Advance Access publication on 11 March 2011
doi:10.1093/comjnl/bxr025

Using Decision Tree Classifiers in Source Code Analysis to Recognize Algorithms: An Experiment with Sorting Algorithms

AHMAD TAHERKHANI*

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

We discuss algorithm recognition (AR) and present a method for recognizing algorithms automatically from Java source code. The method consists of two phases. In the first phase, the recognizable algorithms are converted into the vectors of characteristics, which are computed based on static analysis of program code, including various statistics of language constructs and analysis of *Roles of Variables* in the target program. In the second phase, the algorithms are classified based on these vectors using the C4.5 decision tree classifier. We demonstrate the performance of the method by applying it to sorting algorithms. Using leave-one-out cross-validation technique, we have conducted an experimental evaluation of the classification performance showing that the average classification accuracy is 98.1% (the data set consisted of five different types of sorting algorithms). The results show the applicability and usefulness of roles of variables in AR, and illustrate that the C4.5 algorithm is a suitable decision tree classifier for our purpose. The limitations of the method are also discussed.

Keywords: algorithm recognition; decision tree classifier; C4.5 algorithm; roles of variables

Received 18 August 2010; revised 20 January 2011
Handling editor: Iain Stewart

1. INTRODUCTION

Algorithm recognition (AR) can be defined as understanding algorithms through their identification and classification from source code. By recognizing pieces of source code as particular algorithms and viewing these pieces as a whole, we can acquire partial understanding of the meaning of the code. The aim is to abstract the purpose of the code and recognize, for example, those parts that have potential for improvement. The goal of our research is to develop methods for automatic AR.

AR method can be used in automatic assessment tools to provide feedback for student about the algorithm they have used and to help teachers with their workload in grading student submissions in large programming courses. Specifically, the contribution of AR in this field would be to confirm that students have used the required algorithm. The existing automatic assessment tools, such as Boss [1], CourseMarker [2] and WebCAT [3], are capable of carrying out various useful functionalities and analyzing many different aspects of the target program, including program correctness, programming style, program structure, use of specific language construct and even run time efficiency (for an overview of the field, see [4]). However, none of them is able to analyze how the problem has been solved in terms of the used algorithms. It is difficult to automatically check a programming assignment such as 'Write a program that sorts an array using Quicksort that switches to Insertion sort when the sorted area is less than 10 items'. The final output of the program is a sorted array and gives no clue as to what algorithm has been used in reality. A simple approach 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 problem can be solved using AR, which provides methods to automatically recognize algorithms from source code. Despite

the extensive work already done, there still seems to be a lack of an adequate and efficient technique that is able to solve this problem. This is the main application and purpose of our research.

Another application of AR is in computer programming contests.[1] In these contests, the contestants are required to implement various algorithms. AR can be used to automatically check the implemented algorithms and give feedback to the contestants. AR would also be a valuable tool to help the jury in their evaluation task.

Moreover, AR can also be potentially used in *source code optimization*, that is, tuning existing algorithms or replacing them with more efficient ones. This is a key problem in developing compilers for parallel processing machines: how to identify algorithms that can be parallelized and how to replace them with new parallel algorithms that compute the same results. Metzger and Wen present an overview of this topic in [5].

In addition, AR techniques can also be utilized in source-to-source translations, specifically, in the well-known method *program translation via abstraction and reimplementation* [6], which was introduced to address the weaknesses of the previous method, *source-to-source translation by transliteration and refinement*. In this approach, first an abstract understanding of the original source code is achieved, and then the code is reimplemented in the target language. A slightly similar problem is locating undesirable code segments from the source code. Especially legacy systems may include code segments that do not serve their original purpose anymore and thus should be removed, or may use inefficient algorithms that should be replaced with more efficient ones. AR can potentially be used to automatically detect these kinds of code segments/algorithms, which are commonly referred to as *anti-patterns* (see, e.g. [7]).

1.1. The contributions of this paper

Our long-term plan is to tackle the AR problem by combining several approaches, as we believe that this will produce the best result. The most interesting approaches that we will use are as follows (from these approaches the first is presented in this work and the two others are left for future work).

(i) *Machine learning techniques*: we apply machine learning techniques to recognize algorithms. We explain this dimension in this paper. We discuss how decision tree classifiers can be used in the recognition process.

(ii) *Identifying algorithm-specific code*: the second approach involves identifying algorithm-specific code from non-relevant application data processing code. We will define each supported algorithm (algorithms which the tool will be able to recognize) as schemas/plans, and identify the relationships between the schemas in each algorithm; that is, how the schemas of each algorithm are connected to each other to constitute the algorithm. We will discuss the methods we aim to use in Section 2 in connection with the related work.

(iii) *Dynamic analysis*: the third perspective includes executing the identified algorithm-specific code and analyzing the dynamically generated trace in order to verify its correctness and obtain other useful information.

In this paper, we present a method for automatic AR. The method is based on the static analysis of program code, including various statistics of language constructs and roles of variables. The method consists of two phases. The first phase includes converting algorithms into characteristic vectors. In the second phase, these vectors are used to build a decision tree classifier for recognizing algorithms using the C4.5 algorithm. We have discussed the first phase of the method in more detail in [8–10]. The method presented in these earlier papers, however, uses a manually constructed decision tree for classifying algorithms. In [10], an experiment on sorting algorithms was conducted to test the performance of the method and to evaluate the accuracy of the manually constructed decision tree. In that study, 70 implementations of sorting algorithms of five types (Selection sort, Insertion sort, Bubble sort, Quicksort and Mergesort) were collected as the learning data; the implementation codes were run through an Analyzer, which computed the characteristics used in the recognition process and stored the resulted characteristic vectors in the database. On the basis of the algorithms of the learning data represented as characteristic vectors, a decision tree was manually constructed to guide the recognition process. After this, a total of 217 algorithms, including the same five types of aforementioned sorting algorithms (but different samples), other types of sorting algorithms as well as algorithms from other fields, were collected as the testing data in a separate data collection process. The algorithms of the testing data were then converted into characteristic vectors by the Analyzer and tested using the manual decision tree.

In this paper, the decision tree is constructed automatically using the C4.5 decision tree classifier algorithm. This is the main contribution of this paper: applying machine learning methods in source code analysis to automatize the construction of a decision tree that can be used in recognizing algorithms. In the experiment presented in this paper, a total of 209 samples of the implementations of the aforementioned five types of sorting algorithms were given to the C4.5 algorithm, which builds a decision tree that is used for recognizing algorithms. The average classification accuracy was also evaluated using leave-one-out cross-validation technique. The results presented in Section 6 show that using the instances of the data set, the

[1] See, for example, International Olympiad in Informatics (URL: http://ioinformatics.org/index.shtml) and ACM International Collegiate Programming Contest (URL: http://cm.baylor.edu/welcome.icpc).

leave-one-out cross-validation accuracy was 98.1%. Moreover, the decision tree that the C4.5 algorithm builds on the training data are much more optimal, simpler and understandable than the manually constructed decision tree presented in [8–10].

To limit the scope of this work, we applied the method to the same type sorting algorithms as presented in [10]. Focussing on the same type algorithms and data set allows us to contrast this work with the previous one and see how using machine-learning techniques improves the quality of the decision tree. However, since the method presented in this paper and the method discussed in [10] use different recognition mechanism (in [10], a filtering system is used to reject those algorithms that have less or more numerical characteristics than the minimum or maximum value of the numerical characteristics of the implementations of the algorithms of the learning data), data set (in [10], in addition to the five aforementioned sorting algorithms, the data set includes other algorithms as well) and evaluation method (in [10], *holdout method* [11] is used to divide the data set into the learning and testing data), direct comparison between the results of these two studies is not reasonable. As we will discuss later in the paper, we removed the non-sorting algorithms (which are called the 'Other' algorithms in [10]) from the testing data used in the experiment conducted in [10] and re-evaluated the performance of the decision tree presented therein. This provides a better basis to contrast the results presented in this paper with those obtained in [10].

Note that in this paper, we do not focus on dealing with algorithms that do not belong to the target set of the five aforementioned types of sorting algorithms. This issue was considered in the experiment presented in our previous study [10]. In that study, we used the numerical characteristics (see Section 4) to filter out the implementations of these non-sorting algorithms. The results of that study showed that the number of the algorithms that are not a member of the target set but are falsely recognized as such is very small. These cases occurred in less that one percent of the testing data, and thus are not a big problem. Therefore, the experiment presented in this study continues the previous one by focussing on the data set consisting of the implementations of the same types of the sorting algorithms as were used as a part of the data of that experiment, and by evaluating the performance of the proposed method on distinguishing the sorting algorithms from each other. The proposed method recognizes algorithms by applying machine-learning methods and by constructing an automatic decision tree using the C4.5 algorithm.

