Automatic Algorithm Recognition Based on Programming Schemas

Ahmad Taherkhani

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

Abstract. A method for recognizing algorithms by detecting algorithmic schemas is presented. The method uses the findings of the studies on programming schemas, according to which experts develop schemas, high-level cognitive constructs that abstract knowledge of programming structures, and use them in comprehending and solving similar problems that differ in lower level details. We introduce a set of schemas for sorting algorithms that consists of loops, their nesting relationship, beacon-like algorithm-specific features and operations, etc., and use these abstract concepts to recognize implementations of sorting algorithms.
We have developed a prototype for detecting schemas and conducted an experiment to evaluate the performance of the presented method on sorting algorithms and their variations implemented in Java. The tested implementations are recognized with the average accuracy of 88,3%. This is a promising result that shows the applicability of the method in context and level of students' implementations. By identifying the algorithm-specific code from the given program, the schema detection method improves our previous method for automatic algorithm recognition.

Keywords: POP-II.B. algorithm recognition, schema recognition, program comprehension, white-box analysis; POP-I.B. roles of variables

1 Introduction

Students may have many misconceptions related to data structures and algorithms (see, e.g., [16]). In a study reported in [22], we analyzed freshmen students' sorting algorithm implementations and discovered some of these misconceptions related to sorting algorithms. As an example, students use unnecessary swap operations in their Insertion and Selection sort implementations.

Automatic assessment systems that are based on black-box testing cannot detect such misconceptions. They execute the given algorithm using a specified input and compare the output with the output of a model solution. Since problematic students' solutions may well produce the acceptable output, the bad quality of these solutions will not be revealed. White-box analysis can address these problematic and inefficient implementations. Once detected, it is beneficial to give personal feedback on these impractical and poor solutions and make students rethink the algorithm in question. Moreover, from a teacher's point of view, when students are requited to implement a specific algorithm, white-box analysis is able to verify the usage of that specific algorithm, allowing the teacher to focus on the cases that do not implement the required algorithm, instead of checking all the implementations.

In our previous work, we have introduced a method for algorithm recognition [20], and validated it with authentic students' implementations [22]. That previously introduced method is not able to identify algorithm-specific code from application data processing code and processes all the given code as relevant algorithmic code. This may reduce the accuracy of recognition. In addition, in order to identify problematic solutions in students' implementations, detecting algorithm-specific code is necessary. The main contribution of this paper is to address this deficiency by introducing a new method based on static analysis for detecting algorithmic schemas. This schema detection method enhances our previous method in terms of white-box testing. With algorithm-specific code detected, further analysis can be continued only on this algorithmic code, leaving the non-relevant application-specific code out of the process. The schema detection method is inspired by the work on programming schemas, such as [2] and [18]. We have implemented the method in a prototype instrument called Aari (an Automatic Algorithm Recognition Instrument), applied it to Java implementations of five basic sorting algorithms (Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort) and their variations, and conducted an experiment to evaluate the performance of the method.

Our main research questions are:

1. *How can we develop an automatic algorithmic schema recognition method for recognizing implementations of sorting algorithms that is based on the programming schema research?*
2. *How accurately this automatic algorithmic schema recognition method would perform?*

The rest of the paper is organized as follows. We discuss program comprehension and programming schema models, as well as automatic program comprehension techniques briefly in Section 2. In Section 3, we explain our method in general and present its theoretical background. The method is applied to sorting algorithms, which is discussed in Section 4. Section 5 describes the data used in the experiment. We present the results of the experiment on sorting algorithms in Section 6, followed by a discussion in Section 7. Finally, we conclude in Section 8 that the results of our experiment on sorting algorithms show that it is possible to develop automatic schema recognition methods by using the programming schema research. We also conclude that the presented schema recognition method performs well with standard sorting algorithms and their inefficient variations. We discuss some directions for future work in this section as well.

2 Related work

Program Comprehension (PC) is a process in which a human builds his or her own mental representation of a program. Understanding programs involves different elements including *external representation* (how the target program is represented to the programmer), *assimilation process* (the strategy, such as *top-down*, *bottom-up* or *integrated*, used for building a mental representation of the target program using the knowledge base and the given representation of the program) and *cognitive structure* (programmer's knowledge base including the prior programming knowledge and the domain knowledge related to the target program, and the mental representation that the programmer has built). Several models for PC and the concept of programming schemas has been introduced, including [2], [10] and [18]. In [15], an overview of the PC models is presented and their implications for teaching and learning introductory programming is discussed. We will get back to programming schemas in next section where we discuss our schema detection method.

