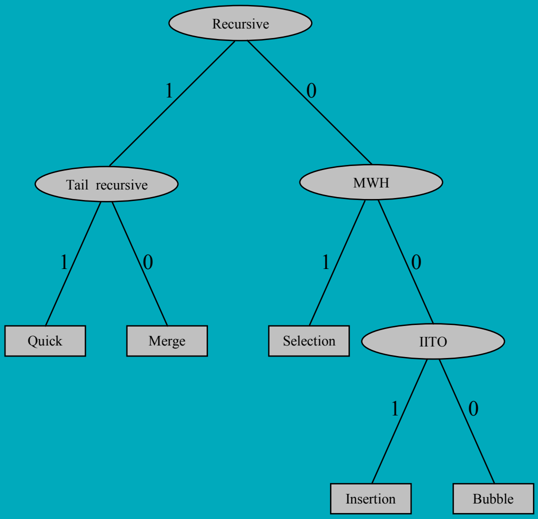
Department of Computer Science and Engineering

Automatic Algorithm Recognition Based on Programming Schemas and Beacons

A Supervised Machine Learning Classification Approach

Ahmad Taherkhani



Aalto University

DOCTORAL
DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 17/2013

Automatic Algorithm Recognition Based on Programming Schemas and Beacons

A Supervised Machine Learning Classification Approach

Ahmad Taherkhani

A doctoral dissertation completed for the degree of Doctor of Science (Technology) (Doctor of Philosophy) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on the 8th of March 2013 at 12 noon.

Aalto University
School of Science
Department of Computer Science and Engineering
Learning + Technology Group

Supervising professor
Professor Lauri Malmi

Thesis advisor
D. Sc. (Tech) Ari Korhonen

Preliminary examiners
Professor Jorma Sajaniemi, University of Eastern Finland, Finland
Dr. Colin Johnson, University of Kent, United Kingdom

Opponent
Professor Tapio Salakoski, University of Turku, Finland

Aalto University publication series
DOCTORAL DISSERTATIONS 17/2013

ISBN 978-952-60-4989-2 (printed)
ISBN 978-952-60-4990-8 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-4990-8

Unigrafia Oy
Helsinki 2013

Finland

Publication orders (printed book):
The dissertation can be read at http://lib.tkk.fi/Diss/

NORDIC ECOLABEL
441 697
Printed matter

A! Aalto University

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Abstract

Author
Ahmad Taherkhani

Name of the doctoral dissertation
Automatic Algorithm Recognition Based on Programming Schemas
and Beacons: A Supervised Machine Learning Classification Approach

Publisher Aalto University School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 17/2013

Field of research Software Systems

Manuscript submitted 11 September 2012 **Date of the defence** 8 March 2013

Permission to publish granted (date) 18 December 2012 **Language** English

□ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

In this thesis, we present techniques to recognize basic algorithms covered in computer science education from source code. The techniques use various software metrics, language constructs and other characteristics of source code, as well as the concept of schemas and beacons from program comprehension models. Schemas are high level programming knowledge with detailed knowledge abstracted out. Beacons are statements that imply specific structures in a program. Moreover, roles of variables constitute an important part of the techniques. Roles are concepts that describe the behavior and usage of variables in a program. They have originally been introduced to help novices learn programming.

We discuss two methods for algorithm recognition. The first one is a classification method based on a supervised machine learning technique. It uses the vectors of characteristics and beacons automatically computed from the algorithm implementations of a training set to learn what characteristics and beacons can best describe each algorithm. Based on these observed instance-class pairs, the system assigns a class to each new input algorithm implementation according to its characteristics and beacons. We use the C4.5 algorithm to generate a decision tree that performs the task. In the second method, the schema detection method, algorithms are defined as schemas that exist in the knowledge base of the system. To identify an algorithm, the method searches the source code to detect schemas that correspond to those predefined schemas. Moreover, we present a method that combines these two methods: it first applies the schema detection method to extract algorithmic schemas from the given program and then proceeds to the classification method applied to the schema parts only. This enhances the reliability of the classification method, as the characteristics and beacons are computed only from the algorithm implementation code, instead of the whole given program.