In future work, when we extend our method to cover other fields of algorithms, constructing manual decision trees would become more and more clumsy, difficult, inaccurate and perhaps impossible for considerably larger sets of target algorithms. Hence, the other contribution of this work is that it demonstrates that machine-learning methods, and specifically the C4.5 algorithm, can be used to handle construction of decision trees for AR purposes.

The second phase of the method, mainly the process of the automatic generation of the decision tree, is briefly outlined in a short paper earlier [12]. The present paper elaborates on the process of the construction of the decision tree, explains the recognition process based on the tree and presents a detailed discussion on the evaluation process. The necessary background information is also presented in this paper, and the application of the method to sorting algorithms as well as the applicability of the C4.5 algorithm as a machine-learning method in building an accurate decision tree classifier for recognizing algorithms is discussed in detail. In addition, the experiment conducted in the paper along with the results of evaluating the decision tree using leave-one-out cross-validation technique is discussed in detail and various metrics indicating the average classification accuracy are computed and explained. This paper also discusses the differences between the decision tree used in the experiment presented in this paper, and the manual decision tree used in the experiment conducted in [10].

It should be noted that our method is developed mainly with the educational application in mind. Therefore, as described in Section 6, the data set used in the process is collected from textbooks, student submissions, etc., so that it represents educational materials. Extending the method to cover other fields of algorithms and further developing it to deal with real-life systems remains for future work.

We start by discussing the AR problem in Section 2, where we also present some related work briefly. Section 3 discusses decision tree classifiers in general, and the C4.5 algorithm in particular, and gives a brief overview on the important related issues. Section 4 includes the description of the method, and in Section 5, we explain an experiment where the method has been applied in recognizing sorting algorithms. The results of the evaluation of the classification accuracy are presented in Section 6. Finally, Section 7 presents some discussion and conclusions, and gives some directions for future work. The limitations of the work are also discussed in this section. Note that as the first phase of the method as well as the process of data collection and preparation are presented in [10] in more detail, we discuss these issues very briefly in this paper.

2. AR AND RELATED WORK

In AR, the task is to recognize and classify algorithms. Recognizing algorithms covers identifying different types of algorithms that carry out the same task (e.g. sorting), as well as algorithms that perform different tasks.

There exist different algorithms that perform the same computational task, such as sorting an array or finding the minimum spanning tree of a graph. For example, the sorting problem can be solved by using Bubble sort, but also by Quicksort, Mergesort or Insertion sort, among many others. However, the problem of recognizing the applied algorithm has several complications. First, while essentially being the

same algorithm, Quicksort, as an example, can be implemented in several considerably different ways. Each implementation, however, matches the same basic idea (partition of an array of values followed by the recursive execution of the algorithm for both partitions), but they differ in lower level details (such as partitioning, pivot item selection method and so forth). Moreover, each of these variants can be coded in several different ways, for instance, using different loops, initializations, conditional expressions and so on.

In addition to the aforementioned variations that make AR a difficult and challenging task, there are also other issues that contribute to its complexity: 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 inlined within the code.

There are also computational complexity issues related to AR. From this perspective, AR can be considered as 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, as described in [13], is proved undecidable by Bar-Hillel *et al.* [14]. This problem is undecidable because there exists no algorithm that can show in a finite amount of time, whether two given input set of syntactic rules are equivalent; that is, whether they define the same language. On the other hand, the problem of AR can be regarded to be a problem of deciding whether two given algorithms are equivalent, that is, whether they perform the same task or solve the same problem. In order to be able to decide whether two algorithms solve the same problem, the functionality of those algorithms must be understood first. This means that being able to tell whether two algorithms solve the same problem can be regarded equal to being able to tell what problem those two algorithms solve. Thus, AR and syntactical equivalence problem can be regarded to belong to the same category, and this implies that AR problem can also be considered as an undecidable problem. We approach the problem by converting it into the problem of extracting the characteristics of algorithms and examining algorithms as characteristic vectors (see Section 4). Furthermore, we limit the scope of our work to include a particular group of algorithms. In addition, we are not looking for a perfect matching, but aim at developing a method that provides statistically reasonable matching results.

2.1. Related work

We present a brief overview of previous work related to AR. See [9] for a more comprehensive survey. This discussion does not cover the works on decision tree classifiers or other subfields of machine learning.

AR problem can be viewed from different perspectives and in connection with different approaches. In the following, we give a brief overview of some of these approaches. The method that we discuss in this paper is based on machine-learning techniques and thus differs from the approaches presented in this Section.

Clone detection research is close to our research, *identifying implementations of some predefined set of algorithms for human inspection* that would support understanding the purpose of the code. However, some clone-detection techniques may find some identifiers, such as variable names or comments, beneficial when trying to find similarities between code fragments, while these identifiers are not highly valuable in AR problem. Our method, as an example, does not make use of comments at all.

The following approaches in clone detection techniques can be discerned (see, e.g. [15] and [16] for more information): textual approach (text-based comparison between code fragments), lexical approach (the sourced code is transformed into a sequence of tokens and these are compared), metrics-based approaches (the comparison is based on the metrics collected from source code), tree-based approaches (clones are found by comparing the subtrees of the abstract syntax tree of a program) and program dependency graphs (the program is represented as program dependency graphs and isomorphic subgraphs are reported as clones).

As we discussed in Section 1, one direction of our future work pertains to identifying algorithm-specific code from the non-relevant application data processing code. We will define the supported algorithms (which exist in the knowledge-base of the system) as schemas and the relations between schemas. By matching the target algorithm against these schemas and by using the corresponding relations, we will identify the algorithm-specific code from the application-specific code. The recognized algorithm-specific code can then be displayed to the user to help him/her locate the algorithm within the source code. We will use *Knowledge-based program understanding techniques* in our future work to deal with this issue. In knowledge-based program understanding techniques, programs are understood by comparing the target program with the plans stored in the knowledge base of the system. Since the functionality of the plans in the knowledge base is known, the functionality of the target program can be discovered, if there is a match (see, e.g. [17]).

Program similarity evaluation techniques, i.e. plagiarism detection techniques are used to find the degree of similarity between programs. On the basis of how programs are analyzed, these techniques can be divided into two categories: *attribute-counting* techniques (where the similarity between programs is evaluated based on some characteristics; see, for example, [18, 19]) and *structure-based* techniques (structure of programs are analyzed to find their similarities; see, e.g. [20, 21]). Since the focus of the program similarity evaluation techniques is on the style and structure of a program rather than recognizing algorithms, these techniques are not highly relevant to AR as such. However, as we will discuss in Section 4 when presenting

our method, we use software metrics that are widely used in these techniques.

3. DECISION TREE CLASSIFIERS AND C4.5 ALGORITHM

In this section, we briefly discuss decision tree classifiers and the C4.5 algorithm. For more information about decision tree classifiers in general and the C4.5 algorithm in particular, see, for example, [22] and [23], respectively.

Decision tree classifiers use *classification* to divide different instances of a set into appropriate classes. Classification belongs to supervised learning, where first a set of known instances, called *training set*, is introduced to a system. The system classifies each instance of the set, associates each class with the attributes of each instance and learns to what class each instance belongs. On the basis of what the trained system has learned in the *learning phase*, it is able to classify instances of a previously unseen set.

There are several issues related to decision trees, such as how to deal with missing attributes, how to measure the quality of decision trees, etc. In the following, we briefly discuss two most important issues and explain how they have been dealt with in the C4.5 algorithm. The C4.5 algorithm is a widely used and the most well-known algorithm for building decision tree classifiers and has a good combination of error rate and speed [24]. Using the techniques presented in the following, the C4.5 algorithm provides an accurate, readable and comprehensible model about the structure of the data and the relationship between the attributes and this structure. Therefore, we chose the C4.5 algorithm to build the decision tree in our research.

3.1. Finding the best attribute

It is important to select attributes that can discriminate between different classes of data in the best possible way. The attribute that best divides the training data will be located in the root of the tree [24]. To find such an attribute, all the attributes are examined using some *goodness measure* [22, 24]. The earlier version of the C4.5 algorithm used *information gain* to evaluate the tests and find the best split. Information gain is based on entropy, a measure used in information theory. Entropy indicates the average information needed to identify instances of a set. Later, Quinlan (the developer of the C4.5 algorithm) noticed that information gain favors the tests that result in many outcomes. This causes problems when the outcomes of this kind of tests have no value with regard to the classification, for example, because of the small number of the instances associated with each outcome. Therefore, he introduced *information gain ratio* to fix this problem through adjusting the gain of these kinds of tests. The information gain ratio is the ratio of the information gain to the split information. It gives the information that is obtained by the ratio of the information relevant to the classification produced by the split, to the information that is provided by the split itself.