Many attempts have been made to develop methods and tools to automate PC. The objective is to help programmers in software engineering related tasks or achieve other goals related to programming. The characteristics that influence cognitive strategies used by programmers, influence the requirements for supporting tools as well [19]. For example, the top-down and bottom-up strategies are reflected in a supporting tool so that "The top-down process requires browsing from high-level abstractions or concepts to lower level details, taking advantage of beacons", and, "Bottom-up comprehension requires following control-flow and data-flow links as well as cross-referencing navigation between elements in the program and situation models defined by Pennington." [19].

Most automatic PC techniques use a knowledge base, where the basic idea is to store stereotypical pieces of code, often called *plans*, in a knowledge base and match the target program against these pieces. Since the functionalities of the plans in the knowledge base are known, the functionality of the target program can be discovered if a match is found between the target program and the plans. As with the assimilation process in PC models, there are three main techniques to perform matching: top-down, bottom-up, and hybrid technique. Top-down techniques (see, e.g., [8]) use the goal of the program to select the right plans from the knowledge base. This speeds up the process of selecting the plans and makes the matching more effective. However, the main disadvantage of these techniques is that they need the specification of the target program, which is not necessarily available in real-life, especially in case of legacy systems. In bottom-up approach (see for example [7]), the matching starts at the low-level of abstraction, with statements and small plans, and proceeds toward the higher-level of abstraction to discover the goal of the target program. The main concern with bottom-up techniques is efficiency. As the statement can be part of several different plans and the same plan can be part of different bigger plans, the process of matching statements and plans can become ineffective for real-life programs with thousands of lines of code. Hybrid techniques use the combination of the two techniques. For example, in [11], plans are first recognized in a bottom-up manner. After this, general plans are suggested by the system to be matched against the program in a top-down manner. These general plans are proposed by using a well-organized plan library, where each plan is identified by an index and specialization and implication links to the other plans. By using the indexing facility, the system is able to quickly associate a piece of the source code with a plan in the knowledge base. The automatic PC tools do not present a clear evaluation on their performance, thus it is difficult to say how accurately they work.

3 Algorithmic schemas

This section discusses the theoretical foundation for schema detection technique in general. In the next section, we will discuss how the technique can be applied to sorting algorithms.

Theoretical background

In programming, schemas can be defined as formalized knowledge structures [4]. Schemas abstract detailed knowledge of programming structures and help experts in understanding and solving the similar tasks that differ in lower level of abstraction and implementation details. Several studies on schemas, on the process of schema creation and on the differences between experts and novices in programming activities that are related to possession of schemas, such as program comprehension, are reported. Studies of Soloway and Ehrlich are among most important reports on the topic [4]. Soloway and Ehrlich define schemas (which they call plans) as "generic program fragments that represent stereotypic action sequences in programming" [18]. They used *plan-like* and *unplan-like* programs (the former meaning programs that use stereotypical plans and conform to rules of programming discourse, as opposed to the latter that do not) to study how the performance of experts and novices differs when working with these two types of programs. The authors show that while plan-likeness or unplan-likeness of the given program does not have dramatical effect on the performance of the novices (because they do not possess programming plans and discourse rules yet), the experts perform significantly better with plan-like programs, since they have developed programming plans and conventions which they use when solving the tasks. Soloway and Ehrlich also emphasize the importance of *critical lines*, which are highly informative and representative lines that are strong indications of existing a plan, and can be utilized in recognizing and verifying the plan.

Beacons are important elements of several program comprehension models that by providing a link between source code and the process of verifying the hypotheses driven from the source code, help programmers to accept or reject their hypotheses about the code. Beacons are statements that indicate the existence of a particular structure or operation in code and thus play an important role in understanding programs by experts [2]. As an example, the existence of a swap operation, especially inside a pair of loops, indicates sorting of array elements [2].

We believe that the results and ideas of the studies on schemas can be utilized in automatic recognition of algorithmic schemas. In the same way that experts develop abstracted schema knowledge and use it to recognize and solve the similar programming problems that differ in lower level details, an automatic schema recognition system could use abstracted stereotypical implementations of algorithms as a knowledge base to recognize various implementation instances that differ in implementation details. For each supported algorithm, the system must be provided with schemas, subschemas, beacons and critical lines specific to that algorithm. These can include use of loops, specific operations (such as swap), recursion, *roles of variables* (discussed below) and other algorithm-specific beacons and critical lines. The overall recognition is resolved by putting these separately recognized elements together and examining their relationships in terms of nesting, execution order, etc., just like experts do. As we will explain in the next section, for a Quicksort algorithm implementation, as an example, this may include pivot selection, partitioning, swap operation and recursive call. The given program is analyzed in order to detect these schemas and their relationships, and the detected schemas are matched to the schemas from the knowledge base of the system. Thus, the method can be considered as a bottom-up method.