We discuss several empirical studies conducted to evaluate the performance of the methods. Some results are as follows: evaluated by leave-one-out cross-validation, the estimated classification accuracy for sorting algorithms is 98,1%, for searching, heap, basic tree traversal and graph algorithms 97,3% and for the combined method (on sorting algorithms and their variations from real student submissions) 97,0%. For the schema detection method, the accuracy is 88,3% and 94,1%, respectively.

In addition, we present a study for categorizing student-implemented sorting algorithms and their variations in order to find problematic solutions that would allow us to give feedback on them. We also explain how these variations can be automatically recognized.

Keywords algorithm recognition, schema detection, beacons, roles of variables, program comprehension, automated assessment

ISBN (printed) 978-952-60-4989-2 **ISBN (pdf)** 978-952-60-4990-8

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2013

Pages 254 **urn** http://urn.fi/URN:ISBN:978-952-60-4990-8

To the memory of
my brother and my father-in-law
who both were a big part of my life.

Preface

Back in 2007, after working in industry for several years, I contacted Professor Lauri Malmi, the supervisor of this dissertation, for possibilities on working in his research group to complete my master's thesis. He hired me to a perfect position which was funded by the Academy of Finland. The project was much larger than a master's thesis and after completing my thesis, he offered me an opportunity to continue working on it to pursue my doctoral degree. Having always desired to do my PhD and being interested in the topic, I asked for a leave of absence from my work at Accenture and used the opportunity. It was Lauri who made it all possible: he gave me an opportunity to become a part of the computing education community, allowed me to focus almost full-time on my research and provided me an excellent guidance throughout the project. So thank you Lauri, I really appreciate all of these.

Doctor Ari Korhonen provided great ideas and insightful suggestions which are reflected throughout this thesis. His friendly discussions, can-do attitude and critical thinking always helped me forward. I thank him also for showing me how to be a good instructor.

I thank all the members of our research group, the Learning + Technology Group (LeTech), for providing a friendly environment in which to work and for their comments during our weekly meetings. I especially thank Doctor Jan Lönnberg for his help.

I am grateful to the pre-examiners, Professor Jorma Sajaniemi and Doctor Colin Johnson, for taking the time to check my thesis and for their valuable observations and suggestions. Sajaniemi also read my Licentiate's thesis and one of my journal publications and provided constructive feedback. It is also an honor to have Professor Tapio Salakoski as my opponent.

I would also like to thank the participants of the international conferences who provided comments and feedback on my presentations. The same goes for the international working group that I had the chance to participate in.

I thank my parents for giving me an opportunity for an education and an abiding respect for learning.

Last, and most importantly, I would like to thank my kids, Ava and Arad, for being so special and my wife Elmira for her enormous patience and support and continuing encouragement in whatever I decide to do.

Helsinki, December 28, 2012,

Ahmad Taherkhani

Contents

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Ahmad Taherkhani, Ari Korhonen and Lauri Malmi. Recognizing Algorithms Using Language Constructs, Software Metrics and Roles of Variables: An Experiment with Sorting Algorithms. *The Computer Journal*, Volume 54 Issue 7, pages 1049–1066, June 2011.

II Ahmad Taherkhani. Using Decision Tree Classifiers in Source Code Analysis to Recognize Algorithms: An Experiment with Sorting Algorithms. *The Computer Journal*, Volume 54 Issue 11, pages 1845–1860, October 2011.

III Ahmad Taherkhani, Ari Korhonen and Lauri Malmi. Categorizing Variations of Student-Implemented Sorting Algorithms. *Computer Science Education*, Volume 22 Issue 2, pages 109–138, June 2012.

IV Ahmad Taherkhani, Ari Korhonen and Lauri 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*, Tahko, Finland, 10 pages, 15–18 November 2012, to appear.

V Ahmad 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, 12 pages, 6–8 September 2011.

VI Ahmad Taherkhani and Lauri Malmi. Beacon- and Schema-Based Method for Recognizing Algorithms from Students' Source Code. *Submitted to Journal of Educational Data Mining*, 23 pages, submitted in June 2012.

VII Ahmad Taherkhani. 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, UK, 12 pages, 21–23 November 2012.

Author’s Contribution

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

Taherkhani carried out the literature survey, formulated the method, designed and implemented the system, collected and analyzed the data, performed the evaluation and wrote most of the paper, with Korhonen and Malmi providing feedback and suggestions throughout the process.

