make from Fig. 6 is that in the case of recursive algorithms, the Mergesort algorithms are recognized better in comparison with the Quicksort algorithms (73 vs. 65%). This can be explained as follows. The reason for the failure in recognizing the Mergesort algorithms is that the number of the numerical characteristics of these false negative cases (six Mergesort algorithms) has been out of the permitted limit. The same reason applies for the false negative cases of the Quicksort algorithms, but in addition, there is another reason beyond this in this case, namely, failure in recognizing the temporary role. Five out of eight false negative cases in the Quicksort algorithms occur because of the difference between
the number of the numerical characteristics of these Quicksort algorithms and those in the learning data, and the remaining three of them because of the absence of the temporary role in the algorithms or failure in recognizing it. As illustrated in the decision tree in Fig. 4, when the role analyzer fails to recognize the temporary role, a Quicksort will be recognized falsely as a Mergesort. On the other hand, after a bit of tuning, the role analyzer performed very well in the case of the Selection sorts. The role analyzer detected all most-wanted holder roles correctly, resulting in a fair percentage of correct recognition of the Selection sorts, i.e. 83% (Fig. 4). All the false negative cases of the Selection sorts (17%) occurred due to the difference between the numbers of the numerical characteristics.

The results of the true positive and false negative cases for the sorting algorithms depicted in Fig. 6 can be also presented as percentages. These percentages are metrics called the true positive rate (equivalently, sensitivity or recall), the false negative rate and the precision. These metrics are defined based on the true positive, false positive and false negative cases as follows [36]. The true positive rate is the proportion of the positive algorithms that are correctly recognized. It can be computed as follows:

$$\text{True positive rate} = \frac{\text{True positive}}{\text{True positive} + \text{False negative}}.$$ 

The false negative rate is the proportion of the positive algorithms that are incorrectly recognized as negative. It is computed as follows:

$$\text{False negative rate} = \frac{\text{False negative}}{\text{True positive} + \text{False negative}}.$$ 

Precision is the proportion of the actual positive algorithms to all algorithms recognized as positive. It thus can be counted by the following formula: 

$$\text{Precision} = \frac{\text{True positive}}{\text{True positive} + \text{False positive}}.$$ 

Table 5 shows the value of true positive rate, false negative rate and precision for the five sorting algorithms. The number of the false positive cases, which are used in computing the precision, are also shown in the table. The number of the true positive and false negative cases can be seen from Fig. 6. A large value of the true positive rate indicates that there are few positive algorithms that are falsely recognized as negative. As can be seen from the table, the value of the true positive rate of non-recursive sorting algorithms are larger than those of recursive sorting algorithms. This implies, as discussed earlier, that the Quicksort algorithms and Mergesort algorithms are more often misclassified than the other non-recursive algorithms. A quick glance at the values of the true positive rate and false negative rate in Table 5 reveals that the larger value of the true positive rate means the smaller value of the false negative rate. Consequently, the value of the false negative rate for the non-recursive sorting algorithms are smaller than those of the recursive sorting algorithms. The value of the precision for the non-recursive sorting algorithms is 100%, indicating that there is no single algorithm that is falsely recognized as Insertion, Bubble or Selection sort algorithm. However, as discussed earlier, one algorithm was falsely recognized as a Quicksort and one as a Mergesort.

By manually verifying those false negative cases that have occurred because of the different numbers of their numerical characteristics, and by assigning the right type to them in the database, the number of the true positive cases will increase and the number of the false negative cases will decrease dramatically. This will extend the knowledge base of the Analyzer so that in the future, it will recognize the same false negative cases correctly. This will result in a substantial improvement in the performance of the Analyzer.

### TABLE 5. The value of the true positive rate, false negative rate and precision, as well as the number of the false positive cases for each sorting algorithm. See Fig. 6 for the number of the true positive and false negative cases.