In order to distinguish between two or more algorithms with similar algorithmic schemas, algorithm-specific features are used. Source code can be analyzed to discover patterns that implement features specific to the way each algorithm works. These patterns can be employed as beacons to recognize borderline cases. As an example, adjacency of the two compared elements in Bubble sort is a feature specific to Bubble sort that does not occur in the other sorting algorithms and their variations discussed in this paper. This feature can be used to differentiate between Bubble sort and the other algorithms that appear to have the similar algorithmic schemas, as we will discuss in the next section. A useful tool that can be used as beacons in schema recognition is roles of variables.

Roles of variables

Roles of variables [12] are specific patterns how variables are used in source code and how their values are updated during program execution. Roles were originally introduced to help students learn programming. However, the concept can be used to analyze programs with different purposes. We have used roles of variables in algorithm recognition in our previous work [20,21]. Roles can be analyzed automatically using data flow analysis and machine learning techniques (see [1] and [6]).

As described in [13], there are 11 different roles recognized at present. More detailed discussion of roles of variables is beyond the scope of this paper. See the Roles of Variables Home Page[1] for a more comprehensive information.

Roles of variables, as patterns that repeatedly occur in code and provide links to program structures, help students in processing program information and forming schemas [14]. From the algorithmic schema detection point of view, variables used in an implementation of an algorithm have roles that are specific to the way that algorithm works. For example, for sorting algorithms, *most-wanted holder* (a variable that holds the most desirable value that is found so far when going through a set of values) implies Selection sort. As another example, a *temporary* role (a variable that holds a value for a short period of time) indicates a swap operation. Thus, automatic detection of roles in an implementation of an algorithm can facilitate recognition of the corresponding algorithmic schema and help to distinguish between borderline cases.

4 Detecting algorithmic schemas for sorting algorithms

In this section we present a set of algorithmic schemas for sorting algorithms that we formulated by analyzing the implementations of the data set discussed in Section 5.

In our previous study [22], we classified students' implementations of sorting algorithms in order to see what kind of variations of well-known sorting algorithms students use. Our goal was

[1] `http://www.cs.joensuu.fi/~saja/var_roles/`

to discover common problematic implementations in students' submissions and develop methods and tools to give automatic feedback on these problematic solutions in term on white-box testing. We noticed that students have many misconceptions related to sorting algorithms. The two most common misconceptions relate to Insertion sort and Selection sort. Many student use unnecessary swap operations in their implementations of these two sorting algorithms, which make the solution inefficient. More specifically, with regard to Insertion sort, many students' implementations exchange the elements in the inner loop instead of shifting them (even some textbooks introduce this version of Insertion sort; see, e.g., [3] and [17]). We call these versions **Insertion sort WS** (Insertion sort With Swap). In the case of Selection sort, many students' implementations exchange in the inner loop each element smaller than the element pointed by the loop counter of the outer loop, instead of storing the position of the smallest (or largest) element found in a single pass and exchanging it in the outer loop. We call these variations **Selection sort WILS** (Selection sort With Inner Loop Swap). Figure 1 shows a typical implementation of these variations of Selection sort (Fig. 1a) as well as a typical implementation of a standard Selection sort (Fig. 1b). We concluded in [22], that in order to recognize these kind of variations in students' solutions and give instant automatic feedback to students, we need to develop a schema detection method. Therefore, to demonstrate how the problematic solutions in students' work can be automatically recognized, in addition to the standard basic algorithms, Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort, we cover Insertion sort WS and Selection sort WILS variations in this paper as well.

```
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. 1a)

```
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. 1b)

Fig. 1. An example of a typical implementation of an inefficient variation of Selection sort that uses swap in the inner loop instead of the outer loop, is shown in Fig. 1a. Fig. 1b shows an implementation of a standard Selection sort

In the following, we present a brief implementational definition for the standard basic algorithms listed above. This will help to understand the algorithmic schemas of these algorithms, and the technique used for detecting these schemas.

4.1 Bubble sort

Implementation of a standard Bubble sort algorithm consists of two nested loops and a swap operation which is done in the inner loop. A basic feature of Bubble sort outlined by textbooks and articles is that the two elements of the sortable list compared in each pass are adjacent. We will use this feature to differentiate between the algorithmic schema of Bubble sort and the other non-recursive borderline cases.