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

Taherkhani is the sole author of this paper.

Publication III: “Categorizing Variations of Student-Implemented Sorting Algorithms”

Taherkhani carried out the literature survey, analyzed the data, performed the categorization and wrote most of the paper, with Korhonen and Malmi providing feedback and suggestions throughout the process.

Publication IV: “Automatic Recognition of Students’ Sorting Algorithm Implementations in a Data Structures and Algorithms Course”

Taherkhani analyzed the data, performed the evaluation and wrote most of the paper, with Korhonen and Malmi providing feedback and suggestions throughout the process.

Publication V: “Automatic Algorithm Recognition Based on Programming Schemas”

Taherkhani is the sole author of this paper.

Publication VI: “Beacon- and Schema-Based Method for Recognizing Algorithms from Students’ Source Code”

Taherkhani formulated the method, designed and implemented the system, collected and analyzed the data, performed the evaluation and wrote most of the paper, with Malmi providing feedback and suggestions throughout the process.

Publication VII: “Schema Detection and Beacon-Based Classification for Algorithm Recognition”

Taherkhani is the sole author of this paper.

1. Introduction

1.1 Motivation

Data structures and algorithms are central topics in programming education. Basic programming education requires that students solve a large number of programming exercises. To help teachers assess students' work, especially in large courses, a number of automatic assessment tools are developed, including Boss [43], CourseMarker [37] and WebCAT [25].

Ala-Mutka [1] listed topics which could be analyzed using the existing tools. These included 1) functionality, 2) efficiency, 3) testing skills, 4) special features like memory management, 5) coding style, 6) programming errors, 7) software metrics, 8) program design, 9) use of specific language features, and 10) plagiarism. A more recent survey by Ihantola et al. [38] shows new activities in the field, such as integration of automatic assessment tools and learning management systems, more sophisticated ways to evaluate program functionality, and integration of manual and automatic assessment.

However, in these two comprehensive surveys, no tools have been reported, which could automatically analyze what kind of algorithms students use in their programs and how they have implemented them. For example, an automatic assessment tool verifies the correctness of a sorting algorithm by examining whether the program produces the requested correct output. However, the tool cannot easily and reliably assess that student have actually used the requested algorithm or give feedback on their implementations[1]. This is the main motivation of the work presented

[1] A simple approach would be to check some intermediate states, but this is very cumbersome and unreliable as students may very well implement the basic algorithm in slightly different ways, for example, by taking the pivot item from the left or right end in Quicksort.

in this thesis; developing methods that can automatically recognize different types of basic algorithms covered in computer science education. We call such methods Algorithm Recognition (AR).

Such methods can be applied to many other problems as well. For example, all the following problems share the common task of recognizing algorithms/parts of source code and thus can apply AR methods: source code optimization [65] (tuning of existing algorithms or replacing them with more efficient ones), clone recognition [4, 60] (recognizing and removing clones as an essential part of code refactoring), software maintenance (especially maintaining large legacy code with insufficient or non-existent documentation), and program translation via abstraction and reimplementation [100] (a source to source translation approach, which involves the abstract understanding of what the target program does).

1.2 Research Questions

Algorithms are well-defined computational procedures that take some value(s) as input and produce some value(s) as output [16] in a finite amount of time. Algorithms consist of specific instructions that should be performed in a specific order to achieve a specific goal. *Programming schemas* are high-level programming knowledge on how to solve a particular problem [21]. *Beacons*, on the other hand, are highly informative statements that imply specific structures in a program [10].

This thesis introduces a static method for recognizing algorithms from Java source code based on the concept of programming schemas (which in this thesis we also call algorithmic schemas or just schemas) and beacons. Algorithms have specific functionalities, and in order to achieve these functionalities, a programmer should use specific abstract patterns (schemas) and elements (beacons) when implementing algorithms. For example, *Roles of Variables* (RoV) [85], which we consider as algorithm-specific characteristics and beacons, explicate the ways in which variables are used in computer programs and provide specific patterns how their values are updated. Roles are concepts that associate variables with their behavior and purpose in a program. To implement an algorithm, a programmer uses a set of variables with particular roles to achieve the particular functionality in question.