3.2. Finding the right size

When constructed, decision trees are often unnecessarily complex and need to be simplified. Complexity is associated with *overfitting*, which in turn causes *generalization* problem. It has been claimed that the quality of a decision tree depends more on the right size than the right split [22]. There can be many different sizes of a decision tree that are correct over the same training set, but the smaller size is preferred. A simpler decision tree is more likely to correctly recognize more instances of a testing set, because it can capture the structure of the problem and the relationship between the class of an instance and its attributes more effectively [25]. In addition to higher accuracy, smaller trees are more comprehensible as well [26]. Choosing the best discriminating attributes helps keep the size of a tree small. Because the problem of finding the smallest decision tree that is consistent with the training set is NP-complete [23, 24], selecting the right tests is very important in generating near-optimal trees. In his survey on automatic construction of decision trees, Murthy [22] lists several methods for obtaining right-sized trees. The most widely used method is *pruning*, where first the complete tree is built. Here, the complete tree means the tree where no splitting will improve the accuracy of the tree on the training data. In the next step, those subtrees with only little impact on the accuracy of the tree are removed. This is how this issue is handled in the C4.5 algorithm as well. Although this approach includes an extra computation for building the parts of the tree that will be eliminated later in the pruning phase, it is justified by the more accurate and reliable final result [23].

4. GENERAL METHOD

In this section, we discuss our method for recognizing algorithms. The method is based on static analysis of source code, including various statistics of the language. The method consists of two consecutive phases. In the first phase, we compute the distinguishing characteristics of implementations of algorithms and convert the implementations of algorithms into the characteristic vectors. In the second phase, we use these vectors to recognize and classify the algorithms using the decision tree classifier algorithm. The first phase is described in detail in our previous paper [10] as a part of the method used in that work and therefore, we only discuss it very briefly here and encourage the reader to see that paper for more information with regard to the first phase. The focus of this paper is on the second phase, which is discussed in the next section in connection with the application of the method on sorting algorithms.

The simplified model of Fig. 1 shows the two phases of the method described in this paper. Comparing this figure with Fig. 2, which illustrates the method described in our previous

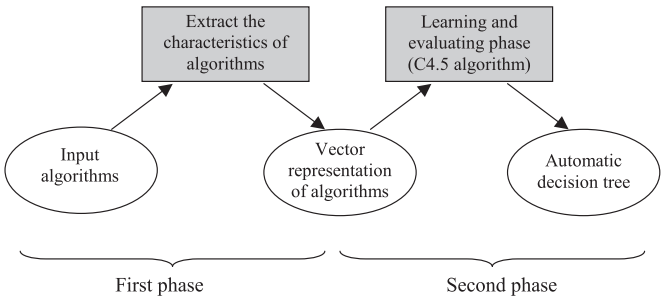

FIGURE 1. A simplified model of analyzing and recognizing algorithms using an automatic decision tree constructed by the C4.5 algorithm, and evaluating the accuracy of the classification using leave-one-out cross-validation technique. Rectangles illustrate the processes, and ellipses illustrate the inputs and outputs.

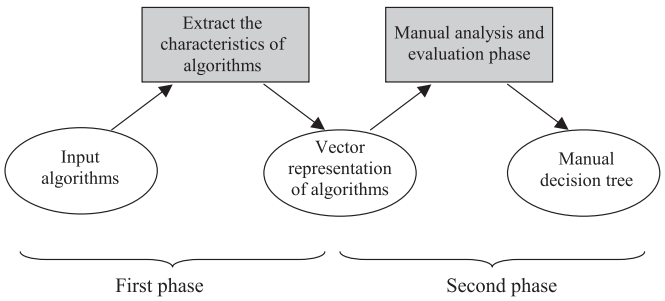

FIGURE 2. A simplified model of the method described in [10]. Algorithms are recognized using a manual decision tree, which is constructed based on the manual analysis of the implementations of the algorithms of the learning data. The performance of the method is evaluated using the implementations of the algorithms of the testing data.

paper [10], helps understand the differences between the two methods and the focus of this paper. As can be seen from the figures, the first phase (that is, converting implementations of algorithms to characteristic vectors) is the same in the two methods and the differences pertain to the second phase. In [10], a decision tree is constructed manually based on the implementations of the algorithms of the learning data, and its evaluation is performed using the implementations of the algorithms of the testing data. In this paper, the data set is given to the C4.5 algorithms as an input, a decision tree is constructed automatically, and the average classification accuracy is evaluated using leave-one-out cross-validation technique.

Table 1 shows the characteristics that we use in recognizing algorithms. We analyzed a number of algorithms and posited a hypothesis that the characteristics depicted in the table can be used to recognize different algorithms. We developed a prototype Analyzer that can automatically compute all the characteristics shown in Table 1 (as we will discuss in the following, for detecting roles of variables, we used the tool described in [27]). Application of these characteristics to sorting algorithms showed that they are sufficient to distinguish between these algorithms with high accuracy (see Sections 5 and 6). Recognizing other fields of algorithms may require more characteristics.

As shown in Table 1, we divided the characteristics of programs into the *numerical* and *descriptive characteristics*. The numerical characteristics are those that can be expressed as positive integers, whereas the descriptive characteristics describe some properties of the algorithm in question and are not expressed as numbers. In addition, other characteristics related to the numerical and descriptive characteristics are computed (see the last three characteristics in Table 1). The abbreviations in the table are used to refer to the corresponding characteristics in Tables 3 and 4, as well as in the other sections of the paper. Among the other numerical characteristics, we used the Halstead metrics [29], which are the characteristics abbreviated as N_1, N_2, n_1, n_2, N and n in Table 1. These metrics are originally used to evaluate the complexity of programs. They are also widely applied in the program similarity evaluation techniques discussed in Section 2.

The *Other characteristics* showed in Table 1 (i.e. *Block/loop information*, *Loop counter information* and *Dependency information*) can be used to identify different patterns that can facilitate the recognition process. We will discuss this in the next section. In the following, we give a brief overview on *Roles of Variables*.

All the computed characteristics are stored in the database. In order to use the computed descriptive characteristics, they are assigned an appropriate numerical value. For example, the characteristic 'Recursive' is represented by 1 in all the recursive algorithms and by 0 in all the non-recursive algorithms.

After converting an algorithm into a set of characteristics, it can be represented as an n-dimensional characteristic vector, where n is the number of the characteristics. Thus, the task of AR is converted into the task of building a decision tree classifier based on the characteristics of the algorithms and using this classifier to identify the algorithms based on the characteristic vectors that represent them.

4.1. Roles of variables

Roles of variables have a significant contribution in the process of recognizing and classifying algorithms. The idea behind roles of variables is that each variable used in a program plays a particular role that is related to the way it is used. Roles of variables are specific patterns how variables are used in source code and how their values are updated. Currently, there are 11 roles recognized that cover all variables used in novice-level programs. We present the definition of the two roles that are among the characteristics presented in Table 2 in Section 5. See [30] or visit the Roles of Variables Home Page (http://www.cs.joensuu.fi/~saja/var_roles/) for the definition of other roles as well as a more comprehensive information about them. For more detailed discussion on roles of variables within

TABLE 1. The numerical, descriptive and other related characteristics computed from algorithms.

Numerical characteristics	Description
NAS	Number of assignment statements in the algorithm
LoC	Lines of code
MCC	McCabe complexity (i.e., cyclomatic complexity) [28]
N_1	Total number of operators in the algorithm
N_2	Total number of operands in the algorithm
n_1	Number of unique operators in the algorithm
n_2	Number of unique operands in the algorithm
N	Program length ($N = N_1 + N_2$)
n	Program vocabulary ($n = n_1 + n_2$)
NoV	Number of variables in the algorithm
NoL	Number of loops. Supported loops are *for* loop, *while* loop and *do while* loop
NoNL	Number of nested loops in the algorithm
NoB	Number of blocks in the algorithm. A block refers to a sequence of statements wrapped in curly braces, for example, a method or a control structure (loops and conditionals)
Descriptive characteristics	**Description**
Recursive	Whether the algorithm uses recursion
Tail recursive	Whether the algorithm is tail recursive
Roles of variables	Roles of the variables of the algorithm
Auxiliary array	Does the algorithm use an auxiliary array (for the algorithms that use arrays in their implementation)
Other characteristics	**Description**
Block/loop information	Information about blocks and loops, including starting and ending lines, length and interconnection between them (how they are positioned in relation to each other)
Loop counter information	Information about how the value of loop counters are initialized and updated. This is used to determine incrementing/decrementing loops (the value of the loop counter increases/decreases after each iteration)
Dependency information	Direct and indirect dependencies between variables (variable i is directly dependent on variable j, if i gets its value directly from j. If there is a third variable k on which j is directly or indirectly dependent, i also becomes indirectly dependent on k. A variable can be both directly and indirectly dependent on another one)

the context of our method, see [10].