4.2 Insertion sort

Implementation of a standard Insertion sort includes two nested loops and a shift operation performed in the inner loop. An essential feature that we use to differentiate between the algorithmic schema of Insertion sort and the other non-recursive borderline cases is that in Insertion sort (as well as in Insertion sort WS), the elements of the sortable list are processed successively; the second element is compared with the first element, the third element is compared with the second and the first one, and so forth (see, for example, [5,9]).

4.3 Selection sort

Implementation of a standard Selection sort also includes two nested loops, where, as discussed above for the differences between standard Selection sort and Selection sort WILS, the position of the smallest (or largest) element is stored in each pass in the inner loop, and in the outer loop, this element is exchanged with the element pointed by the loop counter of the outer loop. This smallest (or largest) element found in the inner loop has the most-wanted holder role.

4.4 Quicksort

We divide the implementation of a standard Quicksort into two parts: the overall algorithmic schema and the partition algorithmic schema. The overall algorithmic schema includes selecting a pivot, partitioning the sortable list and calling the method recursively for both partitions. For implementing the partition part, we discerned four patterns from the analyzed implementations that will be discussed below.

4.5 Mergesort

Like Quicksort, we divide the implementation of a standard Mergesort into two parts: the overall algorithmic schema and the merging algorithmic schema. The overall algorithmic schema consists of dividing the sortable list into two halves, sorting each half recursively and merging the sorted halves into the final sorted list. Based on the analyzed data, the implementations of merging part varies greatly compared with, for example, the partition part of Quicksort implementation. We will get back to this below.

Figures 2 and 3 summarize the implementational definitions of the aforementioned sorting algorithms into the corresponding algorithmic schemas. The indentations illustrate the nesting relationship between the loops and blocks.

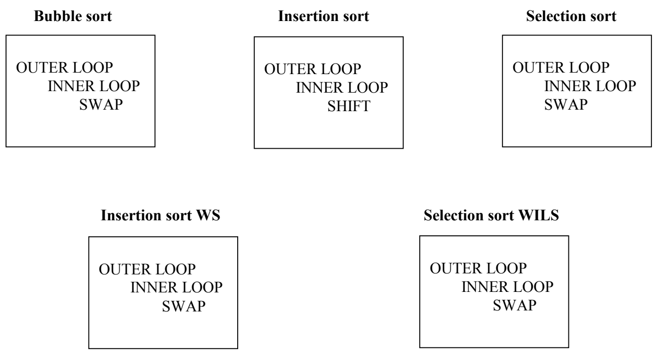

Fig. 2. Algorithmic schemas for the non-recursive sorting algorithms and their variations

As can be seen from Figure 2, the algorithmic schemas for Bubble sort, Insertion sort and Selection sort are different enough to be distinguishable. Thus, there is no need to use other beacon-like features of these algorithms (such as most-wanted holder role in Selection sort) to differentiate between them. However, Bubble sort, Insertion sort WS and Selection sort WILS have exactly the same algorithmic schemas. Therefore, to distinguish between these algorithms, we need to use their other beacon-like features. As discussed above, in Insertion sort, the elements of the given list are handled in a successive manner. This implies that in implementations

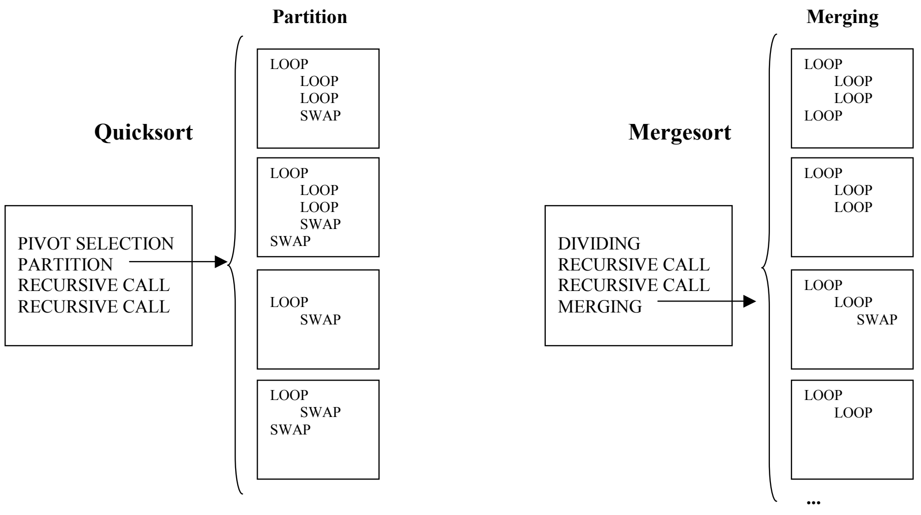

Fig. 3. Algorithmic schema for Quicksort and Mergesort. Other schemas for merging in Mergesort are also possible

