Beacon- and Schema-Based Method for Recognizing Algorithms from Students' Source Code

AHMAD TAHERKHANI and LAURI MALMI
Aalto University
Department of Computer Science and Engineering
P.O.Box 15400, FI-00076 AALTO, Finland
{ahmad, lma}@cs.hut.fi

In this paper, we present a method for recognizing algorithms from students' programming submissions coded in Java. The method is based on the concept of *programming schemas* and *beacons*. Schemas are high-level programming knowledge with detailed knowledge abstracted out, and beacons are statements that imply specific structures in a program. The method automatically searches for schemas from the given program and compares the extracted schemas with those from the knowledge base to recognize the algorithm-specific code for further processing. In the next step, several characteristics and beacons specific to the given algorithm are computed from the code. These characteristics and beacons are then used as the learning data and given to the C4.5 algorithm, which builds a classification tree that can be used to classify previously unseen implementations of algorithms.

The method and its performance is demonstrated in the case of basic sorting algorithms and their variations implemented both in various learning resources, that is, textbooks and websites, ($N = 209$ programs) and in genuine student submissions in a first year data structures and algorithms course ($N = 159$). The empirical study conducted for evaluating the performance of the classification by leave-one-out cross-validation technique shows that the estimated true positive rate is 97.0%. The results demonstrate the feasibility of the idea of recognizing algorithms based on algorithm-specific programming schemas and beacons. The method can be used as white-box analysis to verify that students have implemented the required algorithm and to give feedback to students on their problematic implementations. We discuss the applications of the method from both teachers' and students' point of view.

Keywords: Algorithm recognition, detecting programming schemas, roles of variables, white-box analysis, decision tree classifiers

1. INTRODUCTION

Data structures and algorithms are central topics in programming education. Students need to gain understanding of a variety of them, including basic algorithms and structures for storing, sorting and searching data, and they should learn to look for more advanced algorithms in various fields of computing. The learning goals generally include understanding the main principles and implementation ways of the target algorithms, as well as their theoretical properties concerning time and space efficiency. In this paper, we focus on programming exercises where students need to implement algorithms, and how can we provide automatic feedback on their work. Our special focus is in sorting algorithms, through which we demonstrate the feasibility of the methodology, though the methodology is general and can be applied to a wide area of algorithms.

Basic programming education requires that students solve a large number of practical exercises, which causes a heavy load for teachers in terms of feedback and assessment. Therefore, many automatic assessment tools have been developed and applied for programming education. Well-known tools include Boss [Joy et al. 2005], Course-Marker [Higgins et al. 2002] and WebCat [Edwards 2003], among many others. Two recent surveys map the field well. In 2005, Ala-Mutka [Ala-Mutka 2005] listed topics which could be analyzed by the tools. These included 1) functionality, that is, whether the program produces the requested correct output, 2) program run time efficiency, 3) testing skills, that is, how well students have tested their programs, 4) special features like memory management, 5) coding style, 6) suspicious code, 7) software metrics, 8) program design, 9) use of specific language features, and 10) plagiarism. More recently, Ihantola et al. [Ihantola et al. 2010] analyzed more closely automatic assessment tools published since 2005, that is, after Ala-Mutka's survey. They recognized more recent trends in the field, such as integration of automatic assessment tools and learning management systems, more sophisticated ways to evaluate program functionality, integration of manual and automatic assessment, and development of various sandboxing approaches to secure against malicious student code. Their paper has an extensive list of recent work in the field. However, in these two comprehensive surveys, no tools have been reported, which could automatically analyze and give feedback on what kind of algorithms students use in their programs and how they have implemented them, although implementing various basic algorithms and data structures is a standard practice in programming education. In our research, we have therefore focused on the question: *How could we automatically recognize given algorithms or their variants from student source code?* Functionality testing methods commonly used in automatic assessment systems focus on testing the correctness of program output. They provide very limited support for evaluating the algorithmic solutions. As an example, if students are requested to implement some sorting algorithm, the program output should obviously be a sorted data set, but sorting itself may have been implemented using many different algorithms and their variants. It is very cumbersome to use current methods to assess that students have actually used a requested algorithm, or give any feedback on how well the implementation has been carried out. Moreover, the feedback provided by functionality testing methods does not tell anything to students about the quality of their solutions. Our aim is to develop methods that give instant feedback to students so that they can immediately understand the problems with their solutions and try to fix it. Students can ask the system for help whenever they encounter a problem. This is highly valuable, as it helps students to gain deep understanding and learn better.

We have previously developed two methods for algorithm recognition: a beacon-based classification method and a schema detection method. The former analyzes the given (target) code in terms of various software metrics, roles of variables and various other characteristics and beacons to build a characteristic vector which describes the algorithmic structure of the code. These are given to the C4.5 algorithm which builds a classification tree that can be used to recognize unseen algorithms. The latter method analyzes the code to match the target code against a given set of predefined schemas that correspond to algorithms to be searched from the code. Both methods have been separately successfully applied to analyze algorithm codes drawn from textbooks and websites, as well as authentic student submissions (see [Taherkhani 2011b] and [Taherkhani 2011a], respectively). The current paper has two main contributions. First, we introduce a combined method, which first extracts the schemas corresponding to the target algorithm and then computes the software metrics and the other characteristics and beacons from the schemas to be used in the classification. The combined method separates application code, that is, code not related to target algorithms, from algorithmic code, and thus further processes only the algorithmic code. This makes the recognition more reliable, as non-relevant code does not affect the computed characteristics and beacons. We have implemented the combined method in a prototype instrument called *Aari system* (an Automatic Algorithm Recognition Instrument). Second, this paper covers student-implemented algorithm implementations to build a classification tree. We include problematic solutions implemented by students. This enables teachers to identify these problematic solutions and give feedback on them. It also allows students to understand their misconceptions and get help from the system when encountering problem in their implementations.

The paper is structured as follows. Section 2 presents related work. In Section 3 we explain the overall principles of the analysis method. Section 4 demonstrates how the method has been applied to identifying basic sorting algorithms. Section 5 presents the data set, which is followed by introducing the classification tree built for recognizing sorting algorithms in Section 6. Section 7 presents the experimental data analysis carried out for evaluating the method. Section 8 elaborates on the application of the proposed method in education. Section 9 includes a discussion on the results and Section 10 concludes the paper.

2. RELATED WORK

Several techniques and tools have been introduced to facilitate program comprehension. These techniques work mainly using a knowledge base, where a set of stereotypical programming schemas, which are called *plans*, *idioms*, etc., are stored. In order to recognize the given program, it is matched against these plans. If a match between the plans of the given program and those from the knowledge base is found, these plans of the given program can be considered as identified, since the corresponding plans of the knowledge base are known. This process can be carried out using *top-down*, *bottom-up* or *hybrid* approaches.

Top-down approaches need the specification of the given program to select the suitable schemas from the knowledge base. Although this can be a more effective approach than bottom-up, its major disadvantage is that specifications of target programs are not necessarily available (see, as an example, [Johnson and Soloway 1984]). Bottom-up approaches start from small plans and continue recognizing bigger plans using the information about already recognized smaller plans that are parts of bigger ones. This approach may become ineffective as the size of knowledge base grows (see, for example, [Harandi and Ning 1990]). Hybrid techniques make use of both approaches (see, e.g., [Quilici 1994]).

The knowledge extracted by program comprehension tools can be used for different purposes including restructuring programs, teaching novices, generating documentation from code and finding the location of those parts of code that can be reused [Quilici 1994].

Program comprehension has also been studied from a theoretical perspective and various models for program comprehension and programming schemas have been introduced (see, e.g., [Brooks 1983], [Pennington 1987] and [Soloway and Ehrlich 1984]). The results of these program comprehension models can be used to develop more efficient tools that assist programmers and maintainers in program comprehension related tasks. In her survey on theories, methods and tools in program comprehension, Storey notices that the characteristics that influence cognitive strategies used by programmers, influence the requirements or supporting tools as well. As an example, top-down and bottom-up strategies introduced in program comprehension models are reflected in a supporting tool so that the tool should support "browsing from high-level abstractions or concepts to lower level details, taking advantage of beacons in the code; bottom-up comprehension requires following control-flow and data-flow links" [Storey 2006].

Concept location is a subfield of program comprehension that deals with finding code fragments that implement particular features and domain concepts in a program. Concept location approaches can be divided into dynamic and static techniques [Dit et al. 2013]. Dynamic techniques analyze execution traces and map them to code (see, e.g., [Eisenberg and Volder 2005; Edwards et al. 2006]). Static techniques are based on program dependencies and textual information within code (e.g., [Marcus et al. 2004; Gay et al. 2009]). Concept location approaches are mostly interactive and iterative [Gay et al. 2009], that is, a user initiates the process by formulating a domain concept as a query. Queries are mapped to code using information retrieval methods, such as Latent Semantic Indexing (LSI) (see, e.g., [Marcus et al. 2004]). The results of the query are then evaluated by the user. If the results are not satisfactory, the user reformulates the query and restarts the process. Both dynamic and static techniques have their limitations. Hybrid approaches are developed to address these limitations by using static information to filter the execution traces (see, e.g., [Poshyvanyk and Marcus 2007b; Liu et al. 2007]). As an example, in [Poshyvanyk and Marcus 2007b], the effectiveness and precision of the feature location method is improved by combining LSI information retrieval technique and scenario-based probabilistic ranking (a dynamic technique). Poshyvanyk et al. [Poshyvanyk and Marcus 2007a; Poshyvanyk et al. 2012] introduced a method that combines formal concept analysis and LSI. By producing more relevant search results, the method reduces the effort needed for locating concepts.

Clone detection techniques are used for locating clones in source code. Clone means duplication of some piece of a source code, which is either intentionally copied by a programmer from somewhere else in the same software to be reused directly or with some small modifications, or is created by him or her without awareness of the existence of the same code elsewhere in the software. Since clones make the maintenance task more difficult, clone detection is an essential task in software engineering. Several techniques for detecting clones have been introduced, including the following: textual approach (text-based comparison between code fragments, see, e.g., [Marcus and Maletic 2001; Ducasse et al. 1999]), lexical approach (the source code is transformed into a sequence of tokens and these are compared, see, for example, [Basit et al. 2007; Baker 1995]), metrics-based approaches (the comparison is based on the metrics collected from the source code, as, for example, in [Mayrand et al. 1996; Kontogiannis et al. 1996]), tree-based approaches (clones are found by comparing the subtrees of the abstract syntax tree of a program, see, e.g., [Baxter et al. 1998; Yang 1991]) and program dependency graphs (the program is represented as program dependency graphs and isomorphic subgraphs are reported as clones, as, e.g., in [Krinke 2001; Komondoor and Horwitz 2001]). Roy et al. [Roy et al. 2009] and Bellon et al. [Bellon et al. 2007] compare and evaluate a number of different clone detection techniques and tools in their surveys.