(i) *Most-wanted holder* (MWH): A variable that holds a most desirable value that is found so far from a succession of values.
(ii) *Temporary* (TEMP): A variable that holds a value for a short period of time.

4.2. The tool for detecting roles of variables

A tool developed by Bishop and Johnson [27] for automatic detection of roles of variables is integrated into the Analyzer. The tool detects roles using program analysis techniques, particularly program slicing and data flow analysis. First, all occurrences of each variable in the program are captured. The outcome of this analysis is the program slice for each variable. This is followed by data flow analysis for each program slice. On the basis of the initial analysis of the example programs, the tool associates each role with a set of assignments and usage conditions. To detect roles of variables, the tool compares the assignments and usage conditions of each variable of the target program with these predefined sets. If the user has provided a role for a variable, the tool checks whether all corresponding conditions for the provided role are met by the variable in question. If the conditions are not met, the tool prints the role it believes to be correct and justifies its decision by giving an appropriate message. If there is no role suggested by the user, the tool simply prints the role for the variable in question. More detailed description of the assignments, usage conditions and how the role detector works is beyond the scope of this paper. For more information, see [27].

Bishop and Johnson have developed their role detector for educational purposes. Therefore, the tool allows users to provide

TABLE 2. The descriptive characteristics specific to the sorting algorithms.

Characteristic	Description
MWH	Whether the algorithm includes a variable appearing in MWH role
TEMP	Whether the algorithm includes a variable appearing in temporary role
In-place	Whether the algorithm needs extra memory
OIID	(Outer Incrementing Inner Decrementing) Whether from the two nested loop used in the algorithm, the outer is an incrementing loop and the inner is a decrementing loop
IITO	(Inner Initialized To Outer) Whether from the two nested loop used in the algorithm, the inner loop counter is initialized to the value of the outer loop counter

a role for a variable. Although providing an actual role for a variable is optional, special tags along with the name of the variable and some string (whatever) as the role must be provided for each variable; otherwise the tool will ignore the variable. The tool can be further developed so that for each variable appearing in the program, the required tags, the name of the variable and the required string that must be given as the role are provided automatically. When preparing the data for our experiment, we also assigned a role to all the variables to be able to detect the possible differences between the roles generated by the tool and those that we believed to be correct. All the roles, however, were detected automatically.

Before using the role detector, we tuned it up a little bit in order to improve its performance. As an example, a temporary role typically appears in swap operations, which in turn is commonly used in sorting algorithms. In programs where a swap operation was performed in a separate method, the temporary role was sometimes falsely recognized as a fixed value by the role detector. To solve the problem, we automatically removed the method calls to swap operations in a preprocessing step, and inlined the corresponding swap method bodies in the target programs. As the result, temporary roles were detected much more accurately.

5. CLASSIFYING SORTING ALGORITHMS

We applied our method to five different types of commonly used sorting algorithms: Quicksort, Mergesort, Insertion sort, Selection sort and Bubble sort. In this section, we describe how the method has been applied to the sorting algorithms emphasizing the second phase of the method, which is the classification and recognition part. Hence, the focus of this section is on describing the process of constructing the decision tree and analyzing its structure. We present an empirical evaluation of the performance and accuracy of the classification over the data set in the next section.

Sorting algorithms are very suitable for demonstrating the feasibility of the method due to the fact that there are different types of sorting algorithms that carry out the same computational task. Some of these algorithms are very similar (for example, Insertion sort and Bubble sort algorithms), while the others are clearly different (e.g. Quicksort and Bubble sort algorithms). These features make the task of recognizing different types of these sorting algorithms challenging and provide a suitable basis to evaluate the performance of the method properly. Note that our method performs static analysis and deals with the source code lexically, syntactically and semantically. Hence, the 'similarities' and 'differences' between algorithms in the context of our method means the similarities or differences from the point of view of the analyzed characteristics, which the method uses, and not from other points of view.

In addition to the characteristics discussed in the previous section, we computed the descriptive characteristics of Table 2, which can be used in the process of recognizing sorting algorithms. These characteristics can be easily computed based on those shown in Table 1. *OIID* (Outer loop Incrementing Inner Decrementing) and *IITO* (Inner loop counter Initialized To Outer loop counter) are computed using the characteristics *NoNL*, *Loop counter information* and *Dependency information*. In addition, if an implementation of a sorting algorithm does not use an *Auxiliary array* (the characteristics presented in Table 1), we can conclude that the corresponding sorting algorithm is an *in-place* algorithm; that is, it does not need extra memory to carry out the sorting. Finally, by examining the roles of the variables in the target algorithm detected by the role analyzer, the existence of *MWH* (most-wanted holder role) and *TEMP* (temporary role) can easily be found out.

In the following, we describe how the sorting algorithms are converted into the vectors of characteristics and how these vectors are used in building a decision tree classifier.

5.1. Creating characteristic vectors

We collected a total of 209 sorting algorithms of the five aforementioned types as the learning data (the data set was part of the data used in our previous work [10]). We developed a prototype Analyzer that computes all the characteristics automatically. The Analyzer is implemented in Java and the current version is able to process source code written in Java. It parses the code, computes its numerical and descriptive characteristics and analyzes all the related characteristics shown in Tables 1 and 2. This information is stored in a database consisting of four tables: *Algorithm*, *Block*, *Variable* and *Dependency*. As this is the learning data, the correctness of the type of each algorithm is verified in the database. We also integrated a tool [27] into the Analyzer for automatic detection of roles of variables. This tool was described in Section 4.

TABLE 3. The minimum and maximum of the numerical characteristics of the five types of the sorting algorithms in the data set (see Table 1 for the explanation of the abbreviations).

Algorithm	NAS	LoC	MCC	N_1	N_2	n_1	n_2	N	n
Insertion sort	8/12	13/32	4/6	47/79	45/66	15/24	5/10	92/145	20/34
Selection sort	10/14	16/31	4/5	50/77	46/72	13/22	6/9	96/149	19/31
Bubble sort	8/13	15/30	4/6	44/80	44/69	13/25	5/10	88/149	18/35
Quicksort	6/19	26/57	4/14	79/171	69/131	17/31	5/11	148/302	22/42
Mergesort	11/27	27/56	6/10	104/174	83/143	22/29	6/12	187/317	28/41

TABLE 4. The percentages of the distribution of the descriptive characteristics of the five types of the sorting algorithms in the data set (see Table 1 and 2 for the explanation of the abbreviations.

Algorithm	Recursive	Tail_recursive	In-place	MWH	TEMP	OIID	IITO
Insertion sort	0.0	0.0	100.0	0.0	96.2	100.0	92.3
Selection sort	0.0	0.0	100.0	100.0	97.7	2.3	65.1
Bubble sort	0.0	0.0	100.0	0.0	92.7	17.1	0.0
Quicksort	100.0	100.0	100.0	0.0	66.7	N/A	N/A
Mergesort	100.0	0.0	50.0	0.0	17.6	N/A	N/A

The characteristics OIID and IITO are not applicable for the Quicksort and Mergesort algorithms.

The numerical and descriptive characteristics of each type of the analyzed sorting algorithms are shown in Tables 3 and 4, respectively. For the numerical characteristics shown in Table 3, the first number indicates the minimum and the second number the maximum value of the corresponding characteristic.

As discussed above, in order to use the descriptive characteristics in building the decision tree, we converted them into binary values, 1 indicating the presence of the corresponding characteristic and 0 indicating its absence. A quick glance to the descriptive characteristics in Table 4 reveals how distinguishing attributes these characteristics are, especially *Recursive*, *Tail_recursive* and *MWH*. The characteristic Recursive can perfectly discriminate all the instances of the Insertion sort, Selection sort and Bubble sort algorithms from the Quicksort and Mergesort algorithms in the data set.[2] Tail recursion, on the other hand, completely distinguishes the Quicksort algorithms from the Mergesort algorithms. Also the existence of MWH in a sorting algorithm is an excellent classifier, which differentiates between the Selection sort algorithms and the other sorting algorithms. As we will see, the decision tree that the C4.5 algorithm builds over the learning data set makes use of these exact characteristics. As illustrated in Table 4, the characteristics OIID and IITO (see Table 2) are not counted for the Quicksort and Mergesort algorithms. This is because either the implementations of these algorithms do not have two nested loops, or they are not good classifiers in these cases. However, as we will explain later, IITO is an important classifier for separating the Insertion sort algorithms from the Bubble sort algorithms. Note that since swap operations are commonly used in the implementations of the three non-recursive sorting algorithms, all the implementations of these algorithms include a temporary role. The percentages of the characteristic TEMP illustrated in Table 4 reflect the fact that the role detector we used failed to detect all the temporary roles correctly.