of Insertion sort, the inner loop counter is initialized to the value of the outer loop counter (this is a feature that we have successfully used in our previous work to distinguish between Insertion sort and Bubble sort, as explained in [20,21]). Thus, if the given implementation is recognized as a Bubble sort and has this feature, its type is changed to Insertion sort WS. In addition, as outlined above, an essential feature of Bubble sort is that the two compared elements in the inner loop in each pass are adjacent. This is not the case in Selection sort WILS (see Figure 1). Therefore, if the given implementation is recognized as a Bubble sort and does not have this feature, it is labeled by the type Selection sort WILS.

The algorithmic schemas of the recursive algorithms presented in Figure 3 are self-explanatory. For detecting the pivot selection in Quicksort, different pivot selection strategies are considered, including selecting the pivot item from the left or right end of the sortable list, from the middle of it or using the median of the first, middle and last elements. The candidate recursive method should have two tail recursive calls. In addition, the detected partition part should be executed after the pivot selection and before the recursive calls (either in-lined in the recursive method or in its own method called from the recursive method) as shown in Figure 3. As illustrated in the figure, there are mainly two patterns of using loops in implementations of the partition part, each of which using different numbers of swap operations. Some implementations goes through the elements of the given list in a single loop to find the elements in wrong order relative to the pivot and exchange them, while other implementations uses three loops to go through the list from both side to find and exchange the elements in wrong positions.

With regard to Mergesort, the candidate recursive method has two non-tail recursive calls, the dividing statement is executed before these recursive calls and the merging part after the recursive calls. The algorithmic schemas of the merging part are not as clear as the partition part of Quicksort implementations. In addition to those schemas illustrated in Figure 3, other patterns may also be used in implementations of the merging, including different numbers of consecutive loops. This fact is demonstrated using three dots in the bottom of the merging schema in Figure 3. This make detecting Mergesort schema more difficult than the schemas of the other sorting algorithms. As we will discuss in Section 6, the results of our experiment suggest the same as well.

We gave these schemas as a knowledge base to Aari and implemented the discussed mechanism into it for recognizing the algorithmic schemas of sorting algorithms written in Java language. We also conducted an experiment to evaluate the accuracy of this technique.

5 Experiment

We conducted an experiment to evaluate how accurately the algorithmic schemas of the five basic sorting algorithms (Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort) and the two variations of Insertion sort and Selection sort can be detected using the schema detection technique introduces in Section 4.

Data collection

We used two sets of data in this experiment. The first data set includes the implementations that we used in our previous studies [20] and [21], to build the decision trees discussed in those studies for recognizing sorting algorithms, and to evaluate the performance of those trees. There are a total of 287 implementations in this data set, from which 209 include the implementations of the five aforementioned basic sorting algorithms and the rest 78 include the implementations of other sorting algorithms, like Heapsort, Shellsort and hybrid implementations of, for example, Quicksort-Insertion sort, as well as implementations of algorithms from other fields, such as binary search, etc. The implementations of this data set were collected from various sources including textbooks, the Web and students' submissions (a few Insertion sort and Quicksort implementations only). For the rest of the paper, we call this data set"Multi-source" data set. In addition to the corresponding algorithmic schemas, all the implementations include a Java main method, where an array of integers is initialized and the sorting method is called or the algorithmic code is written in-lined within the main method. Moreover, some of the implementations collected from the Web included some extra code for, for example, testing the algorithm implementation and printing the items of the sorted array.

The second data set, which we call "Submissions" data set for the rest of the paper, was collected from the students' submissions in a first year Data Structure and Algorithms course. We used this data set in [22] to examine what kind of variations of well-known sorting algorithms freshmen students implement and how these implementations can be classified. We also tested how well Aari can automatically recognize these algorithms (see [22] for the results). These submissions were collected in two phases, for two separate sorting algorithm assignments. The first assignment was given at the very beginning of the course before student were formally instructed, and the second was given after sorting algorithms were taught. All the submissions were tested automatically by an automatic assessment system that checked their correctness in terms of block-box testing. Only the submissions that passed these automatic tests are used in the present experiment as follows. In the first phase, we collected 112 submissions and in the second phase 80 submissions. From these 192 submissions, 159 included the implementations of the five aforementioned sorting algorithms and the two variations of Insertion sort and Selection sort. We will come back to these figures in Section 6. For the first assignment, the students were provided with a template of about 100 lines of code that contained a Java main method, a method for generating an array with random integers for testing the solution (this method used two nested loops and several if-else statements) and a method for printing the items of the array before and after sorting. The students implemented their solution in a method of a specified signature. For the second assignment, generating the array of random integers to be sorted and printing its items were performed on the server side. However, some students included the template into their submissions in the second phase as well.

