Schema Detection and Beacon-Based Classification for Algorithm Recognition

Ahmad Taherkhani

Department of Computer Science and Engineering
Aalto University
P.O.Box 15400, FI-00076 AALTO, Finland
ahmad.taherkhani@aalto.fi

Abstract. We introduce a method for recognizing algorithms based on *programming schemas*, which are generic conceptual knowledge with details abstracted out, and *beacons*, which are key statements that suggest existence of specific structures in code. First, the method detects the schemas related to the implementation of the target algorithm and next it computes the characteristics and algorithm-specific beacons from the detected code and uses them as the learning data to construct a classification tree for recognizing new unseen instances.
We demonstrate the method and its performance for searching, heap, basic tree traversal and graph algorithms implemented in Java ($N = 222$). The results show that 94.1% of the schemas are detected correctly and the estimated accuracy of the classification measured by leave-one-out cross-validation technique is 97.3%.

Keywords: POP-II.B. algorithm recognition, detecting programming schemas, program comprehension; POP-I.C. automated assessment

1 Introduction

Programming courses require students to implement a large number of practical exercises. Several automatic assessment tools have been developed to assist teachers in assessing students' work. These tools are capable of analyzing the structure of the program and coding style, verifying that the program works correctly, assessing its run time efficiency, etc. (see, e.g., WebCat [7], CourseMarker [10], Boss [13] and Scheme-robo [20]). Two recent surveys ([1] and [11]) on the functionalities of these tools show that none of them can reliably analyze what kind of algorithms students use to achieve the required functionality and give feedback on how the algorithm is implemented. For example, if students are required to implement a specific sorting algorithm, such as Quicksort, the existing tools can verify by black-box testing that the solution sorts a sequence of integers correctly. However, using these tools, it is very clumsy and unreliable to say whether the implementation conforms to the specification (i.e., whether the algorithm is indeed the required Quicksort, or say Mergesort). This is what we are addressing in our research.

We discuss a combined method for algorithm recognition implemented in a prototype instrument called *Aari system* (an Automatic Algorithm Recognition Instrument). The method combines two different approaches: 1) the schema detection approach, where the implementation of the target algorithm in the given program is detected, and 2) the classification approach, which includes computing characteristics and algorithm-specific beacons that are used as the learning data to train a classifier that is able to classify new previously unseen implementations of algorithms. The main contribution of this paper is to show that the method we have previously presented, as discussed in the following, can be extended to further types of algorithms.

1.1 Background

In our previous work, we have introduced different methods for algorithm recognition and conducted several experiments to show the performance of the methods. We analyzed a set of

different sorting algorithm implementations (Bubble sort, Insertion sort, Selection sort, Mergesort, and Quicksort) and discerned various characteristics, such as Halstead metrics, McCabe complexity and roles of variables in the algorithmic code, which can be used to recognize these algorithms. We built a manually tailored classification tree and conducted an experiment showing that the accuracy of the classification tree was 86% [28]. The next step was to apply the C4.5 algorithm [17] to select the best split from the characteristics and build an automatic classification tree that performs better. Evaluated by leave-one-out cross-validation technique, we showed that the estimated accuracy of the classification was 98.1% [26]. The data sets for these two studies ($N = 287$ and $N = 209$, respectively) were collected mainly from textbooks and the Web. We validated our method by authentic students' sorting algorithm implementations ($N = 192$) and concluded that for the aforementioned sorting algorithms, the average accuracy of the method was about 90% [27]. In addition, using the same data set, we introduced a categorization principle for student-implemented sorting algorithms and their variations [29].

We developed a new method based on programming schemas and evaluated its performance with a data set consisting of the five sorting algorithm implementations from the data sets discussed above ($N = 368$). The method detected 88.3% of the implementations correctly [25].

In this paper, we discuss a combined method where first algorithmic schemas are detected from the target program, and then the characteristics and beacons of the selected algorithmic schemas are further analyzed to build a classification tree. Our previous method for building a classification tree computed the characteristics and beacons from the whole given program. The problem with that method was that the given program might (and often do) include pieces of code irrelevant to the implementation of the target algorithm. By first detecting the piece of code in the given program that implements the target algorithm, and further process only this detected code, the combined method allows us to overcome this limitation and thus achieve a better reliability and performance. While the performance of our methods were tested by sorting algorithms in our previous experiments, the main contribution of this paper is to define schemas and beacons for a new data set consisting of the implementations of searching, heap, basic tree traversal and graph algorithms and apply the combined method to this data to evaluate its performance. The promising results show the generalizability of the proposed method.

We start by presenting related work in Section 2. Section 3 gives an overview of the method. Section 4 presents the data set and discusses the method more specifically for the algorithms of the data set. In Section 5, we introduce the classification tree generated for recognizing these algorithms. Section 6 presents the results followed by a discussion in Section 7. The paper ends in some conclusions and future work.