5.2. The C4.5 classification tree

Figure 3 shows the classification tree generated by the C4.5 algorithm over all the instances of the data set and from the previously discussed characteristics.[3] In the picture, the internal nodes are illustrated as ellipses and include the tests based on which the splits are performed. There are four internal nodes (including the root), and thus, the values of four characteristics are used for the tests. The tree includes five leaves, which correspond to the number of classes, that is, the types of the sorting algorithms. Each leaf is labeled with the associated type. The arcs in Fig. 3 are labeled with either 0 or 1 from each internal node to its children. These values indicate the outcome of the test performed in each corresponding internal node. For example, from the internal node labeled with Tail_recursive, the

[2]Although it is feasible to write, for example, a recursive Bubble sort or a non-recursive Quicksort, it did not occur in our data set. Moreover, it can be argued that whether, as an example, a non-recursive Quicksort is essentially the same algorithm as the commonly known recursive Quicksort.

[3]We used J48, which is an open source Java implementation of the C4.5 algorithm in the Weka data mining software, developed at the University of Waikato. URL: http://www.cs.waikato.ac.nz/~ml/weka/.

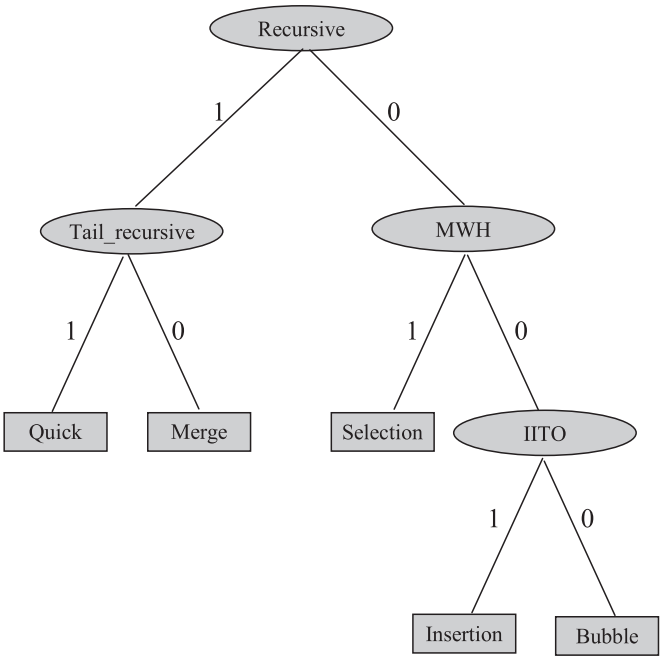

FIGURE 3. The classification tree constructed by the C4.5 algorithm for recognizing sorting algorithms over all the instances of the data set (See Table 1 for the explanation of the abbreviations).

arc that goes to the left is labeled with 1, and the arc that goes to the right is labeled with 0. The former arc indicates that those algorithms that are tail recursive belong to the leaf labeled with 'Quick', and the latter means that those algorithms that are not tail recursive belong to the leaf labeled with 'Merge'.

The characteristics that are used in the classification tree are *Recursive*, *Tail_recursive*, *MWH* (most-wanted holder role) and *IITO* (Inner loop counter Initialized To Outer loop counter). The characteristic that appears in the root node (i.e. Recursive) is the best classifier. It divides the sorting algorithms into two groups, recursive sorting algorithms (Quicksort and Mergesort) and non-recursive sorting algorithms (Insertion sort, Bubble sort and Selection sort). The next classifiers are the two children of the root, Tail_recursive and MWH, which are located at depth 2. Tail_recursive separates Quicksorts from Mergesorts. In the case of the three non-recursive algorithms, MWH separates the Selection sorts from the Insertion and Bubble sorts, since only the Selection sort algorithms include a MWH variable. The Insertion and Bubble sort algorithms do not include selection of a min (or max) element and therefore have no variable appearing in MWH role (see the definition of MWH variable in Section 4). Finally, IITO separates the Insertion sorts from the Bubble sorts.

Not all the analyzed characteristics are used in the decision tree. The C4.5 algorithm tries to minimize the depth of the tree. It classifies the instances using the minimum number of characteristics that give the best result. As the task here is to classify the five types of the sorting algorithms, the decision tree is not so complicated and thus, many characteristics remain unused. If there were more algorithm fields to be classified, more characteristics would presumably be used in the process.

By analyzing the characteristics presented in Tables 3 and 4, it becomes evident why the aforementioned four characteristics are used as the classifiers in the tree. The numerical characteristics of the algorithms, although clearly different, are not the most useful characteristics in discriminating between the algorithms, when compared with the four used characteristics. Similarly, from the descriptive characteristics, In-place, TEMP and OIID have less discriminating value than the used ones.

It should be noted that the purpose of constructing the decision tree of Fig. 3 has been to investigate how an automatic decision tree distinguishes between the five aforementioned sorting algorithms. Thus, the decision tree of Fig. 3 only demonstrates how the five sorting algorithms can be recognized and distinguished from each other. The tree is not capable of classifying other fields of algorithms. When we extend our method to cover other algorithms in future work, the tree will have a mechanism to distinguish, as an example, between tail-recursive algorithms (as opposed to the decision tree of Fig. 3, which classifies any given tail-recursive program as a Quicksort). In a more complex decision tree built for classifying a more comprehensive field of algorithms, the numerical characteristics, for example, can be used to distinguish between different types of algorithms that share other common characteristics with each other (e.g. characteristic such as being tail-recursive).

Figure 4 shows the manual decision tree constructed in our previous work [10] based on the analysis of the implementations of the algorithms of the learning data. As can be seen from the figure, we used the numerical characteristics in that work to filter out those algorithms that have less or more numerical characteristics than the minimum or maximum value of the numerical characteristics of the implementations of the five sorting algorithms of the learning data. This mechanism detects those algorithms that are not a member of the target set of the five sorting algorithms, and prevents them from being further processed (these algorithms are called 'Other' algorithms in [10]). As can be seen from the Figs 3 and 4, the C4.5 algorithm builds a more optimal and simpler tree, with the maximum depth of three. The maximum depth of the manually constructed decision tree is four. In addition to the four characteristics used in the automatic decision tree, the manual decision tree uses also characteristics the TEMP (temporary role), In-place (using extra memory or not) and OIID (Outer loop Incrementing Inner Decrementing).

Decision trees can be converted into sets of rules, where each path in the tree from the root to a leaf is presented as a set of rules [23]. Although it is easy to understand decision trees, rules are even easier to understand [26]. Since the paths of our decision tree are based on the characteristics, the rules also consist of these characteristics. The definitions of the sorting

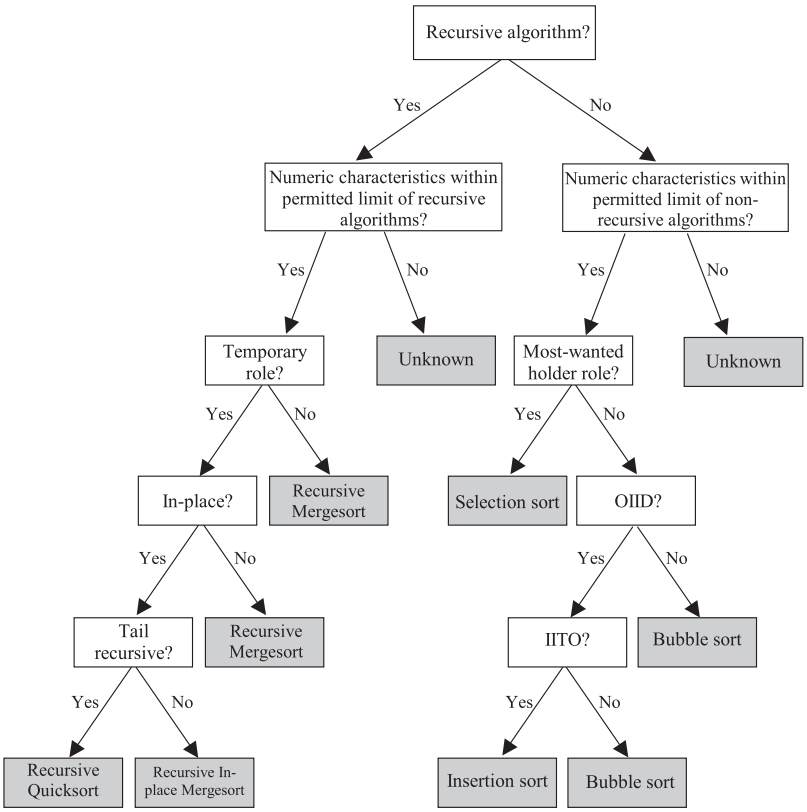

FIGURE 4. The manual decision tree constructed in [10] for recognizing sorting algorithms. The tree is constructed based on the analysis of the implementations of the algorithms of the learning data.