Data preparation

We added a main method to those implementations of the Multi-source data set that did not include one. In addition, we added an array of integers to be able to examine the correctness of the implementation (see [21] and [20] for more information). With regard to the Submissions data set, a few students' implementations used enhanced for-loop introduced in Java 5.0 for traversing over the elements of a collection or an array. These few enhanced for-loops were replaced with the ordinary for-loops, because Aari does not support them at its current state.

6 Results

We tested our schema detection technique on a data set consisting of overall 479 implementations (287 from the Multi-source data set and 192 from the Submissions data set). From these implementations, 368 (209 from the Multi-source data set and 159 from the Submissions data set) included implementations of one of the five basic sorting algorithms (Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort) or the two variations of Insertion sort and Selection sort, as discussed in Section 5. The rest 111 implementations included implementations of other algorithms and were tested to see whether the algorithmic schemas of these implementations can falsely be detected as one of those algorithmic schemas of the target set. We will get back to the results for these implementations shortly.

The total number of each tested algorithm of the target set and the number of the correctly detected algorithmic schemas are depicted in Figure 4. For these algorithms, the algorithmic schemas are detected with the average accuracy of 88,3 percent. All the algorithmic schemas of Selection sort WILS are detected correctly and the algorithmic schemas of Bubble sort, Insertion sort and Quicksort are detected with accuracy of 90 percent or higher. The algorithmic shemas of the Mergesort implementations are detected with lowest accuracy. Table 1 shows the source of the tested implementations and summarizes the results for the algorithms of the target set.

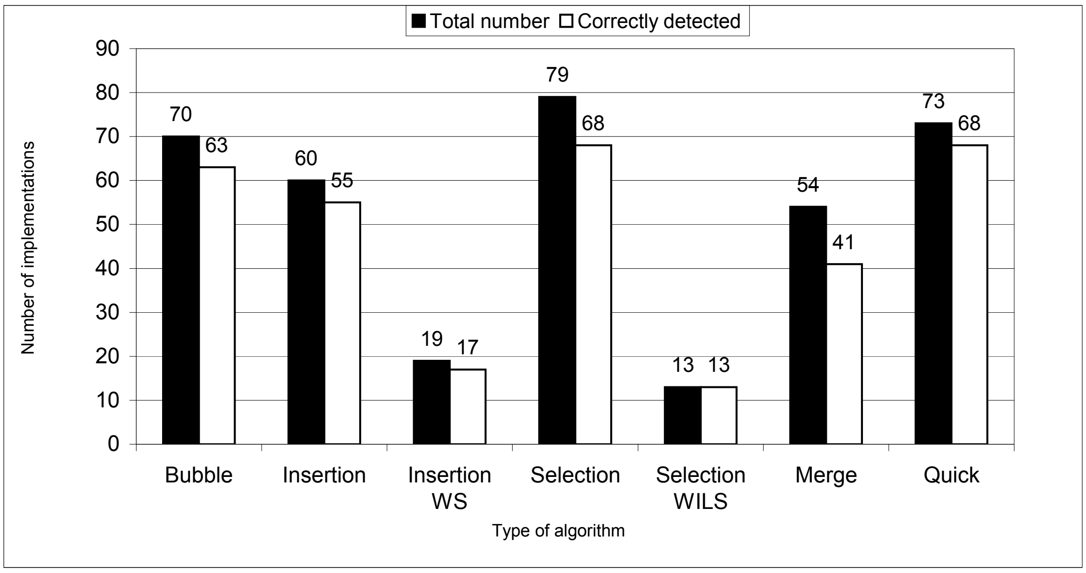

Fig. 4. The total number and the number of correctly detected sorting algorithm implementations