2 Related work

Program comprehension (PC) has been studied from both theoretical and practical points of view. Theoretical studies focus on understanding how programmers comprehend programs, what elements affect the comprehension process, what stages there are in the progress from novice to expert, etc. Different models for PC have been introduced. We will get back to some of these studies in Section 3 when discussing the theoretical background of our method.

Practical studies on PC have focused on developing techniques to facilitate the comprehension process. These techniques have been influenced by the models resulted from the theoretical studies. The characteristics that influence cognitive strategies used by programmers also influence the requirements for supporting tools [24]. By extracting the knowledge from the given program, PC tools can be applied to different problems such as teaching novices, generating documentation from code, restructuring programs and code reuse [16].

PC techniques are mainly based on stereotypical programming plans (also called schemas, idioms, etc.), which are stored in a knowledge base. Understanding the given program is carried out by analyzing the code to find pieces that match the set of plans from the knowledge base. Extracting and matching plans can be performed in *top-down*, *bottom-up* or *hybrid* manner.

Top-down approaches start by the goal of the target program and use it to find the correct plans from the knowledge base. This results in a higher probability to find the right plans from the knowledge base and thus make the searching and matching more effective. However, these approaches need the specification of the target program which are not necessarily available (see, as an example, [12]). In bottom-up approaches, the process of searching and matching is started from small plans and continued to bigger ones. Because small plans can be part of different bigger plans, this technique may become ineffective as the size of knowledge base grows (see, for example, [9]). In hybrid approaches both techniques are used (see, e.g., [16]).

The purpose of algorithm recognition is to determine what algorithm a piece of code implements. Therefore, algorithm recognition facilitates PC. Algorithm recognition can be applied in various tasks including assessing students' work (as discussed in this paper), detecting plagiarism (see, e.g., [8,18]), detecting clones in code (see, for example [2,19]) and source to source program translation via abstraction and reimplementation [30]. In [15], Metzger and Wen discuss a method for replacing algorithms with parallel algorithms that perform the same task. This type of code optimization can be applied to develop compilers for parallel processing machines.

3 Method

In this section, we discuss the combined method very briefly. For a more detailed discussion on the schema detection and classification approaches see [25] and [26], respectively.

The combined method has two main phases illustrated in Figure 1. In the first phase the schemas for the target algorithms are detected (along with the beacons necessary for detecting the schemas). As the result, the code related to the implementation of the algorithm in question is selected for analysis and other non-relevant code is not processed further. The second phase includes extracting and storing the characteristics and beacons[1], building a classification tree and evaluating the estimated accuracy of the classification. In this phase, a classifier is trained to learn how each algorithm class can be associated with specific characteristics and beacons. Thus, the implementations of the learning data are labeled by the correct type of the corresponding algorithm (this is denoted by the dashed arrow in Figure 1). It should be noted that Steps 3 and 4 in the figure are independent from each other. This means that before we build a classification tree, we can evaluate the performance of the classification. 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 illustrates the process of classifying a new unseen data set after the training process of Figure 1. There are two main differences between this process and the training process. First, it starts with testing that the given program works correctly (only error-free inputs will be processed). This step is not needed in the training process of Figure 1, because the data is known. Second, it gets the classification tree already constructed in the training process and ends with recognizing the previously unseen instances of the given data set. These differences are highlighted by gray rectangles in Figure 2. In this paper, we present an experiment that covers the steps illustrated by Figure 1, that is, we discuss the process of building a classification tree and evaluating the estimated accuracy of the classification using leave-one-out cross-validation. We have previously conducted an experiment that covers the steps presented in Figure 2 for sorting algorithms (see [27]). Conducting a similar experiment is out of the scope of this paper and will be reported elsewhere.

3.1 Schemas and beacons

Schemas and beacons are at the core of many program comprehension models. Soloway and Ehrlich define *plans* (which correspond to schemas in their terminology) as stereotypical action

[1] Basically, characteristics, such as software metrics, are the common features that are computed and used for all fields of algorithms, whereas beacons are the features specific to a particular field of algorithm.