Algorithm recognition is close to clone detection, since the purpose of algorithm recognition is to look for similar patterns of algorithmic code in the source code, in the same way that clone detection techniques look for similar code fragments. Similarity in clones is either based on semantic or program text. Detecting semantic similarity is an undecidable problem in general and therefore most approaches and surveys focus on program text similarity [Bellon et al. 2007; Tiarks et al. 2011]. From this perspective, clones can be classified into four types [Tiarks et al. 2011]: 1) exact clone (type-1) is an exact copy of consecutive code fragments without modifications (except for whitespace and comments); 2) parameter-substituted clone (type-2) is a copy where only parameters (identifiers or literals) have been substituted; 3) structure-substituted clone (type-3) is a copy where program structures (complete subtrees in the syntax tree) have been substituted. For parameter-substituted clones, a leaf in the syntax tree can be replaced by another leaf, whereas for structure-substituted clones, larger subtrees can be substituted; and 4) modified clone (type-4) is a copy whose modifications go beyond structure substitutions by added and/or deleted code. Recognizing algorithms from source code can be considered as the type-4 and semantic clones, where according to the recent surveys [Bellon et al. 2007; Tiarks et al. 2011] the current techniques and tools perform poorly.

Machine learning methods are widely used in education to explore and understand the data gathered from educational setting. For example, in [Amershi and Conati 2009], a combination of supervised and unsupervised classification is used to build student models for exploratory learning environments and to identify how students behave while interacting with these environments. The researchers in [Miksatko and Mclaren 2008] use inexact graph matching, text analysis and machine learning classifiers to automatically detect pedagogically interesting patterns produced while students interact with tools for e-discussions. In [Drummond and Litman 2010], supervised machine learning, namely the C4.5 decision tree algorithm, is used to automatically detect student zoning out (i.e., "thinking about other things while [performing a learning task]", as describe in [Drummond and Litman 2010]). The researchers in [Dominguez et al. 2010] used data mining in an e-learning system to generate individualized feedback for students in order to help them find course notes relevant to their problems. Labeke et al. [Labeke et al. 2008] use string similarity metrics to compare and rank timelines of lifelong learners. The authors define timelines as a chronological record of lifelong learners' relevant life episodes and argue that by making it possible to define and share timelines, their system support lifelong learners to develop their common ideas together.

Metzger and Wen [Metzger and Wen 2000] introduce methods for algorithm recognition and replacement with a focus on development of compilers for parallel processing machines as an area of source code optimization. The aim is to improve existing algorithm implementations or replace them with more efficient ones that compute the same results.

3. THE OVERALL METHOD

Figure 1 illustrates the overall process of building a classification tree for algorithm recognition and evaluating the performance of the classification presented in this paper. The process consists of four steps, which are represented as rectangles with white background. Ellipses illustrate the inputs and outputs of the process. The first step is "Detect schemas and related beacons", where the input program is analyzed for recognizing programming schemas and extracting the implemented algorithm from the program. In order to detect schemas and distinguish between two or more algorithm implementations that have similar schemas, it is also necessary to compute the beacons related to these schemas in this step. We will demonstrate this in the case of sorting algorithms in Sections 4. If the implemented algorithm is extracted successfully, the output of the first step is "Detected algorithm", the pure algorithm implementation code with application data processing code left out. Otherwise the output is the same original input program. The second step, "Extract and store characteristics and beacons" computes the characteristics and beacons that are not computed yet. The output of this step is a vector representation of the input algorithm, which is stored in a database to be used in the following step. Because this is the learning data, type information of algorithm implementations is provided by a user to annotate each instance by its correct type. This is illustrated by the "Type information" input in Figure 1 that is inserted into the learning data using a dashed arrow. The third step is "Build a decision tree", where the type-annotated vector representations of the analyzed algorithms are used by the C4.5 algorithm to construct a classification tree. The final step evaluates the estimated performance of the

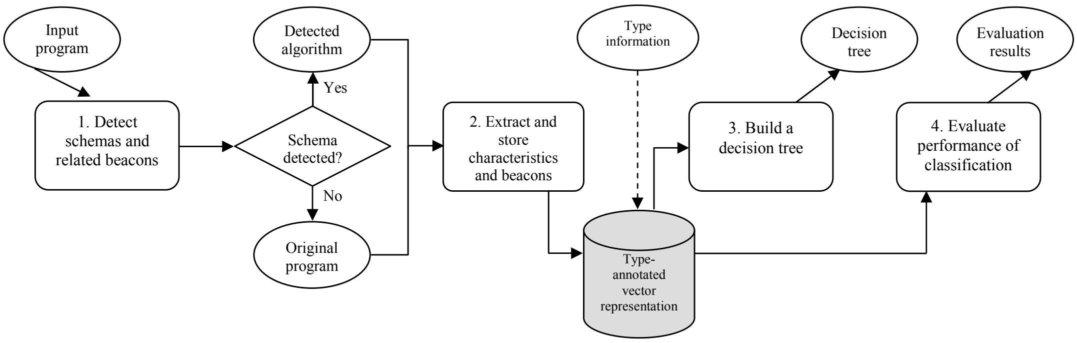
Input program
1. Detect schemas and related beacons
Detected algorithm
Yes
Schema detected?
No
Original program
2. Extract and store characteristics and beacons
Type information
Type-annotated vector representation
3. Build a decision tree
Decision tree
4. Evaluate performance of classification
Evaluation results

Figure 1. An overview of the process of building a decision tree and evaluating the estimated performance of the classification

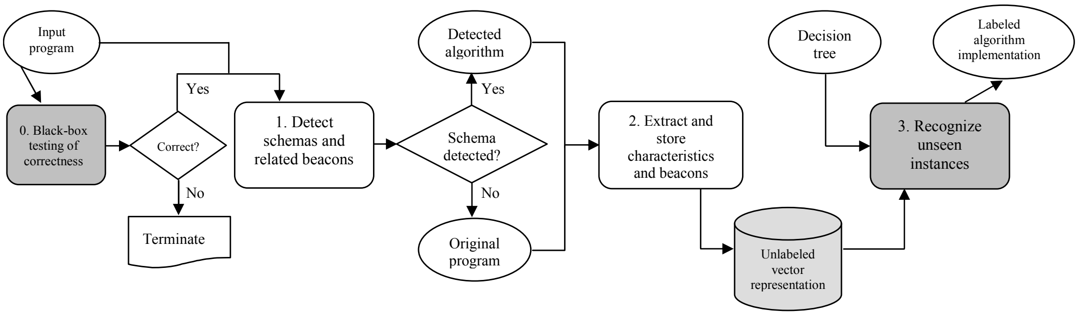
Input program
0. Black-box testing of correctness
Correct?
Yes
No
Terminate
1. Detect schemas and related beacons
Detected algorithm
Yes
Schema detected?
No
Original program
2. Extract and store characteristics and beacons
Unlabeled vector representation
Decision tree
3. Recognize unseen instances
Labeled algorithm implementation

Figure 2. An overview of the process of recognizing previously unseen algorithm implementations

classification using leave-one-out cross-validation technique and outputs the results of the evaluation.

Note that Steps 3 and 4 in Figure 1 are independent from each other, that is, the evaluation of the performance of the classification by cross-validation can be carried out before building a classification tree. We, however, discuss the steps in the presented order, as this clarifies the structure of the paper. Note also that Steps 1 and 2 of the figure are executed as many times as there are instances in the data set, whereas Steps 3 and 4 only once.

Figure 2 shows the process of recognizing previously unseen instances of a data set. The steps with white background are identical in this process and the process illustrated in Figure 1. The different steps are represented by gray rectangles. The process starts with testing the input program automatically by an automatic assessment system that gives feedback about the correctness of the program in terms of black-box testing. This preprocessing phase is performed as a part of input data validation. Only programs which pass all tests are processed further, that is, our method only aims at recognizing algorithms which work correctly. After this, the input program is processed for schema detection and characteristic and beacons extraction and storing in the same way discussed above (Steps 1 and 2). Since the resulted vector representations of input algorithm implementations are not known, there is no type information associated with them. Recognizing unknown algorithm implementations is performed in Step 3 “Recognize unseen instances”, which gets a readily constructed classification tree as an input.

Each instance of the given unseen data set is assigned a type by traversing the classification tree according to the characteristics and beacons of that instance. The output of this final step is the input algorithm implementation labeled by the identified type.

We have previously conducted an empirical study to classify the instances of an unseen data set using another classification tree, as reported in [Taherkhani et al. 2012a]. Conducting a similar empirical study for the classification tree described in the present study is out of the scope of this paper. Instead, as discussed above, we evaluate the estimated performance of the classification using leave-one-out cross-validation (Step 4 in Figure 1).

In this section, we describe the first and second steps of the process of Figure 1 in general. In the next section, we show how these steps work in practice by applying them to sorting algorithms. Sections 6 and 7 explain the third and fourth steps of the process for sorting algorithms, respectively.

It should be noted that we use the term characteristics to mean the common features that are used for all fields of algorithms (e.g., software metrics), whereas by beacons we mean the features that are specific to a particular field of algorithms. We compute the beacons and use them together with the characteristics to be able to differentiate between different types of algorithms in a particular algorithm field, as well as between the algorithms of that field and other fields. We will present the computed beacons for sorting algorithms in Section 4. The computed characteristics are discussed in this section.

3.1. Detecting schemas

We have developed a technique for automatic schema detection that is based on the studies about how programming schemas are created and used in different programming tasks, such as program comprehension. Programming schemas can be considered as formalized knowledge structures [Détienne 1990]. They are high-level programming knowledge that abstract detailed knowledge, and therefore, their possession allows experts to understand programs that perform the same tasks, but differ in low-level implementation details.

In their study on programming schemas, Soloway and Ehrlich concluded that experts perform significantly better in program comprehension tasks with programs that use stereotypical schemas (they use the term *plans* for schemas) and conform to rules of programming discourse. The reason is that experts have developed programming schemas and conventions and they use them in their tasks. On the other hand, for novices, whether the target programs use stereotypical schemas and conform to rules of programming discourse or not, has not much impact on performance, as they lack schemas [Soloway and Ehrlich 1984].

In the same way that experts use schemas as abstract structures to recognize programs performing the same task but implemented with different details, a tool can be developed that stores these high level schemas in its knowledge base and uses them to recognize its inputs. High-level schemas are first extracted from the input program ignoring implementation details. The extracted schemas are then matched against those schemas from the knowledge base. The knowledge base consists of the schemas of all the algorithms that the tool supports. If the high-level schemas of two or more algorithms are similar, algorithm-specific beacons can be used to distinguish between them. We will explain beacons in the next subsection.

Note that a single schema could sufficiently represent simple algorithms, like Bubble sort, in the knowledge base. However, for more complex algorithms, we might need to include several schemas or subschemas in the knowledge base in order to recognize various implementation instances of these algorithms. These algorithms can be implemented in several considerably different ways. As an example, while different implementations of Quicksort follow the same basic idea (partition of the given list followed by recursive execution of the algorithm for both partitions), they might differ, for example, in partitioning part, due to using different patterns of loops and swap operations. We will discuss this more in Section 4, where we present the schemas for sorting algorithms. For a more comprehensive description of the presented schema detection technique, see our previous paper [Taherkhani 2011a].