algorithms as rules are as follows:

(i) Quicksort: Recursive ∧ Tail_recursive.
(ii) Mergesort: Recursive ∧ ¬Tail_recursive.
(iii) Selection sort: ¬Recursive ∧ MWH.
(iv) Insertion sort: ¬Recursive ∧ ¬MWH ∧ IITO.
(v) Bubble sort: ¬Recursive ∧ ¬MWH ∧ ¬IITO.

As can be seen, the shorter the path from the root to a leaf is, the shorter the definition of the corresponding algorithm as rules gets. The rules presented above are indeed very simple, like the decision tree. Covering more algorithms in the classification task will result in a more complicated set of rules. It takes only a quick glance to the definitions above to define the algorithms as the rules. For example, an instance of the Selection sort algorithm is not recursive and includes a MWH role.

6. EVALUATION RESULTS

The classification performance of a classifier should be evaluated by empirical tests. In this section, we discuss the experiment conducted to carry out this evaluation and present the results.

6.1. Data set

Since the goal of this study is to use the C4.5 algorithm to automatically construct a decision tree to recognize sorting algorithms, the data set used as the learning data in this study consists of only sorting algorithms. We used this data set to estimate the accuracy of the classification using leave-one-out cross-validation technique. The implementations of the sorting algorithms of the data set used in this experiment are the same as those used in the experiment presented in [10]. The approach presented in [10] is more general. It identifies non-sorting algorithms and filters them out so that they are not processed any further. Hence, the data set used in the experiment presented in [10] includes other types of algorithms in addition to the sorting algorithms used in the experiment presented in this paper. We discuss the process of data collection and preparation very briefly in the following. For more information, see [10].

We collected a total of 209 sorting algorithms of the five aforementioned types without any preference for particular sources. These algorithms were gathered from various textbooks on data structures and algorithms, as well as from course materials available on the Web. Some of the Insertion sort and Quicksort algorithms were from authentic student work. The distribution of the algorithms in the data set, both in numbers

and in percent, is as follows: Insertion sort 52 (25%), Bubble sort 41 (20%), Selection sort 43 (20%), Quicksort 39 (19%) and Mergesort 34 (16%).

We verified the type and correctness of the algorithms both by investigating the source code and by running them and examining the outputs. If an algorithm of the data set included extra code (i.e. printing statements, code related to interface, etc.), we removed it, since the method is not able to process application code at its current state. However, the implementation of the algorithm itself was left untouched.

6.2. Evaluation of the classification accuracy

There are various techniques to evaluate the classification accuracy and their applicability depends mainly on the data set. *Cross-validation* is a widely used technique, where the data set is divided into N subsets, which include both the training and test set. N different decision trees are constructed. Every time a decision tree is constructed, $N-1$ subsets are used as the training set and one subset is used as the test set to estimate the accuracy of the constructed tree. Thus, all of the subsets are used as the test set, and each of them exactly once. Cross-validation has two important advantages. First, it makes the best use of the available data (compared with *holdout* technique where the data set is divided into mutually independent training and test set). Second, since all the subsets take part in both the training and test set, the instances of the data set are distributed uniformly to the training and test sets. This eliminates the risk of getting a poor accuracy value for a decision tree just because of the unseen instances of the test set happen to vary largely from the instances of the training set [23].

Leave-one-out cross-validation is a special case of cross-validation, where N is equal to the number of the instances in the data set. This makes even better use of the data set when the available data are not large. Clearly, the disadvantage of leave-one-out cross-validation is that it can be computationally expensive for large data sets. As described, our data set is relatively small and therefore, we decided to use this technique to estimate the classification performance. With the 209 algorithms of our data set, we get the training-test subsets created 209 times, that is, leave-one-out cross-validation generates 209 decision trees, each time using 208 algorithms as the training data and one algorithm as the testing data. Thus, the result of the evaluation indicates the average accuracy of the generated trees.

We discuss the results in terms of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) metrics. TP indicates the case where the decision tree correctly recognizes an algorithm that belongs to one of the members of the target set, that is, one of the five types of the sorting algorithms. TN correspondingly indicates rejecting an algorithm, which does not belong to any member of the target set. FP denotes that an algorithm not belonging to the members of the target set is incorrectly recognized as one of the five types of the sorting algorithms, FN correspondingly an algorithm belonging to a member of the target set, which is recognized as another member of the target set. TP and TN cases indicate algorithms that are successfully recognized by the tree, whereas FP and FN cases mean failure in recognizing the algorithms correctly.

Moreover, based on the values discussed above, the following metrics can also be calculated [11] for further discussing the results. True positive rate (TPR), also called *sensitivity*, is the proportion of the positive case algorithms that are correctly captured by the decision tree: $\mathrm{TPR} = \mathrm{TP}/(\mathrm{TP} + \mathrm{FN})$. True negative rate (TNR), also known as *specificity*, is the proportion of the negative case algorithms that are correctly predicted: $\mathrm{TNR} = \mathrm{TN}/(\mathrm{TN} + \mathrm{FP})$. False positive rate (FPR) is the proportion of the negative case algorithms that are incorrectly labeled as a positive case: $\mathrm{FPR} = \mathrm{FP}/(\mathrm{TN} + \mathrm{FP})$. Finally, false negative rate (FNR) is the proportion of the positive case algorithms that are incorrectly labeled as a negative case: $\mathrm{FNR} = \mathrm{FN}/(\mathrm{TP} + \mathrm{FN})$.

Performed on the data set described earlier, the average classification accuracy of the leave-one-out cross-validation is 98.1%. In other words, from the 209 algorithms of the data set, a total of 205 algorithms are classified correctly, and four of them (1.9%) are misclassified. Table 5 shows the overall results. The column 'Total' shows the total number of the algorithms for each algorithm class, the column 'Correct' shows the number of correctly classified algorithms of each class, the column 'False' depicts the number of misclassified algorithms of each class and the column 'Correct%' illustrates the correctly classified algorithms of each class in percent.

Table 6 shows the results in more detail, where the values of the aforementioned metrics are presented for each type of the sorting algorithms. In the table, the first and the second columns show the class and the total number of each sorting algorithm, respectively. The rest of the columns show the values of the metrics in the same order as they are defined earlier.

As Tables 5 and 6 show, all the misclassified algorithms are the implementations of the Insertion sort algorithms. We use the *confusion matrix* to discuss the misclassified algorithms in more detail. The confusion matrix is an $N \times N$ matrix, where each instance I_{ij} indicates the instance that belongs to class I_i, but is classified as class I_j [11]. The instances located on the diagonal

TABLE 5. The number of correctly and falsely classified sorting algorithms.

Algorithm class	Total	Correct	False	Correct%
Quicksort	39	39	0	100.0
Mergesort	34	34	0	100.0
Selection sort	43	43	0	100.0
Insertion sort	52	48	4	92.3
Bubble sort	41	41	0	100.0
Total	209	205	4	98.1

TABLE 6. The value of different metrics indicating the classification accuracy.

Class	Total	TP	TN	FP	FN	TPR	TNR	FPR	FNR
Bubble sort	41	41	0	0	0	1.0	0.0	0.0	0.0
Insertion sort	52	48	0	0	4	0.923	0.0	0.0	0.077
Selection sort	43	43	0	0	0	1.0	0.0	0.0	0.0
Mergesort	34	34	0	0	0	1.0	0.0	0.0	0.0
Quicksort	39	39	0	0	0	1.0	0.0	0.0	0.0

TABLE 7. The confusion matrix of evaluating the classification accuracy by leave-one-out cross-validation technique.