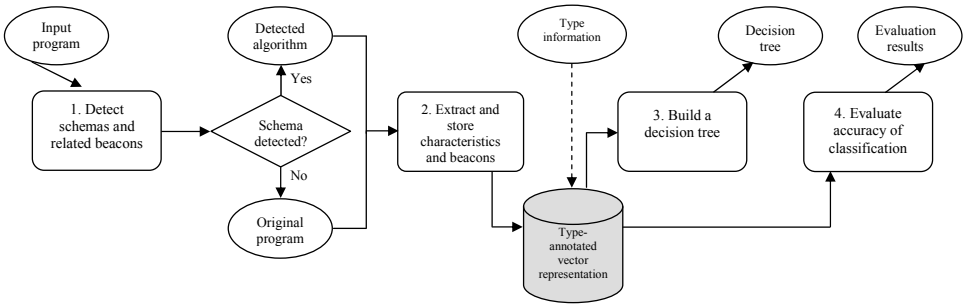

Fig. 1. The process of building a decision tree and evaluating the estimated accuracy of the classification

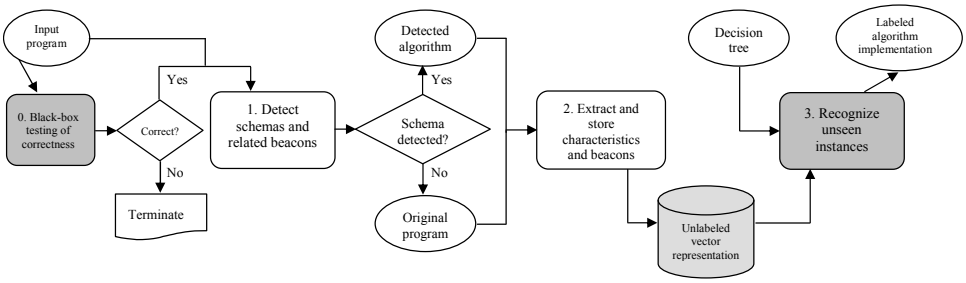

Fig. 2. An overview of the process of recognizing previously unseen algorithm implementations

structures [23]. Détienne defines schemas as formalized knowledge structures in programs [6]. Programmers create and store schemas at different levels of abstraction, and developing schemas is what turns novices into experts. Beacons provide a link between source code and the process of verifying the hypotheses driven from the source code, helping programmers to accept or reject their hypotheses about the code. Beacons suggest the existence of a particular structure in code and experts use them to comprehend programs [4]. In [23], Soloway and Ehrlich define *critical lines* (which can be thought of as beacons) as highly informative and representative lines that are strong indications of a specific plan.

We utilize schemas and beacons to recognize algorithms automatically. The idea is to store abstracted stereotypical implementations of algorithms into a knowledge base of an automatic tool so that the tool can use them to recognize different implementations of those algorithms despite differences in implementation details. This is a similar process as what the experts do while trying to comprehend new programs. We will explain this for our data set in Section 4.

3.2 Creating characteristic and beacon vectors

We compute three types of characteristics: *numerical characteristics*, *truth value characteristics* and *structural characteristics*. The numerical characteristics include *number of operators*, *number of operands*, *number of unique operators*, *number of unique operands*, *program length* (total number of operators + total number of operands), *program vocabulary* (number of unique operators + number of unique operands), *lines of code*, *number of assignment statements*, *cyclomatic complexity (i.e., McCabe complexity [14])*, *number of variables*, *number of loops*, *number of nested loops* and *number of blocks*. Truth value characteristics consist of *recursive* (whether the target algorithm uses recursion), *tail recursive*, *using an auxiliary array* (for algorithms that use arrays) and *roles of variables* (automatically recognized roles of the variables, see [22]).

The structural characteristics help us identify language constructs and different patterns and compute algorithm-specific beacons. These include *block/loop information*, *loop counter information*, and *dependency information*.

We have implemented the method in Aari system. Aari detects schemas and computes the characteristics and beacons for programs written in Java. The input algorithm implementations are converted into characteristic and beacon vectors, which we call *technical definitions* of the implementations. These vectors are given to the C4.5 algorithm [17], which selects the characteristics that best separate the instances of the data set and builds a classification tree.

4 Experiment

We applied the method to searching, heap, basic tree traversal and graph algorithms.

4.1 The analyzed algorithms and data set

We analyzed 10 algorithms from different fields. Since analyzing a bigger set of algorithms is beyond the scope of this paper, our goal was to examine a set of well-known basic algorithms that are commonly discussed in data structures and algorithms courses and textbooks.

We collected a total of 222 algorithm implementations from various textbooks and educational web pages. Table 1 shows the number and the percentage of the implementations of the analyzed algorithms, as well as the abbreviation used for each algorithm in this paper. As indicated in the table, we analyzed the recursive version of depth first search algorithm (DFS) and the non-recursive versions of heap insertion and remove algorithms[2]. Many of the collected programs, especially those collected from the Web, in addition to the code related to the implementation of the algorithm, included non-relevant code as well, such as code related to reading in user provided data, printing the processed data and testing the implementation.