Note that in order to identify schemas and distinguish between the algorithms and algorithm variations that may have similar schemas, we need to identify the schema-related beacons as algorithm-specific features in the first step of Figure 1. We will explain this in the context of sorting algorithms and their variations in Section 4.

3.2. Creating characteristic and beacon vectors

We use the term *characteristics* for the shared features of all algorithms and the term *beacons* to denote algorithm-specific features that can be used to distinguish a particular algorithm from other algorithms.

We divided the characteristics into the following three groups: *numerical characteristics*, *truth value characteristics* and *structural characteristics*, as shown in Table I. The numerical characteristics are common software metrics that represent features of the code as positive integers. The first six characteristics in the table are Halstead's metrics [Halstead 1977]. Structural characteristics enable us to identify language constructs and different patterns as well as algorithm-specific beacons.

We have developed a prototype named Aari that can compute the characteristics and beacons for programs written in Java. The output of this step of computing the characteristics and beacons for a target algorithm implementation is a new representation of the algorithm implementation. That is, the algorithm implementation is now converted into

Table I. The numerical, truth value and structural characteristics

Numerical characteristics	**Description**
N_1	Total number of operators.
N_2	Total number of operands.
n_1	Number of unique operators.
n_2	Number of unique operands.
N	Program length ($N = N_1 + N_2$).
n	Program vocabulary ($n = n_1 + n_2$).
NAS	Number of assignment statements.
MCC	Cyclomatic complexity (i.e., McCabe complexity) [McCabe 1976].
LoC	Lines of code.
NoV	Number of variables.
NoL	Number of loops.
NoNL	Number of nested loops.
NoB	Number of blocks.
Truth value characteristics	
Recursive	Whether the algorithm uses recursion.
Tail recursive	Whether the algorithm is tail recursive.
Roles of variables	Roles of the variables in the program.
Auxiliary array	Does the algorithm use an auxiliary array (for the algorithms that use arrays in their implementation).
Structural characteristics	
Block/loop information	Information about blocks and loops, their starting and ending lines, length and interconnection between them (how they are positioned in relation to each other).
Loop counter information	Information about initializing and updating the value of loop counters. This allows us to determine, as an example, incrementing and decrementing loops.
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).

a characteristic- and beacon-vector, which is a technical definition of the implementation. For recognizing the target algorithm, we need to recognize this technical definition.

The importance of beacons in programming and program comprehension related tasks is emphasized by several program comprehension models. Soloway and Ehrlich [Soloway and Ehrlich 1984] call beacons as *critical lines* and define them as highly informative lines that can be considered as important representative of a plan. Existence of critical lines in a program strongly suggests presence of the corresponding plan in that program. Therefore, critical lines help experts recognize and verify plans. Brooks [Brooks 1983] defines beacons as statements that suggest existence of a specific structure in a program. For example, using swap operation particularly inside a pair of loops is an indication of sorting elements of an array.

One of the beacons that appeared very useful in our method is *roles of variables* which we explain next.

3.3. Roles of variables

Roles of variables are abstracts patterns, how variables are used in programming. These patterns recur in different programs, and can be recognized as a set of standard ways of using variables [Sajaniemi 2002]. For example, a variable that is used for storing a value in a program for a short period of time can be assigned a temporary role. As Sajaniemi and Navarro Prieto argue, roles of variables are a part of programming knowledge that have remained tacit [Sajaniemei and Prieto 2005]. 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 makes it a valuable tool that can be used in teaching programming to novices by showing the different ways how variables can be used in a program.

Studies carried out to investigate the impact of using roles of variables in education show that when utilized in elementary programming courses, roles provide a conceptual framework for students that helps them construct programs and comprehend them better [Kuittinen and Sajaniemi 2004; Byckling and Sajaniemi 2006; Sajaniemi and Kuittinen 2005]. Using roles in programming courses also helps students learn strategies related to deep program structures ("knowledge concerning data flow and function of the program reflect deep knowledge which is an indication of a better understanding of the code" [Kuittinen and Sajaniemi 2004]) as opposed to surface knowledge ("program knowledge concerning operations and control structures reflect surface knowledge, i.e., knowledge that is readily available by looking at a program" [Kuittinen and Sajaniemi 2004]). Although roles of variables were originally introduced to help students learn programming, the concept can offer an effective and unique tool to analyze a program for different purposes. In our research, we have extended the application of roles of variables by applying them in the problem of algorithm recognition.

As reported in [Sajaniemi 2002], Sajaniemi identified nine roles that cover 99% of all variables used in 109 novice-level procedural programs. Currently, based on a study on applying the roles in object-oriented, procedural and functional programming [Sajaniemi et al. 2006], a total of 11 roles are recognized. These roles are presented in Table II[1]. Note that the three last roles shown in the table are related to data structures.

Roles are cognitive concepts [Ben-Ari and Sajaniemi 2004; Gerdt and Sajaniemi 2004], implying that human inspectors may have a different interpretation of the role of a single variable. However, as Bishop and Johnson [Bishop and Johnson 2005] and Gerdt [Gerdt and Sajaniemi 2006] describe, roles can be analyzed automatically using data flow analysis and machine learning techniques.

We used the tool developed by C. Bishop and C. G. Johnson [Bishop and Johnson 2005] for automatic detection of roles. After several improvements, the tool is now able to detect the roles we use as beacons in this paper very accurately.

4. APPLYING THE METHOD TO SORTING ALGORITHMS

In this section, we discuss the application of the method to sorting algorithms. We first present the schemas for the analyzed sorting algorithms. For more information about

[1] See the roles of variables Home Page (`http://www.cs.joensuu.fi/~saja/var_roles/`) for a more comprehensive information on roles.

Table II. The roles of variables and their descriptions

Role	Description
Stepper	A variable that systematically goes through a succession of values.
Temporary	A variable that holds a value for a short period of time.
Most-wanted holder	A variable that holds the most desirable value that is found so far.
Most-recent holder	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.
Fixed value	A variable that keeps its value throughout the program.
One-way flag	A variable that can have only two values and once its value has been changed, it cannot get its previous value back again.
Follower	A variable that always gets its value from another variable, that is, its new values are determined by the old values of another variable.
Gatherer	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.
Organizer	A data structure holding values that can be rearranged is a typical example of the organizer role. For example, an array to be sorted in sorting algorithms has an organizer role.
Container	A data structure into which elements can be added or from which elements can be removed.
Walker	Is used for going through or traversing a data structure.

the schemas of sorting algorithms see [Taherkhani 2011a]. After this, we discuss the beacons specific to the sorting algorithms that we used to build the classification tree.

4.1. Schemas for sorting algorithms

We present the schemas for Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort. These algorithms have been chosen for the analysis, because 1) they are commonly used in textbooks of algorithms, and 2) they form two clearly different groups of algorithms which internally have very similar structures. The latter case emphasizes the need to be able to differentiate between closely related algorithms. To complicate the matter further, we discuss the schemas of two variations: Insertion sort WS (Insertion With Swap) and Selection sort WILS (Selection With Inner Loop Swap). Insertion sort WS is a variation of Insertion sort where instead of shifting the elements in the inner loop, they are swapped. Selection sort WILS is a variation of Selection sort that swaps each element that is in a wrong position compared to the element pointed by the loop counter of the outer loop, instead of storing its position and swap it once in the outer loop. To make the difference clear, a typical implementation of standard Selection sort and Selection sort WILS is shown in Figure 3.

The implementations of Insertion sort WS and Selection sort WILS use unnecessary swaps and thus are somewhat more inefficient than the implementations of the standard corresponding algorithms. We noticed these variations in students' submissions in our work reported in [Taherkhani et al. 2012b], where we classified students' implementations of sorting algorithms to see what kind of misconceptions they have related to these algorithms. Though we recognize that tuning selection sort or insertion to gain efficiency is not a value as such, compared to switching to a more efficient algorithm like Quicksort, these cases can be used to demonstrate small performance issues in coding in general. We therefore set a goal that we could use the gained information to develop methods and tools to give automatic feedback on the students' problematic solutions.

```
int i, j, temp;
for(i = 0; i < table.length-1; i++){
   for(j = i+1; j < table.length; j++){
      if(table[j] < table[i]){
         temp = table[i];
         table[i] = table[j];
         table[j] = temp;
      }
   }
}
```

Fig. 3a)

```
int i, j, temp, min;
for(i = 0; i < table.length-1; i++){
   min = i;
   for(j = i+1; j < table.length; j++){
      if(table[j] < table[min]){
         min = j;
      }
   }
   temp = table[min];
   table[min] = table[i];
   table[i] = temp;
}
```

Fig. 3b)

Figure 3. Fig. 3a shows a typical implementation of Selection sort WILS that uses swap in the inner loop instead of the outer loop. A typical implementation of a standard Selection sort is shown in Fig. 3b

The schemas for Bubble sort, Insertion sort, Selection sort, Insertion sort WS and Selection sort WILS are illustrated in Figure 4. Figure 5 shows the schemas for Quicksort and Mergesort. The nesting relationship between the loops and blocks are depicted by the indentations. In [Taherkhani et al. 2012b] and [Taherkhani 2011a], we have discussed the implementational definitions for these sorting algorithms. For example, implementations of Bubble sort include two nested loops and a swap operation performed in the inner loop. Moreover, the two items that are compared in the inner loop in each pass are adjacent. The schemas illustrated in Figures 4 and 5 further abstract these implementational definitions.

As Figure 4 shows, the schemas of Bubble sort, Insertion sort WS and Selection sort WILS are similar. We use algorithm-specific beacons to differentiate between them. Implementations of Insertion sort WS are distinguished using the following beacons. The outer loop of the two nested loops used in these implementations is incrementing and the inner loop decrementing. Moreover, the inner loop counter is initialized to the value of the outer loop counter. The beacon that differentiates between implementations of Bubble sort and Selection sort WILS is that in Bubble sort, as discussed above, the two compared elements in the inner loop in each pass are adjacent, whereas this is not the case in Selection sort WILS implementations (see [Taherkhani et al. 2012b] for more detailed discussion about these features of these algorithms).