In this thesis, we investigate several research questions. The main research question is:

1. *How could we automatically recognize basic algorithms and their variations from source code?*

By basic algorithm we mean the algorithms that are commonly introduced in learning resources and data structures and algorithms courses as solutions to the classical algorithmic problems, such as sorting algorithms, searching algorithms, graph algorithms, etc.

To address this question, we present different approaches and divide the main research question into the following related questions. We examine the applicability of programming schemas and algorithm-specific characteristics and beacons in AR. We investigate whether basic algorithms can be recognized by analyzing and extracting high-level schemas from the implementation code. Likewise, we examine the usefulness of characteristics and beacons in AR. Thus, two of our research questions are:

2. *Can algorithmic characteristics and beacons be utilized in AR process and how?*

3. *Can programming schemas facilitate automatic AR? How can we implement a method based on schemas?*

To answer these questions, we use a set of characteristics containing various metrics that are selected based on literature overviews. These characteristics are computed for each given algorithm implementation. Moreover, by analyzing how algorithms work, we discern a set of beacons that characterize the function and principle of each algorithm. We utilize these characteristics and beacons in AR process.

With regard to programming schemas, we develop a method that extracts schemas from a given algorithm implementation and identifies the implementation by matching the extracted schemas against a predefined set of schemas and subschemas stored in a knowledge base. We call this method a *Schema Detection Method* and use the abbreviation SDM for it.

Furthermore, we specifically study the usefulness of RoV as beacons in recognizing basic algorithms. We aim to discover how distinctive factors RoV are in identifying algorithmic patterns and how valuable they are in automatic AR process. In this regard, the research question is:

4. *How applicable and useful RoV are in recognizing basic algorithms?*

As discussed above, we analyze basic algorithms to find a set of distinctive and algorithm-specific characteristics and beacons. We then apply machine learning techniques to examine what characteristics and beacons (including RoV) can better separate implementations of different algorithms. The research question connected to this is:

5. *Can machine learning methods, and in particular the C4.5 algorithm, be used in AR problem and how accurate it is?*

We will use the C4.5 algorithm which is a well-known algorithm for generating classification trees. The algorithm selects the characteristics and beacons that can best distinguish between algorithm implementations and uses them in constructing a classification tree that can guide the AR process for a new data set. To investigate the suitability of the algorithm and the accuracy of the classification, we will perform different types of evaluation using various data sets. We will call this method a *Classification Method* and denote it as CLM.

Moreover, we investigate the possibility and advantages of combining the CLM and SDM. Therefore, the research question here is:

6. *How can we combine the SDM and CLM to get more reliable results?*

We name this combined method *Combination of Schema detection and Classification* and abbreviate it as CSC. Finally, one direction of our research is to give automatic feedback to students on their problematic algorithm implementations and make them rethink their solutions. To do this, first we need to discover what kind of problematic algorithm variations students use. We carried out a study focusing on categorizing variations of student-implemented sorting algorithms and testing how accurately Aari, the Automatic Algorithm Recognition Instrument that we developed, can recognize authentic students' sorting algorithm implementations. Thus, the final two research questions are:

7. *How can we classify students' implementations of sorting algorithms? What kind of variations of well-known sorting algorithms students use?*

8. *How accurately Aari can recognize student-implemented sorting algorithms and their variations?*

Similar studies need to be done for all the algorithms and their variations we would like to provide feedback on. The results can be used for developing a tool that gives useful feedback on students' implementations.

1.3 Structure of the Thesis

This thesis is structured as follows. Chapter 2 discusses the AR task, previous work on program comprehension and the other related research fields. Chapter 3 gives an overview on programming schemas and beacons which is based on program comprehension models, presents a definition of RoV along with an example and highlights their connection to program comprehension. Chapter 4 explains decision tree classifiers in general and briefly discusses the C4.5 algorithm. The overall process of AR including the common characteristics of algorithms is presented in Chapter 5, followed by a more specific discussion on the schemas and beacons for an analyzed set of algorithms in Chapter 6. Chapter 7 focuses on the empirical studies, data sets and results. Finally, Chapter 8 discusses related issues, summarizes the results of the thesis, outlines some directions for future work and concludes this thesis with a discussion on validity issues involved in this research.