Table 1. The number and percentage of the implementations of the analyzed algorithms. The last column shows the abbreviation used for the algorithms

Algorithm	**Number (%)**	**Abbreviation**
Non-recursive BinSearch	36 (16%)	NBS
Recursive BinSearch	13 (6%)	RBS
Depth First Search (recursive)	15 (7%)	DFS
Inorder traversal	23 (10%)	InT
Preorder traversal	24 (11%)	PreT
Postorder traversal	22 (10%)	PostT
Heap insertion (non-recursive)	22 (10%)	HeapI
Heap remove (non-recursive)	21 (9%)	HeapR
Dijkstra's algorithm	23 (10%)	Dijkstra
Floyd's algorithm	23 (10%)	Floyd
Total	222	-

4.2 Schemas for the algorithms

Figure 3 illustrates the schemas for the analyzed algorithms. We examined all the implementations of the data set and determined an *implementational definition* for each algorithm. We define an implementational definition of an algorithm as the abstraction of its implementation, which reflects the functionality and structure of the algorithm. Implementational definitions do not include implementation details, such as the type of loops or variables, but only high level structural and functional features of algorithms. The schemas depicted in Figure 3 further abstract the implementational definitions of the analyzed algorithms.

[2] DFS has also a well-established non-recursive version, and heap insertion and remove algorithms have also well-established recursive versions. However, we could not gather enough samples of these versions for analysis.

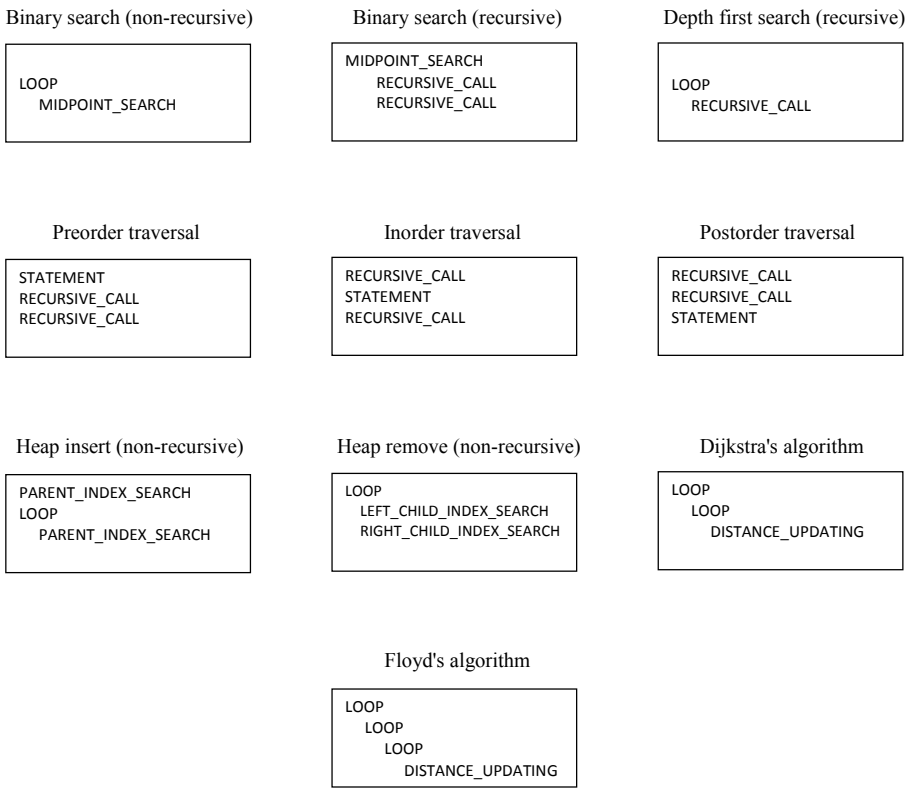

Fig. 3. The schemas for the analyzed algorithms

For the algorithms of the data set that have well-established recursive and non-recursive versions, the version we analyzed is indicated in the parentheses after their name. Furthermore, indentations indicate the nesting relationship between the loops and blocks.

In the following, we elaborate on some parts of the schemas with semantic meaning and explain how they are computed.

- In the schema of binary search algorithm: $MIDPOINT\_SEARCH$ involves computing the midpoint of a sorted sequence, for example, $mid = (low + high)/2$.
- In the schemas of preorder, inorder and postorder traversal algorithms: $STATEMENT$ denotes whatever function (examining, printing, updating) that may be performed when a node of a binary tree is visited.
- In the schema of heap insertion algorithm: $PARENT\_INDEX\_SEARCH$ denotes computing the index of the parent of a given node with index i, which is $i/2$.
- In the schema of heap remove algorithm: $LEFT\_CHILD\_INDEX\_SEARCH$ for a node with index i is $2i$ and $RIGHT\_CHILD\_INDEX\_SEARCH$ correspondingly $2i + 1$. Some implementations compute the index of the right child of a node by simply incrementing the index of its left child by one, instead of computing it using the index of the node[3].
- In the schemas of Dijkstra's and Floyd's algorithms: existence of the operation denoted by $DISTANCE\_UPDATING$ (also called *relaxation* for Dijkstra's algorithm, e.g., in [5]) in code is investigated by examining whether the given implementation includes the following statements in the nested loops: **if** $v.d > u.d + w(u, v)$ **then** $v.d = u.d + w(u, v)$. That is, the process of $DISTANCE\_UPDATING$ for an edge (u, v) involves examining whether the so far found shortest path to the vertex v can be improved by going through the vertex u, and updating the shortest path to v if this is the case.