In addition to the variations discussed above, in this paper we include two optimized versions of Quicksort and Bubble sort algorithms as well. For Quicksort, the optimization concerns the strategy of selecting the pivot item. We cover two classes of Quicksort implementations. The first class includes the Quicksort implementations that select simply the left or right end of the given array as the pivot (we call the implementations of

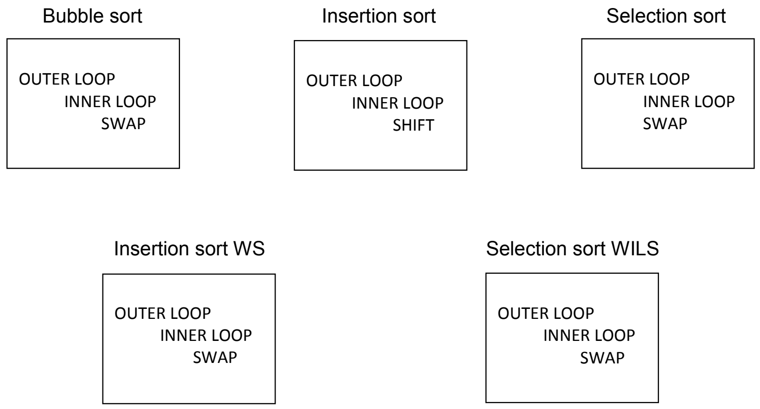

Figure 4. Schemas for implementations of Bubble sort, Insertion sort, Selection sort, Insertion sort WS and Selection sort WILS

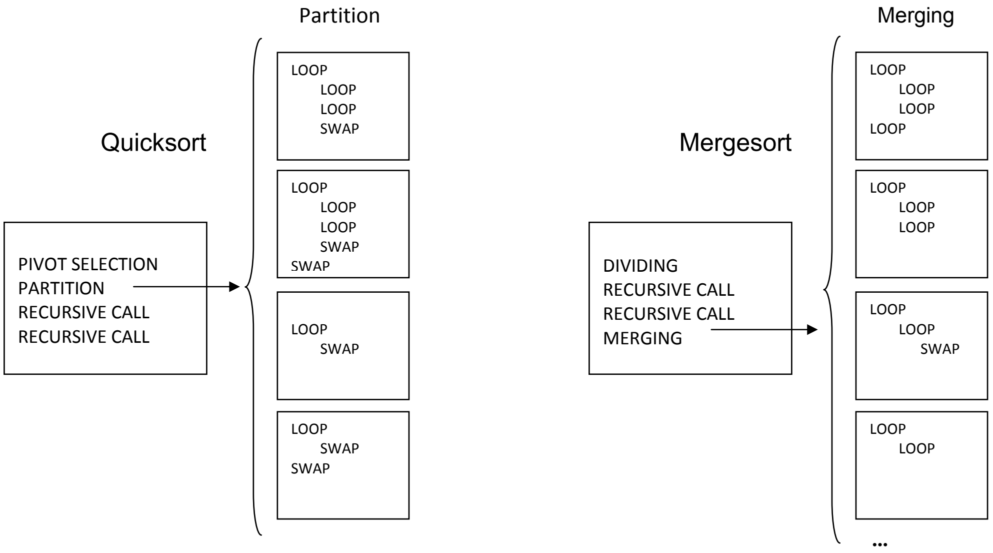

Figure 5. Schemas for implementations of Quicksort and Mergesort. Three dots shown in the Mergesort schemas indicate that merging may have other schemas as well

this class simply Quicksort). This choice of the pivot item results in a worst-case performance on an already sorted array. The second class (i.e., the optimized class) includes the Quicksort implementations that select the middle index or the median of the first, last and middle items of the given array as the pivot (for the rest of the paper we call the implementations of this class Quicksort EP, i.e., Quicksort with Efficient Pivot selection). Dividing Quicksort implementations into these two different classes allows us to give feedback to students about their choice of pivot. In the same way, we divided the implementations of Bubble sort into two classes: Bubble sort WF (Bubble sort With Flag) and Bubble sort. The implementations of the class Bubble sort WF are optimized by using a boolean value as a one-way flag role to indicate whether a swap is done in the inner loop. If no swap is done, the array is sorted and the algorithm can terminate. The implementations of the class Bubble sort do not use such a boolean value, and thus always do all the comparisons, even if the given array is already sorted. Even though students are not encouraged to use Bubble sort algorithm at all, this division of Bubble sort implementations allows us to give more detailed feedback to students about the algorithm they have implemented.

Note that the schemas for the implementations of Quicksort and Quicksort EP are the same, as they are for the implementations of Bubble sort and Bubble sort WF. These optimized and non-optimized versions are separated by beacons, as we will discuss next.

4.2. Beacons for sorting algorithms

Based on the analyses, we found a set of beacons specific to the analyzed sorting algorithms that can be used to separate these algorithms from each other, as well as to distinguish between these and other algorithms. These beacons are computed automatically and used along with the characteristics discussed in Section 3 in building the decision tree that we will present in Section 6.

There are several beacons and as we will discuss in Section 6, not all of them are used as a test in the decision tree. The C4.5 algorithm selects those beacons that can separate the algorithm implementations of the learning data best. The beacons and the algorithm they primarily indicate are as follows:

— *MWH*: whether the implementation of the algorithm includes a variable appearing in a most-wanted holder role. MWH mainly indicates implementations of Selection sort.

— *One_way_flag*: whether the implementation includes a variable appearing in a one-way flag role. This mainly indicates implementations of Bubble sort WF.

— *TEMP*: whether the implementation includes a variable appearing in a temporary role. This mainly indicates implementations of the algorithms that use swap operations.

— *In-place*: whether or not the algorithm implementation uses extra memory. This indicates implementations of the algorithms that do not use auxiliary arrays.

— *Two_nested_loops*: whether the implementation uses two nested loops. This mainly indicates implementations of the non-recursive sorting algorithms and their variations.

— *Swap_outer_loop*: whether a swap operation is used in the outer loop of the two nested loops. This mainly indicates implementations of Selection sort.

— *Swap_inner_loop*: whether a swap operation is used in the inner loop of the two nested loops. This mainly indicates implementations of Bubble sort, Bubble sort WF, Insertion sort WS and Selection sort WILS.

— *OIID*: Outer loop Incrementing Inner Decrementing; whether from the two nested loops used in the implementation of the algorithm, the outer loop is incrementing and the inner decrementing. This mainly indicates implementations of Insertion sort and Insertion sort WS.

— *IITO*: Inner loop counter Initialized To Outer loop counter; whether from the two nested loops used in the implementation of the algorithm, the inner loop counter is initialized to the value of the outer loop counter. This mainly indicates implementations of Insertion sort and Insertion sort WS.

- *Shift_inner_loop*: whether a shift operation is used in the inner loop of the two nested loops. This mainly indicates implementations of Insertion sort.
- *Outer_loop_ini_one*: whether from the two nested loops used in the algorithm, the outer loop counter is initialized to one. This mainly indicates implementations of Insertion sort and Insertion sort WS. In these implementations, often the first element is supposed to be sorted and the second element is compared with the first one and changed with it if needed. Some implementations, however, initialize the outer loop counter to zero and thus perform a useless extra check in the inner loop.
- *Adjacent_compared*: whether the two compared elements in the inner loop are adjacent. This mainly indicates implementations of Bubble sort and Bubble sort WF.
- *Pivot*: whether the implementation includes pivot selection. This mainly indicates implementations of Quicksort and Quicksort EP.
- *Efficient_pivot*: whether the implementation includes efficient pivot selection. This mainly indicates implementations of Quicksort EP.

5. DATA SET

In this work, our goal has been to build a decision tree classifier to classify implementations of the aforementioned sorting algorithms and their variations. Therefore, the data set used for training the classification tree includes only the implementations of these sorting algorithms.

The data we used consists of two data sets: the MULTI-SOURCE data set and the SUBMISSIONS data set. We have collected and used these data sets previously to build a classification tree and evaluate the performance of the classification (the data set MULTI-SOURCE used in [Taherkhani 2011b]), to categorize student-implemented sorting algorithms and their variations (the data set SUBMISSIONS used in [Taherkhani et al. 2012b]), to validate Aari system with authentic students' submissions (the data set SUBMISSIONS used in [Taherkhani et al. 2012a]), and to evaluate the performance of the schema detection techniques (a combination of the data sets MULTI-SOURCE and SUBMISSIONS used in [Taherkhani 2011a]). The instances of the first data set, that is, the MULTI-SOURCE data set, were collected mainly from textbooks and the Web. The second data set (SUBMISSIONS) includes the authentic student submissions in a first year course on data structures and algorithms. The implementations of the data set SUBMISSIONS were collected during two different sorting algorithm assignments; at the very beginning of the course before teaching sorting algorithms, and after giving a lecture on sorting algorithms. The number and percentage of the implementations of each algorithm type for each data set and in total is shown in Table III.

Table III. The number and the percentage of the implementations of each type of sorting algorithm in the MULTI-SOURCE and SUBMISSIONS data sets and in total

Algorithm	**MULTI-SOURCE**	**SUBMISSIONS**	**Total**
Bubble sort	26 (12%)	14 (9%)	40 (11%)
Bubble sort WF	15 (7%)	15 (9%)	30 (8%)
Insertion sort	43 (21%)	17 (11%)	60 (16%)
Insertion sort WS	9 (4%)	10 (6%)	19 (5%)
Selection sort	43 (21%)	36 (23%)	79 (21%)
Selection sort WILS	0 (0%)	13 (8%)	13 (4%)
Quicksort	22 (11%)	15 (9%)	37 (10%)
Quicksort EP	17 (8%)	19 (12%)	36 (10%)
Mergesort	34 (16%)	20 (13%)	54 (15%)
Total	209	159	368