Class	Bubble	Insertion	Selection	Merge	Quick
Bubble sort	41	0	0	0	0
Insertion sort	4	48	0	0	0
Selection sort	0	0	43	0	0
Mergesort	0	0	0	34	0
Quicksort	0	0	0	0	39

are classified correctly. We have five classes of algorithms, thus 5×5 confusion matrix, as shown in Table 7. As can be seen from the table, all the four misclassified Insertion sorts are labeled as a Bubble sort. All the instances of the four other classes are classified correctly.

Note that in a production environment, where the previously unseen algorithms are classified using the decision tree depicted in Fig. 3, the classified algorithms are stored in a database along with their characteristics. Therefore, a user can always manually verify the results and make sure that the algorithms are classified correctly. For example, when the method is applied in programming education, a teacher can examine the students' submissions that are classified as negatives, and evaluate their correctness. If a submission is correct and is implemented according to the instructions, but is misclassified due to any reason, its type can be manually corrected in the database. Since the teacher does not need to examine the true positive cases, using the Analyzer as an assessment tool reduces the teacher's workload considerably, as he or she does not need to evaluate all the submissions manually.

We consider the average classification accuracy of 98.1% obtained by the leave-one-out cross-validation to be a satisfactory result. The results also demonstrate that the computed characteristics are sufficient to discriminate between the five types of the sorting algorithms. Although the decision tree classifier of Fig. 3 does not use many characteristics at this stage, we believe that they will be helpful when other fields of algorithms are taken into the process. However, it is well possible that we will have to use some other characteristics in order to recognize other fields of algorithms, and/or drop some of the current characteristics as redundant.

6.3. The automatic and manual decision trees

Contrasting the automatic decision tree constructed by the C4.5 algorithm with the manual decision tree presented in our previous work [8–10] helps us understand the applicability and suitability of machine learning methods and particularly the C4.5 algorithm in producing decision trees for classifying and recognizing algorithms. This discussion also provides a clearer perspective to see our research in general and to position the work presented in this paper within our overall research.

The results of the experiment conducted in our previous work [10] showed that the manual decision tree used in that work was able to recognize 86% of the implementations of the algorithms of the testing data correctly (the true positive and true negative cases), and it misclassified 14% of the implementations of the algorithms of the testing data (the false positive and false negative cases).

The accuracy of the manual decision tree cannot be directly compared with the average classification accuracy obtained by the leave-one-out cross-validation technique presented in this work. This is because the classification mechanism, the evaluation method and the data set used in these two studies are different. For constructing and evaluating the manual decision tree presented in [10], we used the *holdout method* [11], where the data set is divided into the learning and testing data. On the basis of the manual analysis of the characteristics of the algorithms of the learning data, we built a manual decision tree and used the algorithms of the testing data to evaluate the accuracy of the tree. The learning data (70 instances) consisted of only the implementations of the five types of sorting algorithms discussed earlier in this paper, but the testing data (217 instances), in addition to the five types of the sorting algorithms, included other fields of algorithms as well (these algorithms are referred to as 'Other' in [10]). Furthermore, the manual decision tree uses the numerical characteristics presented in Table 3 to filter out the Other algorithms of the testing data. However, as discussed before, since the aim of this study has been to investigate the applicability of machine-learning methods in building a decision tree classifier for classifying sorting algorithms and see how the C4.5 algorithm distinguishes between the five types of the sorting algorithms, the data set in this study includes only algorithms belonging to the five types of the sorting algorithms, and therefore, there is no mechanism for filtering out non-sorting algorithms. Moreover, in this study, the average classification accuracy is evaluated using the leave-one-out cross-validation as opposed to the holdout method used in [10].

To provide a better basis that allows us to contrast the automatic decision tree with the manual decision tree, we removed the Other algorithms from the testing data of the manual decision tree experiment presented in [10] and re-evaluated the accuracy of that tree using only the five aforementioned types of the sorting algorithms, as is the case in the experiment presented in this work. This change does not make the two decision trees and their evaluations directly comparable yet, but

it gives us more reasonable conditions to compare and contrast them. Note that there are 78 Other algorithms used in the testing data in [10], and removing them will result in a testing data consisting of 139 sorting algorithms. Using 139 sorting algorithms of the aforementioned five types as the testing data, the accuracy of the manual decision tree is 79%; that is, 110 of the algorithms of the testing data are recognized correctly (the true positive cases), and the rest 29 algorithms (i.e. 21%) are recognized incorrectly (the false negative cases). These data provide a more reasonable foundation to cautiously compare the accuracy of the manual decision tree (79%), with the average classification accuracy achieved in this study (i.e. 98.1%).

Although constructing a manual decision tree for recognizing the five types of the sorting algorithms is a feasible task, the C4.5 classifier constructs a much more optimal, logical and understandable decision tree. Automating the process of constructing a decision tree for recognizing a more extensive set of algorithms will be inevitable, because constructing the decision tree manually will become more inaccurate and unreliable as the number of different types of algorithms increases.

Note that, as can be seen in Table 6, since all the algorithms of the data set used in the experiment are a member of the target set (i.e. they belong to a class of the five aforementioned types of the sorting algorithms), TN cases do not occur in the experience. In other words, there is no algorithm within the data set that is not a member of the target set, and is correctly recognized as a negative case (i.e. is correctly rejected). Furthermore, FP cases do not occur in the experiment for the same reason. That is, there is no algorithm within the data set that is not a member of the target set, but is falsely recognized as a sorting algorithm that belongs to one of the five types of the sorting algorithms in the target set.

The results of the experiment conducted to evaluate the accuracy of the manual decision tree used in [10] (which has a mechanism for dealing with the algorithms that are not a member of the target set, as shown in Fig. 4) showed that among the 217 implementations codes of the testing data, there were only two implementations of the Other algorithms that were falsely recognized as a sorting algorithm of the target set. This makes less than one percent of the testing data, which means that FP cases should not be a big problem.

7. DISCUSSION, CONCLUSION AND FUTURE WORK

We have discussed a method for recognizing algorithms and illustrated its performance by applying it to sorting algorithms. The method converts algorithms into characteristic vectors and classifies them based on these vectors. We have also discussed the decision tree built by the C4.5 algorithm and evaluated the average classification accuracy using leave-one-out evaluation technique.

Promising results of applying the method to sorting algorithms (the average classification accuracy of 98.1%) suggest the feasibility of the method. It also shows that in this case, the characteristics extracted from the algorithms can distinctly and adequately describe them. Applying the C4.5 algorithm showed that from the characteristics depicted in Tables 3 and 4, only four characteristics are sufficient for classifying the five types of the sorting algorithms with a high level of average accuracy. As we extend our method to cover other algorithms, however, considerably more of these characteristics would presumably be used in the classification. The characteristics used in this study are based on analyzing the instances of the data set and positing a hypothesis that they can differentiate between algorithms. This hypothesis is backed by the results of this study in the case of the five aforementioned sorting algorithms. However, there well might be other distinguishing characteristics that are not included in Tables 3 and 4. We should investigate other possible characteristics and show their usefulness by empirical tests. It should also be noted that for some characteristics, the strategy based on which they are computed might have an effect on the accuracy of the method. As an example, the Halstead metrics could be counted using different strategies resulting in different values.

In this work, we have focused on the sorting algorithms. Limiting the scope of the work and focussing on the limited types of algorithms make it possible to discuss the issues in an appropriate level of detail. Focussing on the sorting algorithms also allows us to contrast this work with our previous work [10] and understand the improvements that using machine-learning methods and particularly the C4.5 algorithm brings to our method. In our previous work [10], we manually constructed a decision tree to classify the same types of sorting algorithms as in this work. In addition to the four characteristics used in this work, the characteristics TEMP (temporary role), In-place (using extra memory) and OIID (Outer loop Incrementing Inner Decrementing) were also used in that tree. The C4.5 algorithm successfully classifies the algorithms using only four characteristics. This results in a tree with the maximum depth of three, whereas the maximum depth of the manually constructed decision tree was four. In other words, the decision tree constructed by the C4.5 algorithm is simpler and smaller in size. As was discussed in Section 3 in connection with the discussion on decision tree classifiers and the C4.5 algorithm, simpler decision trees are more accurate [25]. Moreover, the fact that the same four characteristics were used in both trees confirms that these characteristics are good classifiers. Indeed, as we cover more fields of algorithms in our future works, constructing proper and accurate manual decision trees will become very difficult and almost impossible. Therefore, machine-learning methods should be used to automatize the process. Although drawing conclusions about the applicability of machine-learning methods on other fields of algorithms should be based on the appropriate empirical experiments, the results of this work strongly suggest that

machine-learning methods are highly suitable for recognizing algorithms. Particularly, based on the results presented in this work, the C4.5 classifier has proven itself as a suitable algorithm for automating the process of building decision trees in AR.