As discussed above, we also tested 111 program that did not include implementations of the five “pure” algorithms listed in Table 1. These tests revealed how many program could be falsely detected as having one of the specified algorithmic schemas. From these 111 implementations, 78 was from the Multi-source data set and 33 from the Submissions data set. The Multi-source implementations included mainly different sorting algorithms (Heapsort, Shellsort, different types of hybrid sorts, enhanced versions of the five algorithm listed in Table 1, etc.), algorithms from other fields such as binary search, as well as some application data (interface code, testing code, etc). The Submissions data set consisted of programs that included different type of sorting algorithms. The major part of these these implementations follow the idea of one of the five aforementioned standard sorting algorithms. However, these implementations differ

Table 1. The number and percent of the correctly detected algorithmic schemas for each type of sorting algorithm. Column Total shows the total number of each algorithm and columns Multi-source and Submissions indicate the source of the implementations (see Section 5)

Algorithm	Total	Detected(%)	Not detected(%)	Multi-source	Submissions
Bubble sort	70	63 (90,0)	7 (10,0)	41	29
Insertion sort	60	55 (91,7)	5 (8,3)	43	17
Insertion sort WS	19	17 (89,5)	2 (10,5)	9	10
Selection sort	79	68 (86,1)	11 (13,9)	43	36
Selection sort WILS	13	13 (100)	0 (0)	0	13
Mergesort	54	41 (75,9)	13 (24,1)	34	20
Quicksort	73	68 (93,2)	5 (6,8)	39	34
Total	368	325 (88,3)	43 (11,7)	209	159

from the standard algorithms in a way that they could not be regarded as one of these standard algorithms. For example, they use auxiliary arrays and do not work in-place and thus cannot be considered in the same category as their in-place corresponding algorithm, they use excessive loops, etc. These implementations also included Heapsort and Shellsort, as well as some less-known sorting algorithms such as Gnome sort, Bozo sort, etc. We had considered all of these 111 implementations as "Others" in our previous works as well [20,22]. From these, 10 implementations (i.e., 10 percent) were falsely detected as including an algorithmic schema of one of the five sorting algorithms. Specifically, from the Submissions data set, 1 Quicksort-Selection sort hybrid implementation was detected as a Selection sort implementation. In the case of the 9 falsely detected schemas of the Multi-source data set implementations, 3 Cocktail sort implementations (also known as Bidirectional Bubble sort which improves the standard Bubble sort by using an additional loop to sort in both directions on each single pass through the sortable array) were detected as Bubble sort implementations, 2 Heapsort implementations were detected as Selection sort implementations and 4 hybrid implementations of Quicksort-Insertion sort and Quicksort-Bubble sort were detected as Quicksort implementations.

7 Discussion

Experts use schemas to recognize various versions of the essentially same problems that differ in lower level details. Grouping the similar problems of lower level abstraction into higher-level schemas and developing these schemas is what turns a novice into an expert when dealing with program comprehension related tasks. During this process of becoming an expert, programmers also become familiar with beacons, that is, those features specific to programs that guide programmers in recognizing and comprehending them.

We have adapted the results from programming schema research and demonstrated the idea of applying them in automatic schema recognition in the case of sorting algorithms. We have introduced a set of stereotypical abstracted schemas of Figures 2 and 3 to Aari and tested how it performs with the implementations that differ in lower level details. Aari also makes use of beacons in the process, for example, to distinguish between the algorithms with the similar schemas. The results show that the method can recognize the algorithmic schemas for the five sorting algorithms and the two of their variations discussed in this paper with the average accuracy of 88,3 percent. The algorithmic schemas of Mergesort are not detected as accurately as the algorithmic schemas of the other algorithms of the target set, because of the large variations in the implementations of merging part, especially in the students' implementations (from the 13 not detected schemas, 12 were from the students' implementations). From the implementations of the algorithms that are not part of the target set, only 10 percent are falsely recognized as one belonging to the target set. This suggests that the presented algorithmic schemas describe the algorithms of the target set in a sufficient level of detail and specificity.

The sorting algorithms used in the experiment are suitable for testing the schema detection method, since these algorithms have both clearly different schemas (e.g., Bubble sort and Quicksort) and very similar schemas (e.g., Bubble sort, Insertion sort WS and Selection sort WILS). Being able to differentiate between these algorithms with the same schemas shows the applicability of beacon-like features of the algorithms. For example, adjacency of the two compared elements of a list in the inner loop is a beacon-like feature specific to Bubble sort that differentiates it from Insertion sort WS and Selection sort WILS. We believe that using the schema detection method, it is possible to recognize basic algorithms of other fields as well. We could use other beacon-like features, such as roles of variables, for recognizing a more comprehensive set of algorithms from other fields. However, we need to evaluate this by empirical tests.