6. CLASSIFICATION TREE

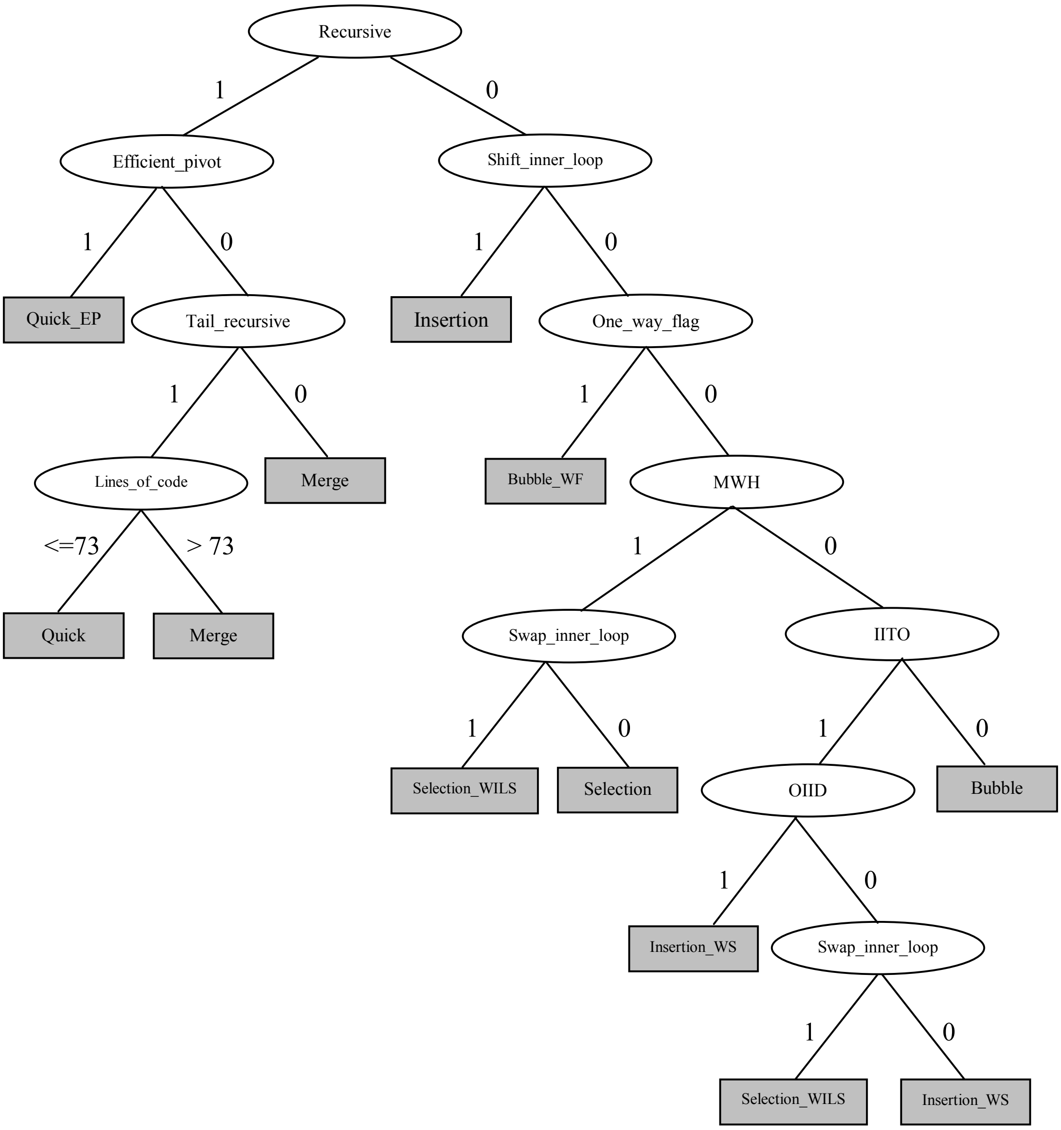

Figure 6. The decision tree classifier generated by the C4.5 algorithm for classifying sorting algorithms and their variations

Classification trees or decision tree classifiers can be used to identify and classify instances of a set into the right classes. Classification trees can be constructed using supervised learning, where first known instances of a training set are used to learn how to classify these instances based on their attributes. After being constructed and trained, classification trees are able to classify the instances of a previously unseen set based on what they have learned in the learning phase. The C4.5 algorithm [Quinlan 1993] is a well-known and commonly used algorithm for constructing classification trees. By using the appropriate techniques for finding the attributes that can separate the classes in the best possible way and for finding the right size for the classification tree, the C4.5 algorithm is able to build understandable trees with highly accurate classification results.

Figure 6 shows the classification tree constructed by the C4.5 algorithm[2] from the instances described in Section 5. Aari analyzes each implementation of the data set and stores the extracted characteristics and beacons in a database. Since the classification tree is produced based on supervised learning and the instances of the data set are used as the training data, each implementation is labeled by its correct type in the database (see Figure 1).

The white ellipses in Figure 6 illustrate the internal nodes that are the tests that determine the splits. The gray rectangles depict the leaves, that is, different sorting algorithms to be recognized. The arcs are labeled with the values that show the outcome of the test performed in each internal node. The classification tree has 12 leaves and 11 internal nodes. The beacons used as the tests to build the classification tree are the following: *Recursive*, *Shift_inner_loop*, *Efficient_pivot*, *Tail_recursive*, *One_way_flag*, *MWH*, *IITO*, *OIID Swap_inner_loop*, and *Lines_of_code* (see Section 4 for the explanations of these beacons). The beacon Recursive is the best split, which is performed in the root. It divides the instances into two recursive and non-recursive groups.

For the recursive instances, the next beacon used as a test is Efficient_pivot, which separates the Quicksort implementations that use efficient pivots, from the Quicksort implementations that do not, and from the Mergesort implementations. The beacon Tail_recursive is used next for separating the Quicksort implementations from the Mergesort implementations. There were four implementations of Mergesort in the student implementations that were incorrectly recognized as tail recursive. The test Lines_of_code is used to separate these Mergesort implementations from the Quicksort implementations. In addition to the algorithm-specific code, these implementations (like all the other students' implementations of the first sorting algorithm assignment; see the data description in Section 5) included a template provided to the students that contained a Java main method, a method that generated an array with random integers and a method that printed the elements of the array before and after sorting.

For the non-recursive instances, the first beacon that is used as a test is Shift_inner_loop. This beacon separates the implementations of Insertion sort from the

[2] Weka data mining software includes J48, an open source Java implementation of the C4.5 algorithm, which we used to build the classification tree. URL: `http://www.cs.waikato.ac.nz/~ml/weka/`

implementations of the other non-recursive sorting algorithms and their variations. The next beacon is the variable role One_way_flag, which separates the implementations of optimized Bubble sort algorithm from the other instances. The variable role MWH is the next beacon that is used to separate the implementations that include most-wanted holder role from those that do not. In the students' implementations, there were a few implementations of Selection sort WILS that include MWH role, since they store the value of the smallest (or largest) element or its position in each pass in the inner loop (as in a standard Selection sort algorithm), even though they do not use it to swap in the outer loop. These implementations perform swap in the inner loop and exchange each element smaller (or larger) than the element pointed by the loop counter of the outer loop in each pass (see Figure 3). The beacon Swap_inner_loop is used next to separate these inefficient Selection sort implementations from the standard Selection sort implementations. For those instances that do not have most-wanted holder role, IITO is the next beacon used as a test to separate Bubble sort implementations from those instances that include this beacon. In Bubble sort implementations, from the two nested loops used in the implementation, the inner loop counter is not initialized to the value of the outer loop counter, but rather to zero or to the length of the given array. The next beacon used for distinguishing between those implementations that include IITO beacon (i.e., the implementations of Insertion sort WS and Selection sort WILS) is OIID. In Insertion sort WS, from the two nested loops used in the implementation of the algorithm, the outer loop is incrementing and the inner decrementing, whereas the implementations of Selection sort WILS do not have this beacon. However, two instances of Insertion sort WS in students' work are implemented in a way that this beacon is not recognized in them. As the results, the C4.5 algorithm uses the beacon Swap_inner_loop to differentiate between these implementations and the implementations of Selection sort WILS. These two student implementations of Insertion sort WS correctly shift the elements in the wrong positions to right, but after each shifting, add the element pointed by the loop counter of the outer loop into this recently shifted position in the inner loop, instead on doing it once in the outer loop after all the necessary shifts are performed in the inner loop. This make these implementations inefficient and thus we classify them as Insertion sort WS. Figure 7 shows the implementation of these variations and compares it with a typical implementation of a standard Insertion sort.

From the characteristics and beacons discussed in Sections 3 and 4, only 10 are selected by the C4.5 algorithm for building the classification tree. The C4.5 algorithm tries to minimize the depth of the classification tree and constructs an accurate classification tree by using a minimum number of beacons. Building a manual classification tree that fulfills these requirements is a very difficult and tedious task and to do it, we need to use appropriate algorithms like the C4.5 algorithm.

```
int i, j, target;
for(i = 1; i < table.length; i++){
   target = table[i];
   j = i;
   while(j > 0 && table[j-1] > target){
      table[j] = table[j-1];
      j--;
      table[j] = target;
   }
}
```

Fig. 7a)

```
int i, j, target;
for(i = 1; i < table.length; i++){
   target = table[i];
   j = i;
   while(j > 0 && table[j-1] > target){
      table[j] = table[j-1];
      j--;
   }
   table[j] = target;
}
```

Fig. 7b)

Figure 7. Fig. 7a shows an implementation of Insertion sort WS that adds the element pointed by the loop counter of the outer loop into each shifted position in the inner loop. Fig. 7b illustrates a typical implementation of a standard Insertion sort

The classification tree of Figure 6 is not able to classify algorithms other than the analyzed sorting algorithms, as the purpose of building the tree has been to find a set of most suitable beacons for classifying these sorting algorithms. The learning data that is used to train the classification tree included only the implementations of these sorting algorithms. Recognizing other algorithms would require computing the related schemas and beacons and building the appropriate classification tree that can recognize them based on these algorithm-specific schemas and beacons.

Each class represented as a leaf in a classification tree can be defined by a set of rules which covers the corresponding path from the root to that leaf [Quinlan 1993]. This allows us to present the beacon-based technical definitions of the analyzed algorithms and their variations as rules. The nine classes included in the classification tree of Figure 6 can thus be defined by the sets of rules illustrated in Table IV. As an example, implementations of Selection sort are those that are not recursive, do not have shift in the inner loop, do not have one-way flag role, have MWH role and do not have swap in the inner loop.

Table IV. Definition of the analyzed algorithms and their variations as rules extracted from the decision tree classifier of Figure 6

Algorithm class	**Rule**
Selection sort	¬ Recursive ∧ ¬ Shift_inner_loop ∧ ¬ One_way_flag ∧ MWH ∧ ¬ Swap_inner_loop
Insertion sort	¬ Recursive ∧ Shift_inner_loop
Bubble sort	¬ Recursive ∧ ¬ Shift_inner_loop ∧ ¬ One_way_flag ∧ ¬ MWH ∧ ¬ IITO
Quicksort	Recursive ∧ ¬ Efficient_pivot ∧ Tail_recursive ∧ Lines_of_code <= 73
Mergesort	Recursive ∧ ¬ Efficient_pivot ∧ (¬ Tail_recursive ∨ (Tail_recursive ∧ Lines_of_code > 73))
Selection sort WILS	¬ Recursive ∧ ¬ Shift_inner_loop ∧ ¬ One_way_flag ∧((MWH ∧ Swap_inner_loop) ∨ (¬ MWH ∧ IITO ∧ ¬ OIID ∧ Swap_inner_loop))
Insertion sort WS	¬ Recursive ∧ ¬ Shift_inner_loop ∧ ¬ One_way_flag ∧ ¬MWH ∧ IITO ∧ (OIID ∨ (¬ OIID ∧ ¬ Swap_inner_loop))
Bubble sort WF	¬ Recursive ∧ ¬ Shift_inner_loop ∧ One_way_flag
Quicksort EP	Recursive ∧ Efficient_pivot

7. ESTIMATED CLASSIFICATION PERFORMANCE

In this section, we present the empirical study conducted for evaluating the estimated classification performance. The results of the empirical study conducted for evaluating the performance of the schema detection method (i.e., the first step of the method) are

Table V. Results of the classification performance

Algorithm	Correct	False	Correct%	False%	Total
Bubble sort	38	2	95.0	5.0	40
Insertion sort	60	0	100.0	0.0	60
Selection sort	79	0	100.0	0.0	79
Mergesort	52	2	96.3	3.7	54
Quicksort	35	2	94.6	5.4	37
Insertion sort WS	15	4	78.9	21.1	19
Selection sort WILS	12	1	92.3	7.7	13
Bubble sort WF	30	0	100.0	0.0	30
Quicksort EP	36	0	100.0	0.0	36
Total	357	11	97.0	3.0	368

reported in [Taherkhani 2011a] (88.3% of the schemas were detected accurately). Thus, in this paper we focus on the performance of the classification.