Roles of variables turned out to be distinctive factors. The MWH provides a very good discriminator to distinguish the implementations of the Selection sort algorithms from the implementations of the Insertion and Bubble sort algorithms (see the decision tree depicted in Fig. 3). Initial manual inspection revealed that there is no other single characteristic within the computed characteristics that can do this. It is interesting to investigate in future work, how roles preserve their value as good discriminators when other fields of algorithms are taken into the process and the number of target algorithms is very much larger or the target program is much larger and includes irrelevant application data processing code. Larger number of algorithms means the existence of more variables that appear in both different and the same roles. Will roles of variables be as useful as they are in recognizing sorting algorithms, if there appear many similar roles in the target algorithms? Clearly, existence of, for example, MWH role in some algorithms other than Selection sort algorithms will result in its importance to be reduced as a good distinguishing attribute. This potential problem can be dealt with using a multi-level recognition process, where the target algorithms are first separated by other factors and roles are applied in a lower level to recognize the algorithms more precisely.

Another issue with using roles of variables as distinctive factors in AR is the accuracy of an automatic role detector. This is one of the main concerns of our research: how accurately an automatic role detector can detect a wide set of roles that presumably appear in other fields of algorithms? The role detector we used in this study performed reasonably and we are looking for even better tools to use in our future studies.

We recognize that our method is statistical by nature and therefore, we cannot claim that it could ever achieve 100% accuracy in the recognition problem. For example, even though did not happen in our data, the target algorithm may be a non-recursive Quicksort or a recursive Insertion sort and thus not classified correctly by the decision tree. Furthermore, at its current state, the method is sensitive to 'noises' around the target algorithms, especially with regard to the numerical characteristics. Simple additions of application-specific code within the algorithm-specific code results in difference in the numerical characteristics. This will cause problem in future work when the numerical characteristics will presumably be used to distinguish between larger numbers of different algorithms. We need to address this problem by applying techniques from knowledge-based program understanding. Identifying algorithmic schemas from the source code and using them in the recognition process will enhance the method considerably. These schemas help us to differentiate between algorithm-specific code and application data processing code in larger systems and select the algorithm-specific code from the source code for further processing. Algorithmic schemas also help us to recognize different algorithms having similar values in the characteristics. As we discussed at the beginning of this paper, this is the approach we will adopt in our future work.

The choice of the training data is crucial in our research, like in all machine-learning techniques. When we take other fields of algorithms into the process in future work, the algorithms of the training data should be chosen carefully so that the classification would be based on the common representations of the target algorithms. Bad and non-representative training data is a threat to the validity of our research.

It should be noted that the current system assumes that the algorithms work correctly. Recognizing incorrect algorithms is out of the scope of this paper. Dynamic analysis methods, such that are applied in automatic assessment tools, could be used for that. As discussed before, we will extend our method in future work to cover this perspective as well.

As already stated, we need to further develop the method to cover other fields of algorithms. The results presented in this work provide encouragement and motivation to do so.

ACKNOWLEDGEMENTS

The author thanks Lauri Malmi, Ari Korhonen and Jorma Sajaniemi for their valuable comments.

FUNDING

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

REFERENCES

[1] Joy, M., Griffiths, N. and Boyatt, R. (2005) The BOSS online submission and assessment system. *ACM J. Educ. Resour. Computing*, **5**, 1–28.

[2] Higgins, C., Symeonidis, P. and Tsintsifas, A. (2002) The marking system for CourseMaster. *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education*, Aarhus, Denmark, June 24–26, pp. 46–50. ACM, New York, NY, USA.

[3] Edwards, S.H. (2003) Rethinking computer science education from a test-first perspective. *Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, USA, October 26–30, pp. 148–155. ACM, New York, NY, USA.

[4] Ala-Mutka, K. (2005) A survey of automated assessment approaches for programming assignments. *Comput. Sci. Educ.*, **15**, 83–102.

[5] Metzger, R. and Wen, Z. (2000) *Automatic Algorithm Recognition and Replacement*. The MIT Press, USA.

[6] Waters, R.C. (1988) Program translation via abstraction and reimplementation. *IEEE Trans. Softw. Eng.*, **14**, 1207–1228.

[7] Brown, W.J., Malveau, R.C., McCormick H.W. III, and Mowbray, T.J. (1998) *AntiPatterns Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons.

[8] Taherkhani, A., Malmi, L. and Korhonen, A. (2009) Algorithm recognition by static analysis and its application in students' submissions assessment. *Proceedings of the 8th Koli Calling International Conference on Computing Education Research*, Koli, Finland, November 13-16 , 2008, pp. 88–91. ACM New York, NY, USA.

[9] Taherkhani, A., Malmi, L. and Korhonen, A. (To appear) Using roles of variables in algorithm recognition. *Proceedings of the 9th Koli Calling International Conference on Computing Education Research*, Koli, Finland, October 29 – November 1, 2009. Department of Information Technology, Uppsala University. http://arachne.it.uu.se/research/publications/reports/2010-027/2010-027.pdf#page=5

[10] Taherkhani, A., Korhonen, A. and Malmi, L. (2010) Recognizing algorithms using language constructs, software metrics and roles of variables: an experiment with sorting algorithms. *Comput. J.*, **54**, 1049–1066.

[11] Tan, P.-N., Steinbach, M. and Kumar, V. (2006) *Introduction to Data Mining*. Addison-Wesley, USA.

[12] Taherkhani, A. (2010) Recognizing sorting algorithms with the C4.5 decision tree classifier. *Proceedings of the 18th International Conference on Program Comprehension*, Braga, Portugal, June 30 – July 2, pp. 72–75. IEEE.

[13] Harel, D. and Feldman, Y. (2004) *Algorithmics The Spirit of Computing*. Addison-Wesley.

[14] Bar-Hillel, Y., Perles, M. and Shamir, E. (1961) On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, **14**, 143–172.

[15] Roy, C.K., Cordy, J.R. and Koschke, R. (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, **74**, 470–495.

[16] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E. (2007) Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, **33**, 577–591.

[17] Harandi, M. and Ning, J. (1990) Knowledge-based program analysis. *Softw. IEEE*, **7**, 74–81.

[18] Grier, S. (1981) A tool that detects plagiarism in pascal pograms. *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education*, Seattle, WA, USA, October 1–2, 2005, pp. 15–20. ACM, New York, USA.

[19] Jones, E.L. (2001) Metrics based plagiarism monitoring. *Proceedings of the Sixth Annual CCSC Northeastern Conference on The Journal of Computing in Small Colleges*, Middlebury, VT, USA, April 20–21, 2001, pp. 253–261. ACM, New York, NY, USA.

[20] Mozgovoy, M. (2007) Enhancing Computer-Aided Plagiarism Detection. Doctoral dissertation, University of Joensuu.

[21] Wise, M.J. (1996) Yap3: improved detection of similarities in computer program and other texts. *Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, PA, USA, February 15–17, 1996, pp. 130–134. ACM, New York, NY, USA.

[22] Murthy, S.K. (1998) Automatic construction of decision trees from data: a multi-disciplinary survey. *Data Min. Knowl. Discov.*, **2**, 345–389.

[23] Quinlan, J.R. (1993) *C4.5: Programs for Machine Learning*. Morgan Kaufmann, USA.

[24] Kotsiantis, S.B. (2007) Supervised machine learning: a review of classification techniques. *Inform. Int. J. Comput. Inf.*, **31**, 249–268.

[25] Quinlan, J.R. (1986) Induction of decision trees. *Mach. Learn.*, **1**, 81–106.

[26] Kohavi, R. and Quinlan, R. (1999) Decision tree discovery. *Handbook of Data Mining and Knowledge Discovery*, pp. 267–276. University Press.

[27] Bishop, C. and Johnson, C. G. (2005) Assessing roles of variables by program analysis. *Proceedings of the 5th Baltic Sea Conference on Computing Education Research*, Koli, Finland, November 17–20, pp. 131–136. University of Joensuu, Finland.

[28] McCabe, T.J. (1976) A complexity measure. *IEEE Trans. Softw. Eng.*, **SE-2**, 308–320.

[29] Halstead, M. (1977) *Elements of Software Science*. Elsevier Science Inc., New York, NY, USA.

[30] Sajaniemi, J. (2002) An empirical analysis of roles of variables in novice-level procedural programs. *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Arlington, VA, USA, September 3–6, pp. 37–39. IEEE Computer Society Washington, DC, USA.