[3] If the tree root is at index 0, the parent, left child and right child of each node is located in $(i - 1)/2$, $2i + 1$ and $2i + 2$. We have considered these cases in the implementation of our schema detection method as well.

Note that the schemas of Figure 3 show abstract typical implementations of the algorithms and that slightly different implementations are also possible. For example, some implementations of non-recursive heap remove algorithm might perform $LEFT\_CHILD\_INDEX\_SEARCH$ once before the loop and at the end of the loop. As another example, some implementations of Dijkstra's algorithm might have more than one loop within the outer loop. We have not shown these details in the schemas but considered them in the implementation of Aari system.

4.3 Beacons

We analyzed the implementations of the data set to find the following set of beacons specific to the analyzed algorithms that can be used for identifying them. We automatically compute these beacons and give them, along with the computed characteristics discussed in Section 3, to the C4.5 algorithm which selects the best separating beacons and characteristics to generate a decision tree for the classification task. We will present the decision tree in Section 5.

- *MPSL*: MidPoint Search in a Loop; whether the implementation of the algorithm includes searching midpoint of an array within a loop. This mainly indicates implementations of non-recursive binary search algorithm.
- *MPBR*: MidPoint Before Recursion; whether the implementation includes searching midpoint before two recursive calls. This mainly indicates implementations of recursive binary search algorithm.
- *REIL*: REcursion In Loop; whether the implementation includes a recursive call within a loop. This mainly indicates implementations of depth first search algorithm.
- *TSRC*: Two Sequential Recursive Calls; whether the implementation includes two sequential recursive calls. This mainly indicates implementations of preorder and postorder tree traversal algorithms and separates these implementations from implementations of inorder traversal algorithm.
- *TPNI*: Two Parent Nodes Index search; whether the implementation includes searching the indexes of two parent nodes before and after a loop. This mainly indicates implementations of heap insertion algorithm.
- *LRCI*: Left and Right Child node Index search; whether the implementation includes searching the indexes of the left and right child nodes within a loop. This mainly indicates implementations of heap remove algorithm.
- *DUTWL*: Distance Update within TWo nested Loops; whether the implementation includes distance updating (i.e., relaxation) performed within two nested loops. This mainly indicates implementations of Dijkstra's algorithm.
- *DUTHL*: Distance Update within THree nested Loops; whether the implementation includes distance updating performed within three nested loops. This mainly indicates implementations of Floyd's algorithm.

5 Classification tree

Figure 4 illustrates the decision tree classifier generated by the C4.5 algorithm[4]. Using Aari, we automatically analyzed all the implementations of the data set (Table 1) and stored the computed characteristics and beacons in a database. As depicted in Figure 1, because the implementations of the data set are used as the training data and the decision tree is generated based on supervised learning, each implementation is labeled by its correct type in the database.

In the decision tree, the internal nodes, which are depicted by the white ellipses, include the tests that determine the splits. The leaves are illustrated by the gray rectangles and indicate the

[4] J48 (from Weka data mining software), which is an open source Java implementation of the C4.5 algorithm is used to construct the classification tree. URL: `http://www.cs.waikato.ac.nz/~ml/weka/`