Cross-validation is a well-known technique for evaluating estimated classification performance. In this technique, the instances of the data set are divided into N subsets. These subsets consist of both training and validation instances. In order to evaluate the classification performance, N different classification trees are generated. From the total N subsets, $N - 1$ subsets are used as training instances to generate a classification tree and one subset is used as the validation instance to evaluate the performance of the generated tree. Therefore, in cross-validation technique, all the subsets participate in generating a tree as the training instances and in evaluating the performance of the classification model as the validation instances. We use leave-one-out cross-validation technique to evaluate the estimated classification performance. In leave-one-out cross-validation, the data set is divided into as many subsets as there are instances in the data set. This uses the data set even more efficiently, as each instance of the data set is once left out to be used for validating the tree generated by the other instances of the data set. There are total of 368 implementations in our data set, thus 368 classification trees are generated in total, using 367 implementations as the training instances and one implementation as the validation instance for each tree.

As shown in Table V, the estimated true positive rate calculated by the leave-one-out cross-validation on the data set discussed above is 97.0%. From the 368 instances, the number of the correctly classified instances is 357 and the number of the incorrectly classified instances is 11 (i.e., 3.0%). The total number of the implementations of each algorithm is shown in the column Total. The number of the correctly classified implementations of each algorithm class is shown in the column Correct and the number of the incorrectly classified implementations of each algorithm class is illustrated in the column False. The percentage of the correctly and incorrectly classified implementations of each class is depicted in the column Correct% and False%, respectively.

We use the following widely used metrics in order to discuss the results of the classification performance in more detail: *True Positive (TP)*, *False Positive (FP)* and *False Negative (FN)*. A TP case occurs when an implementation of an algorithm is correctly recognized. A FP case occurs when an implementation is incorrectly assigned to a class.

Table VI. The value of different metrics that demonstrate the estimated classification performance achieved by leave-one-out cross-validation method

Algorithm	TP	FP	FN	FNR	TPR (recall)	Precision	Total
Bubble	38	4	2	0.050	0.950	0.905	40
Insertion	60	0	0	0	1	1	60
Selection	79	0	0	0	1	1	79
Merge	52	1	2	0.037	0.963	0.981	54
Quick	35	1	2	0.054	0.946	0.972	37
Insertion_WS	15	1	4	0.211	0.789	0.938	19
Selection_WILS	12	2	1	0.077	0.923	0.857	13
Bubble_WF	30	1	0	0	1	0.968	30
Quick_EP	36	1	0	0	1	0.973	36
Total	357	11	11	-	-	-	368

Table VII. The confusion matrix that shows how each instance is recognized as the result of leave-one-out cross-validation method; the row headings indicate the actual type of each algorithm and the column headings indicate what type it was recognized as

Classified as –>	Bub.	Inser.	Selec.	Merge	Quick	Inser. WS	Selec. WILS	Bub. WF	Quick EP
Bubble	38	0	0	0	0	1	1	0	0
Insertion	0	60	0	0	0	0	0	0	0
Selection	0	0	79	0	0	0	0	0	0
Merge	1	0	0	52	1	0	0	0	0
Quick	0	0	0	1	35	0	0	0	1
Insertion WS	2	0	0	0	0	15	1	1	0
Selection WILS	1	0	0	0	0	0	12	0	0
Bubble WF	0	0	0	0	0	0	0	30	0
Quick EP	0	0	0	0	0	0	0	0	36

Finally, a FN case occurs when an implementation that belongs to a class is not assigned to it.

Moreover, based on these metrics, we calculate the following measures for further discussing the performance: *true positive rate (TPR)*, *false negative rate (FNR)* and *precision*. TPR (i.e., recall), is the proportion of correctly recognized positive case implementations from all implementations that should have been recognized as positive cases : *TPR = TP/(TP + FN)*. Correspondingly, FNR is the proportion of incorrectly rejected implementations from all implementations that should have been recognized as positive cases: *FNR = FN/(TP + FN)*. Precision is proportion of correctly recognized positive case implementations from all implementations recognized as positive cases, that is, precision = TP/(TP+FP) [Tan et al. 2006]. Table VI summarizes the results of the classification and the values of the computed metrics for each algorithm class.

To discuss the correctly and incorrectly classified implementations, a *confusion matrix* can be used. A confusion matrix is an $N \times N$ matrix, where N is number of classes. Each instance I_{ij} shows the instance that is from the class I_i, but is identified as the class I_j [Tan et al. 2006]. Those instances positioned on the diagonal are identified correctly. Since there are nine different algorithm classes in our empirical study, we have confusion matrix of size 9×9.

As can be seen from Table VII, for example, from 40 Bubble sort implementations 38 are classified correctly and two are classified incorrectly, one as an implementation of Insertion sort WS, and one as an implementation of Selection sort WILS. As another example, from the 37 implementations of Quicksort, 35 implementations are classified correctly and two implementations are misclassified, one as a Mergesort and one as a Quicksort EP. All the implementations of Insertion sort, Selection sort, Bubble sort WF and Quicksort EP are classified correctly.

8. APPLICATION IN EDUCATION

Algorithms constitute the basic building blocks of computer science. They have an important role in programming and data structure courses. Although in real-life programming projects programmers often use library functions, students should learn the core topics and key algorithms in practice. Deep learning of algorithms involves programming assignments for implementing algorithms, in addition to lectures. As a result, students need to implement several algorithms in order to successfully complete programming courses. Students' implementations need to be checked and personal feedback and suggestions for improvement should be given to students as an important part of learning. However, assessing programming assignments manually is a laborious task. The method introduced in this paper enables teachers to automatically analyze students' implementations in order to check whether they have implemented the required algorithm, and to provide feedback on problematic solutions. Students can use this information to improve their code and learn better.

In the following, we discuss the application of the method in education from both teachers' and students' point of view.

8.1. From a teacher's point of view

In the following, we discuss four ways of how the proposed method can help teachers in their work of assessing student-implemented algorithms and giving feedback to students.

First, different automatic assessment tools are developed to help teachers with their workload of assessing students' work. There is a wide range of topics that these tools cover. For example, some tools check in a black-box manner that a submitted solution works correctly (e.g., [Joy et al. 2005]), some check students' testing skills and some others evaluate programming style. Introduced in this paper, Aari system is a tool that fills an important gap in this range: it checks that the submitted solution implements the required algorithm. The existing automatic assessment tools cannot do this, as they only check that the solution produces the expected output that correspond to a particular input, telling nothing about the algorithm that produces the output (i.e., sorting an array or computing a minimum spanning tree). Knowing what algorithm the submission implements is important because a teacher needs to be sure that students have indeed implemented the required algorithm which can help them learn it in practice.

In data structures and algorithms courses, as an example, sorting is often covered by introducing some basic sorting algorithms, such as Selection sort and Insertion sort, as well as some more efficient algorithms like Quicksort and Mergesort. If a teacher gives an assignment of implementing a Quicksort algorithm so that students can gain a deep understanding of the algorithm, he/she needs some means to check that they have indeed implemented a Quicksort, and not, for example, a Selection sort. In large courses with several hundreds of students, this is a rather time-consuming task. The teacher can apply the method introduced in this paper to ensure this and grade the assignments automatically. Many institutions use automatic tools to check whether students' submissions work correctly (black-box testing), to help students learn algorithms by visual algorithm animation and simulation (e.g., [Malmi et al. 2004]), and for other similar tasks. Aari system can be integrated into these assessment tools and learning environments that are already in used in institutions.

Second, a teacher can use the introduced method to discuss code level issues with students. The method enables a teacher to identify many issues of code quality. He/she can discuss bad programming practices that make the code complicated, such as using too many unnecessary variables, extra storage, etc. These issues help the teacher highlight important higher level principles with students. Why is it important to keep code simple and readable? What is the relation between code readability vs. efficiency? Why bother tuning algorithm code manually, when current compilers often do similar optimizations? These issues detected automatically from students' code allow the teacher urge students to take a critical view on their code.

Third, our algorithm recognition method evaluates the quality of solution automatically. This allow a teacher give better personal comments for the students and pick up interesting examples to discuss with them. For example, Quicksort requires very well-aimed modifications to select the pivot, in order to avoid its well-known $O(n^2)$ worst-case performance with already sorted input. The teacher can use the results of recognizing student-implemented Quicksort algorithms by Aari system to discuss this with students. This is just an example of problems that might occur in students' code and that need to be addressed in all data structures and algorithms courses. Catching these issues by manual investigation, especially in case of large courses and practical programming assignments, is very laborious and there is an obvious need for automating the feedback.

Fourth, students have misconceptions on different topics in basic programming courses, including data structures and algorithms. This is shown, for example, in [Seppälä et al. 2006] in case of heap algorithms and in [Taherkhani et al. 2012b] in case of sorting algorithms. When implementing Insertion or Selection sort, many students include unnecessary swaps in their code ([Taherkhani et al. 2012b]). In case of Insertion sort, as an example, swapping two items in the inner loop instead of shifting the item is a clear indication that the idea of the algorithm has not been understood correctly. These types of misconceptions cannot be detected by existing black-box automatic assessment systems. Our method can detect these issues and help teachers to understand students' misconceptions. This information can be used to give feedback to students on their misconceptions and inappropriate solutions and ask them to correct their solution.

8.2. From students' point of view

Students could use the informative feedback that our method can provide on their implementations to gain a better understanding of their code. As discussed above, students have misconceptions on different algorithms. They, as an example, make problematic implementation choices in the case of sorting algorithms. Using the proposed method, students can get immediate feedback on these kinds of implementation choices. We can reasonably assume that these types of feedback could be justified and beneficial for students and make them rethink their solutions and resubmit a better version. It should be noticed that we need to do similar studies to examine student-implemented variations in other fields of algorithms in order to make Aari a comprehensive feedback providing system.

Students could also use the proposed method to learn good programming style and coding practices. A student can implement, say a Quicksort algorithm, which works correctly and uses efficient pivot selection, etc. In addition to make sure that his/her implementation is correct in these regards, the student would also be interested to know whether his/her code follows good programming style. He/she can submit his/her code to the system and get feedback that the submission works and is efficiently implemented, but, for example, uses more variables that are usually used in Quicksort implementations, uses unnecessary arrays, etc. The student can fix the code and resubmit it and check the feedback again. The student can, for example, use the system's feedback on roles of variables to check the quality of his/her code. As discussed in Section 3, various studies show that using roles of variables have an positive impact on programming education by, for example, providing a conceptual framework for students that helps them construct programs and comprehend them better [Kuittinen and Sajaniemi 2004; Byckling and Sajaniemi 2006; Sajaniemi and Kuittinen 2005]. After detecting a correct implementation of a Selection sort, as an example, Aari system can give feedback that the implementation uses two most-wanted holder roles, which implies that in addition to storing the position of the so far found smallest/largest item in the inner loop, the implementation stores the value of the item in that position as well (we found that many students indeed do implement Selection sort in this way, see [Taherkhani et al. 2012b]). The student may find these types of feedback beneficial to avoid such a programming style that unnecessarily complicates the code.