<table>
<thead>
<tr>
<th>Class</th>
<th>True positive rate (%)</th>
<th>False negative rate (%)</th>
<th>False positive</th>
<th>Precision (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>91.4</td>
<td>8.6</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Bubble</td>
<td>83.3</td>
<td>16.7</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Selection</td>
<td>82.8</td>
<td>17.2</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Quicksort</td>
<td>65.2</td>
<td>34.8</td>
<td>1</td>
<td>93.7</td>
</tr>
<tr>
<td>Mergesort</td>
<td>72.7</td>
<td>27.3</td>
<td>1</td>
<td>94.1</td>
</tr>
</tbody>
</table>
The reason for this is that the characteristics of algorithms process. Out that some other characteristics are needed in the recognition extending the Analyzer to cover other algorithms, it may turn are adequate to recognize other algorithms as well. When algorithms distinctly. However, it remains an interesting the computed characteristics can describe the five sorting other fields of algorithms. The results of recognizing the labeling them as Unknown. In future work, we need to cover recognizing the five above-mentioned sorting algorithms, or other sorting algorithms. However, this would not have been the case if the other algorithms had included a most-wanted holder role, these algorithms are easily and reliably separated from the other sorting algorithms. However, this would not have been the case if the other algorithms had included a most-wanted holder role as well. One way to deal with this problem is to use a multilevel recognition model, where algorithms are first divided in a higher level based on other characteristics, and the roles are applied to a lower level to refine and enhance the recognition process.

The presence of the roles in the recognition process also poses the next challenge. As discussed in [37, 38], roles of variables are cognitive concepts. This implies that people with different backgrounds may assign different roles to the same variable, or the same variable may be assigned more than one role, or even the same person may assign different roles to the variables that appear in the same situation. In contrast, an automatic role analyzer that assigns roles to variables deals with roles as technical concepts, not cognitive. This means that the roles produced by an automatic analyzer may well differ from those assigned by a human. In other words, it is not easy to develop an automatic role analyzer that can detect the roles with a very high degree of accuracy (i.e. the roles that are in accordance with humans’ perception of roles). The exact challenge that can arise from this issue in AR is that if a role assigned by a human to a variable of an algorithm of the learning data in the manual analysis phase of the algorithms is different from that produced by an automatic analyzer, the algorithm may be recognized falsely. However, progress in this field has been made, and the role analyzer we are currently using [34] already gives satisfactory results, though we are looking for getting access to even better ones.

The decision tree presented in Section 4 is manually constructed. It is a simple decision tree that has been built to illustrate a simplified example of how our method can be applied to recognizing sorting algorithms. The decision tree needs to be built automatically using an accurate algorithm. A good decision tree classifier algorithm will build a more accurate, concise and near-optimal decision tree. The problem of finding the optimal and smallest decision tree is an NP-complete problem (see [39, 40]). However, the decision tree of Section 4 can be optimized by, for example, performing the tail-recursive test (on the left subtree of the root) in a node near the root, rather than near the leaf. By eliminating the nodes that
perform ‘Temporary role?’ and ‘In-place?’ tests, this will result in a more accurate tree, e.g. due to the fact that those Quicksort algorithms that either do not have a temporary role, or the role analyzer has failed to recognize their temporary role, can be recognized more accurately (Section 5). Moreover, as we cover other fields of algorithms, the number of the characteristics and the nodes of the tree will increase to the extent that make it almost impossible to build a proper decision tree manually. We have already automated the process of building the decision tree for recognizing sorting algorithms using the C4.5 decision tree classifier algorithm [40] and reported the results in [41].

As part of future work, SV techniques can be utilized by the Analyzer to highlight the recognized algorithm from source code to emphasize the variables that have been used in the recognition process along with their roles, and to illustrate the other related aspects. All these features can greatly facilitate the understanding process.

The current system assumes that the algorithms work correctly. Recognizing incorrect algorithms is out of the scope of the method, but dynamic analysis methods, such as those applied in automatic assessment tools, could be used for that.

As we pointed out, our future work will include the extension of the Analyzer to cover other fields of algorithms. By the promising results presented in this work, we are encouraged and motivated to do this.

**FUNDING**

This work was partially supported by the Academy of Finland under grant number 111396.

**REFERENCES**


on Computer Science Education, Kansas City, Missouri, USA, February 14–15, pp. 118–129. ACM, New York, NY, USA.