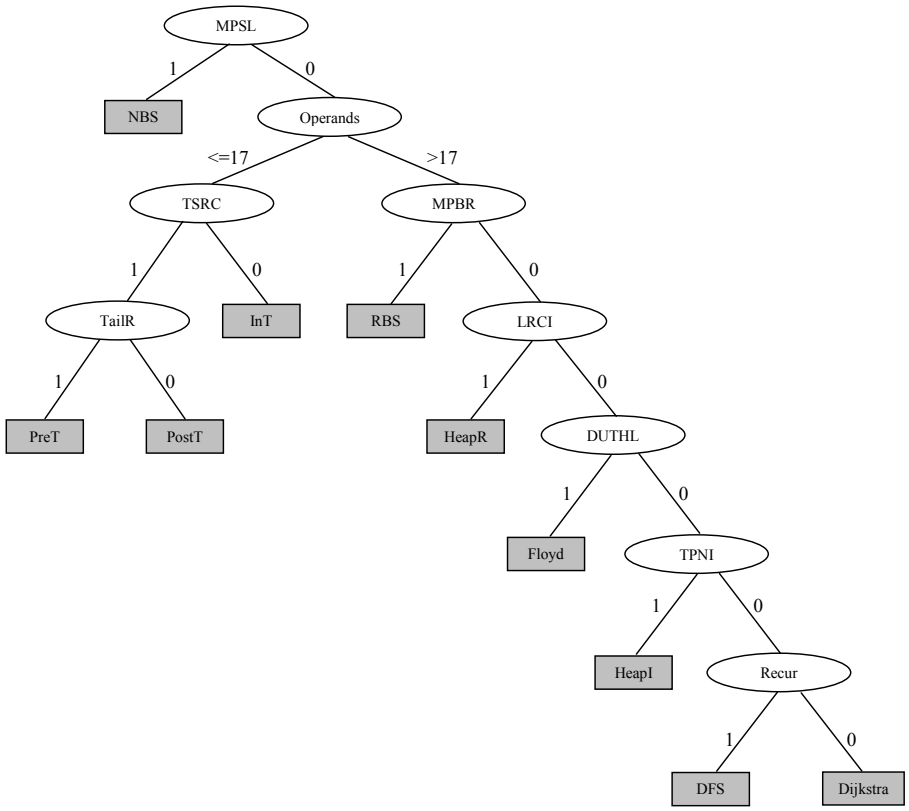

Fig. 4. The classification tree constructed by the C4.5 algorithm on the algorithm implementations presented in Table 1 (see the table for the abbreviations)

analyzed algorithms. The information of the arcs shows the outcome of the test performed in each internal node and determines which child is visited next. The decision tree has 10 leaves and 9 internal nodes (with the root included). From the beacons discussed in Section 4, the following six beacons are selected by the C4.5 algorithm to be used as the tests in the classification tree: *MPSL*, *TSRC*, *MPBR*, *LRCI*, *DUTHL* and *TPNI*. In addition, the characteristics *Recursive*, *Tail_recursive* and *number of operands* are used as the tests in the decision tree as well.

Table 2. Definition of the analyzed algorithms as rules based on the classification tree

Algorithm class	**Rule**
Non-recursive BinSearch	MPSL
Recursive BinSearch	¬ MPSL ∧ Operands > 17 ∧ MPBR
Depth first search	¬ MPSL ∧ Operands > 17 ∧ ¬ MPBR ∧ ¬ LRCI ∧ ¬ DUTHL ∧ ¬ TPNI ∧ Recur
Inorder traversal	¬ MPSL ∧ Operands <= 17 ∧ ¬ TSRC
Preorder traversal	¬ MPSL ∧ Operands <= 17 ∧ TSRC ∧ TailR
Postorder traversal	¬ MPSL ∧ Operands <= 17 ∧ TSRC ∧ ¬ TailR
Heap insertion	¬ MPSL ∧ Operands > 17 ∧ ¬ MPBR ∧ ¬ LRCI ∧ ¬ DUTHL ∧ TPNI
Heap remove	¬ MPSL ∧ Operands > 17 ∧ ¬ MPBR ∧ LRCI
Dijkstra's algorithm	¬ MPSL ∧ Operands > 17 ∧ ¬ MPBR ∧ ¬ LRCI ∧ ¬ DUTHL ∧ ¬ TPNI ∧ ¬ Recur
Floyd's algorithm	¬ MPSL ∧ Operands > 17 ∧ ¬ MPBR ∧ ¬ LRCI ∧ DUTHL

Based on the decision tree of Figure 4, we can describe each analyzed algorithm (i.e., each class represented as a leaf in the decision tree) as a set of rules [17]. Each set of rules presents the beacon-based technical definition of the corresponding algorithm and covers the path from the root to the leaf for that algorithm. These rules are shown in Table 2. We can, for example, express

the implementations of Floyd's algorithm as those that do not have $MPSL$, have number of operand more than 17, do not have $MPBR$, do not have $LRCI$ and include $DUTHL$.

6 Results

We evaluated both the performance of the schema detection method and the estimated accuracy of the classification using the data set described in Table 1. We first present the results of the schema detection followed by the results of the estimated accuracy of the classification.

6.1 Results of schema detection

All the algorithmic schemas of the implementations of recursive and non-recursive binary search, as well as the implementations of inorder and postorder traversal algorithms are detected correctly. For DFS, preorder traversal and heap insertion implementations the accuracy of the schema detection method is more than 90%. For the rest of the implementations, the accuracy of the method is more than 80%. From all the 222 implementations, the schemas for 209 implementations are detected correctly, that is, the average accuracy of the method is 94,1%. Table 3 summarizes these results.