Another interesting way that a student could use our method is that when he/she encounters a problem while implementing an algorithm, he/she can submit the implementation to Aari system. If Aari recognizes the faulty algorithm, it can provide an appropriate diagnosis and explanation for the problem and suggest a solution. This requires storing algorithm implementations with common misconceptions, errors and problematic solutions in the system's knowledge base and implementing a mechanism to recognize them. This idea is also discussed by Metzger and Wen [Metzger and Wen 2000] as a potential application of their method in programming education.

We need to further evaluate our method in an educational setting and investigate how students use the system and how useful they find it. Since the classification model cannot predict unseen algorithm implementations with perfect correctness, making a summative assessment on whether something is right or wrong might have a negative effect. Therefore, we should give feedback in form of suggesting something that the student should look at. In this regard, we need to situate it in the educational setting and assess how it could improve education. We will investigate this in a separate work.

9. DISCUSSION

We have built a classification model for classifying basic sorting algorithms and their variations which, when measured by leave-one-out cross-validation technique, reaches an estimated true positive rate of 97.0%. Except for the implementations of Insertion sort WS, all the implementations of the other classes are recognized with the true positive rate of above 90% (see Table VI). This suggests that the idea of automatic algorithm recognition based on schemas and beacons is feasible. Moreover, since the results show that we can identify these sorting algorithms and their variations that are so close to each others in terms of implementational definitions (for example, Bubble sort, Bubble sort WF, Insertion sort WS and Selection sort WILS), it is logical to conclude that the algorithms from other fields could be recognized based on algorithm-specific schemas and beacons with a reasonable degree of correctness as well. For the presented set of sorting algorithms, we have found a number of beacons that help us distinguish between the implementations of these algorithms. There are much more beacons for these algorithms than what the C4.5 algorithm selects to build an accurate classification tree. This gives us a reason to believe that also for identifying other algorithms, we can find a set of distinguishing algorithmic-specific beacons. This, however, should be evaluated by empirical tests.

Our data set contains the algorithm implementations collected from different sources including textbooks and the Web (the MULTI-SOURCE data set) and authentic student submissions (the SUBMISSIONS data set). Student submissions constitute about half of the implementations of our data set. Novice programmers use such programming style that makes the resulted code different from those written by experts. Novices use extra variables, unnecessary assignment statement, etc. This wide variation in the implementations of the data set that is used as a part of the training data to construct the classification tree of Figure 6 suggests that the tree is trained so that it is able to classify the wide variation of unseen instances. Moreover, the results of the classification performance can be regarded good also from this perspective.

9.1. Recognizing unseen data

In [Taherkhani et al. 2012a], we used the SUBMISSIONS data set to test how the classification tree built using the MULTI-SOURCE data set performs on previously unseen implementations of Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort algorithms. That classification tree was able to classify the students' implementations submitted in two separate rounds with the true positive rate of 71% and 81%, respectively. The classification model discussed in this paper is expected to perform better on unseen implementations of sorting algorithms implemented by students, since the aforementioned student implementations are also used to train it, and more importantly, because the classification model has a mechanism to deal with the variations of these sorting algorithms that students implement. As an example, there are four student implementations of Mergesort in the data set that use such an unconventional programming style that Aari system incorrectly recognized them as tail recursive implementations (see the classification tree of Figure 6). Presence of this kind of Mergesort implementations in the classification model means that if the similar unusual implementations are encountered in an unseen data set, there is another test in the model (i.e., lines of code) to check whether they are implementations of Mergesort algorithm, although these cases are indeed expected to occur very rarely. This is also a question of validity of our research: the classification tree will fail to recognize those instances that are implemented in a considerably different way than the instances of its training set. As can be seen from the confusion matrix of Table VII, this kind of unconventional programming style also resulted in one Mergesort to be recognized as a Bubble sort because of failure in detecting the recursion.

9.2. Roles of variables

Roles of variables can be utilized as good beacons in the process of algorithm recognition. We used three roles in this research, which are temporary, most-wanted holder and one-way flag. Temporary role did not appear to be a powerful beacon for differentiating between the implementations of the algorithms discussed in this paper, as it is used almost in all the implementations, for example, in swap operations (although most Mergesort implementations do not use swap because of merging, some of them do). However, most-wanted holder and one-way flag roles are very good beacons. Most-wanted holder role distinguishes all the implementations of Selection sort (and some implementations of Selection sort WILS that have a variable appearing in most-wanted holder role, even though they do not use it) from the other implementations. Likewise, one-way flag role differentiates all the implementations of Bubble sort WF from the implementations of the other algorithms (see Figure 6). Although roles of variables alone are not enough to differentiate between all the implementations, they provide a useful complementary tool for recognizing and classifying algorithms when used along with the other beacons. In this paper we have used roles of variables as truth value characteristics. In the case of more complex algorithms, roles can also be computed as numerical characteristics, if they appear to be more informative when used as such (e.g., if the number of particular role in two or more different algorithm implementations can be a distinguishing factor).

We conducted a similar empirical study for building a classification tree and evaluating the performance of the classification in [Taherkhani 2011b]. That classification tree was able to classify only implementations of Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort algorithms. We used the first data set discussed in Section 5 in that empirical study. The estimated classification performance using leave-one-out cross-validation technique was 98.1%. The method described in [Taherkhani 2011b] did not make use of the schema detection technique, but processed all the code of the given program as algorithm-specific code. The present paper addresses that issue and thus improves the method discussed in [Taherkhani 2011b]. We will come back to this in the following.

We also conducted an empirical study to evaluate the performance of the schema detection technique in [Taherkhani 2011a] using the same data set as used in this paper. The tested instances included the implementations of Bubble sort, Insertion sort, Insertion sort WS, Selection sort, Selection sort WILS, Quicksort and Mergesort algorithms. On average, 88.3% of these implementations were recognized accurately. We did not use a classification tree for classifying the implementations in that work. Therefore, if a schema is not detected for some input, recognition fails and the process terminates.

In this work, we have combined the schema detection and classification techniques. We first identify the algorithm-specific code from the non-relevant application data processing code and consider only the algorithm-specific code for further processing, leaving application data processing code out of the process. Only if the schema detection fails, then the original program is further processed. This makes the method more reliable compared with the method used in [Taherkhani 2011b]. Moreover, in this paper, we approach the algorithm recognition problem from the beacon perspective adapted from program comprehension models. We introduce a set of beacons that in addition to the implementations of the algorithms tested in [Taherkhani 2011b], are able to distinguish between the following four algorithm implementations as well: Bubble sort WF, Insertion sort WS, Selection sort WILS and Quicksort EP. Recognizing these variations allows us to give more detailed feedback to students about their problematic and inefficient solutions, in addition to automatically recognize the submissions. In summary, we have achieved almost the same estimated classification performance as reported in [Taherkhani 2011b], but with a more reliable method and with a classification tree that is able to recognize different variations of the basic sorting algorithms as well. The tree is also able to deal with a wider variations of unseen instances, as it has been trained with the implementations from a wider variations. The method discussed in the present paper also improves the method used in [Taherkhani 2011a] in the cases of failure in the schema detection phase; the recognition process is continued in terms of beacon-based recognition using the classification tree also for the failed cases.

Extending the method to cover other fields of algorithms includes efforts for data collection, analyzing algorithms for identifying beacons, system development and performance evaluation. We extended our method to 10 new algorithms, including searching, heap, basic tree traversal and graph algorithms (a data set containing 222 implementations of these algorithms), as reported in [Taherkhani 2012]. It took about three months work of one researcher to perform all the above mentioned steps. It should be noted that for evaluating the performance of the method, we need to gather data from different sources, since no data set exists or is publicly available for this purpose.

9.3. Limitations

Our method has its limitations. First, due to the statistical nature of the method, we cannot claim that the method is able to recognize new previously unseen algorithms 100% accurately. However, even humans make errors and cannot always achieve perfect correctness. The method can classify the solutions in a reasonable precision and thus be of great help for teachers. Second, quality of learning data is very important in supervised machine learning methods. Even though we have collected the instances of our data sets randomly with no preferences to a particular source or alike (which makes our samples representative) it is always possible that some testing instances are implemented in such an idiosyncratic way that cannot be recognized correctly. Third, in order to recognize each type of algorithm, the classification model needs to have a mechanism for it, that is, we need to train the model for that algorithm. Algorithm implementations that the classification tree has no mechanism to recognize will be falsely classified as one of the classes that the tree has been trained to recognize. We have illustrated this in [Taherkhani et al. 2012a], where we asked the students to implement any sorting algorithm, and we conducted an empirical study to recognize the implementations using a classification tree which was trained to recognize implementations of Quicksort, Mergesort, Selection sort, Insertion sort and Bubble sort algorithms. Evaluating the decision tree of Figure 6 on unseen data is beyond the scope of this paper and the reader is encouraged to read [Taherkhani et al. 2012a] for more information on this.

10. CONCLUSION AND FUTURE WORK

We have introduced a method for algorithm recognition that first detects the algorithm-specific code and then extracts the algorithm-specific beacons that are used to recognize implementations of algorithms by a classification tree. We have implemented the method into Aari system for a set of sorting algorithms and their variations and shown that using leave-one-out cross-validation, the method achieves an estimated true positive rate of 97.0%. This can be considered as a very good result that shows the feasibility of the method. The results also show that implementations of sorting algorithms can be represented as characteristic- and beacon-vectors, and that the characteristics and beacons discussed in this paper can describe them and distinguish between them very accurately. By combining the schema detection and classification techniques, we enhance our previous methods discussed in [Taherkhani 2011b] and [Taherkhani 2011a].

We have used roles of variables as beacons in the method and demonstrated that roles can provide a powerful tool in the process of algorithm recognition. From the 11 roles recognized so far, as listed in Section 3, our classification tree uses only two roles. It seems logical to expect that when other fields of algorithms are covered, other roles of variables would also play a role in the process and roles of variables offer even stronger beacons.

The proposed method can be used in educational environment for many purposes. An instructor can utilize the method for checking that students have implemented the required algorithm. This will reduce the workload of the instructor, especially in large courses. In this case, before applying the method to carry out the white-box analysis, an automatic assessment system checks the correctness of the submissions in terms of block-box analysis. In the situations where students are required to implement a particular algorithm but some implementations are classified as negative cases, an instructor can check the result of the classification from the database, where the implementation code and beacons are stored. The method can also be used for giving feedback to students about their inefficient and problematic implementations and making them rethink their solutions. Examples of these kinds of solutions include implementations of Insertion sort and Selection sort algorithms that use unnecessary swaps (we have called these implementations as Insertion sort WS and Selection WILS in this paper, respectively). However, to evaluate how our method can support student learning, we need to investigate the effects of the method on students and show it by empirical data. This is left for future work.

Based on the evaluation results, the method presented in this paper has proved to work well for sorting algorithms and their variations. We have extended the method to cover algorithms from different fields as well. We collected a new data set including 222 implementations of searching, heap, basic tree traversal and graph algorithms. We showed that the schema detection method successfully detected 94.1% of the instances, and the estimated true positive rate of the classification was 97.3% (see [Taherkhani 2012]). This demonstrates that the method can be extended to cover other fields of algorithms with good results. Thus, the proposed method could be used more widely in programming education to provide advanced feedback for students about their algorithmic solutions, either automatically, or semi automatically — providing first valuable analysis information for the teacher, who could then provide better feedback for students in less time.