Although real-world applications and programming projects use existing standard libraries to implement sorting and many other functionalities and topics covered in a data structures and analysis courses, students should learn the core topics and key algorithms in practice. In addition to lectures, learning involves programming assignments for implementing these algorithms. These programming assignments should be checked and personal feedback and suggestions for improvement should be given to students as an important part of learning. Our goal is to develop methods and tools to automatically analyze students' implementations in order to check whether they have implemented the required algorithm, and to provide feedback on poor solutions. We have shown that the method performs well in this context. We cannot claim that this technique scales up to analyze complex systems.

By allowing us to locate algorithmic-code from other application-specific data in source code and find subschemas from the given implementation of an algorithm, schema recognition method improves our previous method for white-box analysis. For example, schema recognition method is needed in order to find unnecessary swap operations and their location from students' implementations in the case of Insertion sort WS and Selection sort WILS, and to distinguish between the implementations of these variations and other standard algorithms.

The correctness of students' solutions should also be checked and only error-free solutions be selected for white-box analysis. Automatic assessment systems that perform black-box testing could be used for that, as we did in [22].

8 Conclusion and future work

We have discussed a method for schema recognition and outlined its connection with the findings of studies on program comprehension and programming schemas. We have implemented the method into a previously developed prototype, Aari system, and demonstrated that the method performs well with sorting algorithms. By enabling us to select only algorithm-related code for further analysis, the schema recognition method considerably enhances our previous method of algorithm recognition.

Using our previous method for algorithm recognition and the schema recognition method discussed in this paper, Aari can assist instructors in checking students' work in mass courses. In addition to checking the correctness of students' solutions (which can be done using black-box testing tools), an instructor often needs to know whether the solution implements the required algorithm. Moreover, it is beneficial to detect inefficient and poor solutions such as using unnecessary swaps or bad pivot selection strategy, and give feedback to students to fix the problems. These are just some examples on how systems like Aari that perform white-box analysis can help instructors in their work by assessing the implemented algorithm, and help students in learning by making them rethink their solutions.

In our future work, we will integrate the schema recognition method into our previous method [20] to build a decision tree that can classify students' implementations and their variations. We will use the schema recognition method to detect subschemas and beacons from the given programs, and use them as learning data to train Aari so that it can classify unseen algorithm instances and their variations based on these subschemas and beacons. Future work will also include extension of the schema recognition method to cover other fields of algorithms.

9 Acknowledgments

I thank Lauri Malmi and Ari Korhonen for their comments.

References

1. 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.
2. R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.
3. N. Dale, D. Joyce, and C. Weems. *Object-Oriented Data Structures Using Java.* Jones and Bartlett Publishers, second edition, 2002.
4. 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.
5. E. H. Friend. Sorting on electronic computer systems. *Journal of the ACM*, 3(3):134–168, 1956.
6. P. Gerdt and J. Sajaniemi. 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*, pages 178–182. ACM, New York, NY, USA, 2006.
7. M. Harandi and J. Ning. Knowledge-based program analysis. *Software IEEE*, 7(4):74–81, 1990.
8. 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.
9. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, New Jersey, USA, second edition, 1998.
10. N. Pennington. Comprehension strategies in programming. *Empirical studies of programmers: second workshop*, pages 100–113, 1987.
11. A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.
12. 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.
13. J. Sajaniemi, M. Ben-Ari, P. Byckling, P. Gerdt, and Y. Kulikova. Roles of variables in three programming paradigms. *Computer Science Education*, 16(4):261–279, 2006.
14. J. Sajaniemi and M. Kuittinen. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1):59–82, 2005.
15. C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. H. Paterson. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports on Innovation and technology in computer science education - ITiCSE-WGR '10 (Bilkent, Ankara, Turkey, June 26–30, 2010)*, pages 65–86. ACM New York, NY, USA, 2010.
16. O. Seppälä, L. Malmi, and A. Korhonen. Observations on student misconceptions – a case study of the build-heap algorithm. *Computer Science Education*, 16(3):241–255, September 2006.
17. C. A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis.* Prentice Hall Inc, New Jersey, USA, 1998.
18. E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.
19. M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.
20. A. Taherkhani. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *The Computer Journal; doi: 10.1093/comjnl/bxr025*, 2011.
21. 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; doi: 10.1093/comjnl/bxq049*, 2010.
22. A. Taherkhani, A. Korhonen, and L. Malmi. Recognizing students' sorting algorithm implementations in a data structures and algorithms course. *ACM Transactions on Computing Education (TOCE)*, submitted.