Table 3. The number and percent of the correctly detected algorithmic schemas

Algorithm	**Detected (%)**	**Not detected (%)**	**Total**
Non-recursive BinSearch	36 (100)	0 (0)	36
Recursive BinSearch	13 (100)	0 (0)	13
Depth first search	14 (93,3)	1 (6,7)	15
Inorder traversal	23 (100)	0 (0)	23
Preorder traversal	23 (95,8)	1 (4,2)	24
Postorder traversal	22 (100)	0 (0)	22
Heap insertion	21 (95,5)	1 (4,5)	22
Heap remove	18 (85,7)	3 (14,3)	21
Dijkstra's algorithm	19 (82,6)	4 (17,4)	23
Floyd's algorithm	20 (87,0)	3 (13,0)	23
Total	209 (94,1)	13 (5,9)	222

6.2 Results of the evaluation of the estimated classification accuracy

Cross-validation is a well-known technique for estimating the performance of a classification model. Cross-validation has different types. In k-fold cross-validation, the data set is partitioned into k subsets that include both training and validation data. The accuracy of the classification is evaluated by constructing k different classification trees using $k-1$ subsets as the training set to construct a classification tree and one subset as the validation set to evaluate the performance of the constructed tree. We evaluated the estimated accuracy of the classification using leave-one-out cross-validation technique, where a single instance of the data set is used as the validation data and the remaining instances are used as the training data. Therefore, the training set includes $N-1$ instances and the validation set a single instance. This process is repeated such that each instance of the data set is used once as the validation data. Our data set includes 222 implementations, and therefore 222 classification trees are constructed using 221 instances as the training data and one instance as the validation data for each tree.

The results of the evaluation of the estimated classification accuracy are summarized in Table 4. The column Total shows the total number of the implementations of each algorithm. The column Correct (%) shows the number and percentage of the correctly classified implementations

of each algorithm and the column False (%) indicates the number and percentage of the falsely classified implementations of each algorithm. The last column shows what type the falsely classified implementations are recognized as. From 222 instances of the data set, 216 instances are classified correctly (97.3%) and 6 instances are classified falsely (2.7%).

Table 4. The estimated accuracy of the classification evaluated by leave-one-out cross-validation technique

Algorithm	**Correct (%)**	**False (%)**	**Total**	**Falsely recognized as**
Non-recursive BinSearch	35 (97,2)	1 (2,8)	36	Inorder traversal
Recursive BinSearch	13 (100)	0 (0)	13	-
Depth first search	14 (93,3)	1 (6,7)	15	Dijkstra
Inorder traversal	23 (100)	0 (0)	23	-
Preorder traversal	23 (95,8)	1 (4,2)	24	Inorder traversal
Postorder traversal	22 (100)	0 (0)	22	-
Heap insertion	21 (95,5)	1 (4,5)	22	Depth first search
Heap remove	21 (100)	0 (0)	21	-
Dijkstra's algorithm	22 (95,7)	1 (4,3)	23	Depth first search
Floyd's algorithm	22 (95,7)	1 (4,3)	23	Dijkstra
Total	216 (97,3)	6 (2,7)	222	-

7 Discussion

Programming courses require students to solve several practical exercises. Assessing students' solutions especially in large courses is a time-consuming task. A teacher can use the presented method to asses these solutions in the cases where the assignments require students to implement a specific algorithm. This allows the teacher to concentrate on the solutions that do not conform to the specification, instead of assessing all the implementations manually. The method can also be further developed to recognize variations of student-implemented algorithms and provide informative feedback to students about their solutions. This can be done by examining students' solutions and identifying the variations they implement. We have done it in the case of sorting algorithms and reported the results in [29]. Aari system can be trained by the implementations of these variations to identify previously unseen similar variations. Another application of the method in computer science education is detecting plagiarism in students' work, which can be achieved with slight modifications.

The method has potential to be applied to software engineering related tasks as well. In clone detection, as an example, the task is to locate similar pieces of code. Another example includes program translation via abstraction and reimplementation [30], which is a well-known technique for source to source translation. With appropriate further developments, our method is capable of performing these tasks for the implementations that are stored in its knowledge base. However, since these activities involve dealing with large-scale software (unlike the implementations in computer science education), the performance of the method in this context should be evaluated with empirical tests before drawing further conclusions.

As discussed in Section 2, we previously applied the schema detection and classification methods to sorting algorithms (see [25] and [26], respectively). In this paper, we have demonstrated that these methods are generalizable by applying them to the algorithms from various fields with practically the same accuracy. This suggests that we can safely claim that the methods can be extended to cover more algorithms from different fields with fairly high accuracy. The main steps of extending the method to cover other types of algorithms include analyzing those algorithms to identify the schemas that can represent them and the beacons that can strongly

indicate these algorithms. Once these are defined, the next step is to develop a tool that can automatically recognize these schemas and extract the beacons (along with the characteristics discussed in Section 3) to be used by the C4.5 algorithm for constructing a suitable classification tree.