REFERENCES

ALA-MUTKA, K. 2005. A survey of automated assessment approaches for programming assignments. *Computer Science Education 15,* 2, 83–102.

AMERSHI, S. AND CONATI, C. 2009. Combining unsupervised and supervised classification to build user models for exploratory learning environments. *Journal of Educational Data Mining 1,* 1, 18–71.

BAKER, B. S. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95), 1995*. IEEE Computer Society Washington, DC, USA, 86–95.

BASIT, H. A., PUGLISI, S. J., MCMASTER, W. F. S., TURPIN, A., AND JARZABEK, S. 2007. Efficient token based clone detection with flexible tokenization. In *Proceedings of the 6th European Software Engineering Conference and Foundations of Software Engineering, ESEC/FSE 2007*. ACM New York, NY, USA, 513–516.

BAXTER, I. D., YAHIN, A., MOURA, L., SANT'ANNA, M., AND BIER, L. 1998. Clone detection using abstract syntax trees. In *Proceedings of the 14th IEEE International Conference on Software Maintenance, Bethesda, Maryland, USA, 16–19 March, 1998*. IEEE Computer Society Washington, DC, USA, 368–377.

BELLON, S., KOSCHKE, R., ANTONIOL, G., KRINKE, J., AND MERLO, E. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering 33,* 9, 577–591.

BEN-ARI, M. AND SAJANIEMI, J. 2004. Roles of variables as seen by CS educators. *SIGCSE Bulletin 36,* 3, 52–56.

BISHOP, C. AND JOHNSON, C. G. 2005. Assessing roles of variables by program analysis. In *Proceedings of the 5th Baltic Sea Conference on Computing Education Research, Koli, Finland, 17–20 November*. University of Joensuu, Finland, 131–136.

BROOKS, R. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies 18,* 6, 543–554.

BYCKLING, P. AND SAJANIEMI, J. 2006. Roles of variables and programming skills improvement. *SIGCSE Bulletin 38,* 1, 413–417.

DÉTIENNE, F. 1990. Expert programming knowledge: A schema-based approach. In *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore, Eds. Academic Press, London, 205–222.

DIT, B., REVELLE, M., GETHERS, M., AND POSHYVANYK, D. 2013. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process (JSEP) 25,* 1, 53–95.

DOMINGUEZ, A. K., YACEF, K., AND CURRAN, J. 2010. Data mining to generate individualised feedback. In *Proceedings of the 10th international conference on Intelligent Tutoring Systems (ITS '10) - Volume Part II*. Springer-Verlag Berlin, Heidelberg, 303–305.

DRUMMOND, J. AND LITMAN, D. J. 2010. In the zone: Towards detecting student zoning out using supervised machine learning. In *Proceedings of the 10th international conference on Intelligent Tutoring Systems (ITS '10) - Volume Part II*. Springer-Verlag Berlin, Heidelberg, 306–308.

DUCASSE, S., RIEGER, M., AND DEMEYER, S. 1999. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99), 1999*. IEEE Computer Society Washington, DC, USA, 109–118.

EDWARDS, D., SIMMONS, S., AND WILDE, N. 2006. An approach to feature location in distributed systems. *Journal of Systems and Software 79,* 1, 57–68.

EDWARDS, S. H. 2003. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, California, USA, 26–30 October*. ACM, New York, NY, USA, 148–155.

EISENBERG, A. D. AND VOLDER, K. D. 2005. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05), 2005*. EEE Computer Society Washington, DC, USA, 337–346.

GAY, G., HAIDUC, S., MARCUS, A., AND MENZIES, T. 2009. On the use of relevance feedback in ir-based concept location. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09), Edmonton, Canada, 20–26 September, 2009*. EEE Computer Society Washington, DC, USA, 351–360.

GERDT, P. AND SAJANIEMI, J. 2004. An approach to automatic detection of variable roles in program animation. In *Proceedings of the 3th Program Visualization Workshop, the University of Warwick, UK, 1–2 July*. The University of Warwick, UK, 86–93.

GERDT, P. AND SAJANIEMI, J. 2006. A web-based service for the automatic detection of roles of variables. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, Bologna, Italy, 26–28 June*. ACM, New York, NY, USA, 178–182.

HALSTEAD, M. 1977. *Elements of Software Science*. Elsevier Science Inc, New York, NY, USA.

HARANDI, M. AND NING, J. 1990. Knowledge-based program analysis. *Software IEEE 7,* 4, 74–81.

HIGGINS, C., SYMEONIDIS, P., AND TSINTSIFAS, A. 2002. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, 24–26 June*. ACM, New York, NY, USA, 46–50.

IHANTOLA, P., KARAVIRTA, V., SEPPÄLÄ, O., AND AHONIEMI, T. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling 2010)*.

JOHNSON, W. L. AND SOLOWAY, E. 1984. Proust: Knowledge-based program understanding. In *Proceedings of the 7th international conference on Software engineering, Orlando, Florida, USA, 26–29 March*. IEEE Press Piscataway, NJ, USA, 369–380.

JOY, M., GRIFFITHS, N., AND BOYATT, R. 2005. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing 5,* 3, 1–28.

KOMONDOOR, R. AND HORWITZ, S. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis, SAS 2001, Paris, France, 16–18 July, 2001*. Springer, 40–56.

KONTOGIANNIS, K. A., DEMORI, R., MERLO, E. M., GALLER, M., AND BERNSTEIN, M. 1996. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering 3,* 1–2, 77–108.

KRINKE, J. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE '01), 2001*. IEEE Computer Society Washington, DC, USA, 301–309.

KUITTINEN, M. AND SAJANIEMI, J. 2004. Teaching roles of variables in elementary programming courses. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education (Leeds, United Kingdom, 2004)*. 57–61.

LABEKE, N. V., POULOVASSILIS, A., AND MAGOULAS, G. 2008. Using similarity metrics for matching lifelong learners. In *Proceedings of the 9th international conference on Intelligent Tutoring Systems (ITS '08)*. Springer-Verlag Berlin, Heidelberg, 142–151.

LIU, D., MARCUS, A., POSHYVANYK, D., AND RAJLICH, V. 2007. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07),Atlanta, Georgia, 5–9 November, 2007*. ACM New York, NY, USA, 234–243.

MALMI, L., KARAVIRTA, V., KORHONEN, A., NIKANDER, J., SEPPÄLÄ, O., AND SILVASTI, P. 2004. Visual algorithm simulation exercise system with automatic assessment: Trakla2. *Informatics in Education 3,* 2, 267–288.

MARCUS, A. AND MALETIC, J. I. 2001. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering, San Diego, California, 26–29 November*. IEEE, Washington, DC, USA, 107–114.

MARCUS, A., SERGEYEV, A., RAJLICH, V., AND MALETIC, J. I. 2004. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04), Delft, The Netherlands, 8–12 November, 2004*. EEE Computer Society Washington, DC, USA, 214–223.

MAYRAND, J., LEBLANC, C., AND MERLO, E. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance (ICSM '96), November 1996*. IEEE Computer Society Washington, DC, USA, 244–254.

MCCABE, T. J. 1976. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 308–320.

METZGER, R. AND WEN, Z. 2000. *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization*. The MIT Press.

MIKSATKO, J. AND MCLAREN, B. M. 2008. What's in a cluster? automatically detecting interesting interactions in student e-discussions. In *Proceedings of the 9th international conference on Intelligent Tutoring Systems (ITS '08)*. Springer-Verlag Berlin, Heidelberg, 333–342.

PENNINGTON, N. 1987. Comprehension strategies in programming. *Empirical studies of programmers: second workshop*, 100–113.

POSHYVANYK, D., GETHERS, M., AND MARCUS, A. 2012. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology (TOSEM) 21,* 4.

POSHYVANYK, D. AND MARCUS, A. 2007a. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC '07), Banff, Alberta, Canada, 26–29 June, 2007*. EEE Computer Society Washington, DC, USA, 37–48.

POSHYVANYK, D. AND MARCUS, A. 2007b. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering 33,* 6, 420–432.

QUILICI, A. 1994. A memory-based approach to recognizing programming plans. *Communications of the ACM 37,* 5, 84–93.

QUINLAN, J. R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, USA.

ROY, C. K., CORDY, J. R., AND KOSCHKE, R. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming 74,* 7, 470–495.

SAJANIEMEI, J. AND PRIETO, R. N. 2005. Roles of variables in experts' programming knowledge. In *Proceedings of the 17th Annual Workshop on the Psychology of Programming Interest Group (PPIG '05), University of Sussex, UK.*

SAJANIEMI, J. 2002. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments, Arlington, Virginia, USA, 3–6 September*. IEEE Computer Society Washington, DC, USA, 37–39.

SAJANIEMI, J., BEN-ARI, M., BYCKLING, P., GERDT, P., AND KULIKOVA, Y. 2006. Roles of variables in three programming paradigms. *Computer Science Education 16,* 4, 261–279.

SAJANIEMI, J. AND KUITTINEN, M. 2005. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education 15,* 1, 59–82.

SEPPÄLÄ, O., MALMI, L., AND KORHONEN, A. 2006. Observations on student misconceptions – a case study of the build-heap algorithm. *Computer Science Education 16,* 3, 241–255.

SOLOWAY, E. AND EHRLICH, K. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering 10,* 5, 595–609.

STOREY, M.-A. 2006. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal 14,* 3, 187–208.

TAHERKHANI, A. 2011a. Automatic algorithm recognition based on programming schemas. In *Proceedings of the 23th Annual Workshop on the Psychology of Programming Interest Group (PPIG'11), University of York, UK, 6-8 September, 2011*.

TAHERKHANI, A. 2011b. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *The Computer Journal 54,* 11, 1845–1860.

TAHERKHANI, A. 2012. Schema detection and beacon-based classification for algorithm recognition. In *Proceedings of the 24th Annual Workshop on the Psychology of Programming Interest Group (PPIG'12), London Metropolitan University, London, UK, 21-23 November, 2012*.

TAHERKHANI, A., KORHONEN, A., AND MALMI, L. 2012a. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling 2012), Tahko, Finland, 15–18 November, 2012*. ACM New York, NY, USA, 83–92.

TAHERKHANI, A., KORHONEN, A., AND MALMI, L. 2012b. Categorizing variations of student-implemented sorting algorithms. *Computer Science Education 22,* 2, 109–138.

TAN, P.-N., STEINBACH, M., AND KUMAR, V. 2006. *Introduction to Data Mining*. Addison-Wesley, USA.

TIARKS, R., KOSCHKE, R., AND FALKE, R. 2011. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Control archive 19,* 2, 295–331.

YANG, W. 1991. Identifying syntactic differences between two programs. *Software–Practice and Experience 21,* 7, 739–755.