In Section 4, we introduced eight beacons for the algorithms discussed in this paper. Two of these beacons, namely *REIL* and *DUTWL* are not used by the C4.5 algorithm in constructing the decision tree of Figure 4. These beacons indicate implementations of depth first search and Dijkstra's algorithms respectively. These two algorithms are distinguished by the characteristic *Recursive* and thus their indicative beacons are left out from the tree. The C4.5 algorithm selects attributes that can discriminate between different classes of data in the best possible way, trying to keep the size of the tree as small as possible.

As in our previous studies, we used the tool developed by C. Bishop and C. G. Johnson [3] for automatically detecting roles of variables in this study. The tool, however, did not detect the roles of the variables in the implementations of the data set accurately enough. With a more accurate role recognizer, roles of variables could have played a distinguishing role in the decision tree of Figure 4 (like they did in our previous studies). For example, the *low* index in implementations of binary search (e.g., $low = middle + 1$) has a *follower* role ([21]) that could be a good beacon for identifying these implementations. We are looking for a better role detector for our future work.

8 Conclusion and future work

We have discussed a combination of two different methods for algorithm recognition and evaluated their performance. As Tables 3 and 4 show, both methods perform very accurately (94,1% and 97,3% of accuracy, respectively). In the combined method, the schema detection method first identifies the code related to the implementation of the algorithm in question. This improves the reliability of the recognition, since the characteristics and beacons are computed from the detected schemas, and not from the whole program.

In real-life programming projects, programmers often use existing standard libraries. However, in a data structures and algorithms course, students need to implement many programming assignments themselves. Aari system can help instructors to check that students have implemented the required algorithm that conforms to the specification. Before this, the correctness of the solutions can be tested by an automatic assessment tool that performs black-box testing.

We applied our methods to sorting algorithms in [25] and [26]. In this paper we have shown that the methods can be extended to cover other fields of algorithms. As a direction of future work, we will further develop the methods to deal with more algorithms and their variations.

9 Acknowledgment

The author would like to thank Lauri Malmi and Ari Korhonen for their valuable comments.

References

1. K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.
2. S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33:577–591, 2007.
3. C. Bishop and C. G. Johnson. Assessing roles of variables by program analysis. In *Proceedings of the 5th Baltic Sea Conference on Computing Education Research, Koli, Finland, 17–20 November*, pages 131–136. University of Joensuu, Finland, 2005.
4. R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, USA, 2009.
6. F. Détienne. Expert programming knowledge: A schema-based approach. In J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore, editors, *Psychology of Programming*, pages 205–222. Academic Press, London, 1990.
7. S. H. Edwards. 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*, pages 148–155. ACM, New York, NY, USA, 2003.
8. B. S. Elenbogen and N. Seliya. Detecting outsourced student programming assignments. In *Journal of Computing Sciences in Colleges*, pages 50–57. ACM, 2007.
9. M. Harandi and J. Ning. Knowledge-based program analysis. *Software IEEE*, 7(4):74–81, 1990.
10. C. Higgins, P. Symeonidis, and A. Tsintsifas. 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*, pages 46–50. ACM, New York, NY, USA, 2002.
11. P. Ihantola, V. Karavirta, O. Seppälä, and T. Ahoniemi. 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)*, 2010.
12. W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. In *Proceedings of the 7th international conference on Software engineering, Orlando, Florida, USA, 26–29 March*, pages 369–380. IEEE Press Piscataway, NJ, USA, 1984.
13. M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3):1–28, 2005.
14. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2:308–320, 1976.
15. R. Metzger and Z. Wen. *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization.* The MIT Press, 2000.
16. A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.
17. J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, USA, 1993.
18. S. S. Robinson and M. L. Soffa. An instructional aid for student programs. In *Proceedings of the 11th SIGCSE technical symposium on Computer science education, Kansas City, Missouri, USA, 14–15 February*, pages 118–129. ACM, New York, NY, USA, 1980.
19. C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74:470–495, 2009.
20. R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*, pages 133–136, Canterbury, UK, 2001. ACM Press, New York.
21. J. Sajaniemei. Visualizing roles of variables to novice programmers. In *Proceedings of the 14th Annual Workshop on the Psychology of Programming Interest Group (PPIG '02), Brunel University, London, UK.*, 2002.
22. J. Sajaniemi. 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*, pages 37–39. IEEE Computer Society Washington, DC, USA, 2002.
23. E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.
24. M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.
25. A. Taherkhani. 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*, 2011.
26. A. Taherkhani. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *The Computer Journal*, 54(11):1845–1860, 2011.
27. A. Taherkhani, A. Korhonen, and L. Malmi. 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*, 10 pages, accepted.
28. A. Taherkhani, A. Korhonen, and L. Malmi. Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms. *The Computer Journal*, 54(7):1049–1066, 2011.
29. A. Taherkhani, A. Korhonen, and L. Malmi. Categorizing variations of student-implemented sorting algorithms. *Computer Science Education*, 22(2):109–138, 2012.
30. R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, 1988.