2. Algorithm Recognition and Related Work

In this chapter, we first present an overview on the algorithm recognition task. This is followed by a discussion on the related research fields.

2.1 Algorithm Recognition (AR)

The task in AR is to identify algorithms from source code. Recognizing an algorithm involves discovering its functionality and comprehending the corresponding program code. The problem in AR is not only to identify and differentiate between different algorithms with different functionalities, but also between different algorithms that perform the same functionality. As an example, in addition to identifying and differentiating between sorting and searching algorithms, different sorting algorithms should also be identified and distinguished. AR can be applied in various problems, such as code optimization, software engineering activities, examining and grading students' work, and so on.

AR is a non-trivial task. To perform the same computational task, such as sorting an array, several different algorithms can be used. For example, the sorting problem can be solved by using Bubble sort, Quicksort, Mergesort or Insertion sort, among many others. However, the problem of recognizing the applied algorithm has several complications. First, while essentially being the same algorithm, Quicksort, as an example, can be implemented in several considerably different ways. Each implementation, however, matches the same basic idea (partition of an array of values followed by the recursive execution of the algorithm for both partitions), but they differ in lower level details (such as partitioning, pivot item selection method, and so forth). Moreover, each of these variants can be coded in several different ways, for instance, using different loops, initializations, conditional expressions, and so on. Another aspect of the complexity of the

AR task comes from the fact that in real-world programs, algorithms are not "pure algorithm code" as in textbook examples. They include calls to other functions, processing of application data and other activities related to the domain. For a more detailed discussion, see Publication I.

With respect to computational complexity, AR problem can be considered as similar to detecting semantic clones (clones that have the same functionality) from source code. As we will discuss in Subsection 2.2.2, detecting semantic clones is undecidable in general [7]. However, as will be described in Chapter 5 when discussing the AR process, we approach the problem by examining schemas and extracting the characteristics and beacons from algorithm implementations and analyzing the implementations as characteristic and beacon vectors. Furthermore, we limit the scope of our work to include a particular group of algorithms. In addition, we are not looking for an absolute exactness. Even humans make errors, and cannot always achieve perfect accuracy. Thus, recognizing algorithms in a *reasonable* precision is our aim.

2.2 Related Work

We can view AR problem from different perspectives and in connection with different research fields. We first give an overview on program comprehension research and then present other related work, explaining their relevance to AR.

2.2.1 Program Comprehension

Program Comprehension (PC) has been studied from both theoretical and practical points of view. Theoretical PC studies have focused on understanding how programmers comprehend programs. These studies introduce PC models that explain elements involved in the process of PC. Practical PC studies have been mainly motivated by finding effective solutions to be used in software engineering tasks and by developing automated tools to facilitate understanding programs [94].

The purpose of AR is to determine what algorithm a piece of code implements. Therefore, algorithm recognition facilitates PC and can be regarded as a subfield of practical PC.

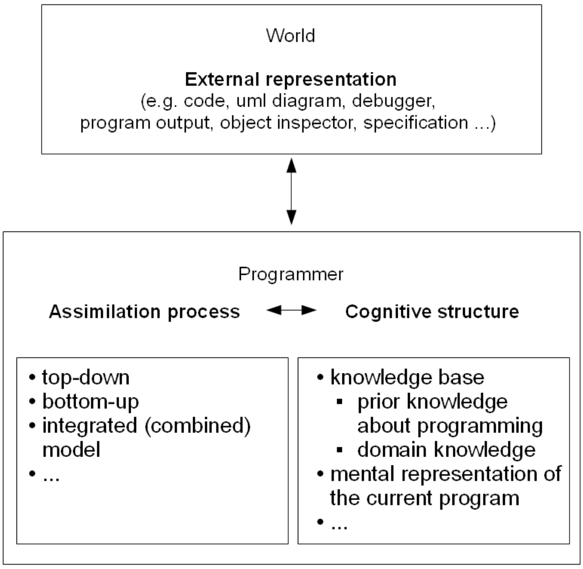

World
External representation
(e.g. code, uml diagram, debugger,
program output, object inspector, specification ...)
Programmer
Assimilation process
Cognitive structure
• top-down
• bottom-up
• integrated (combined) model
• ...
• knowledge base
▪ prior knowledge about programming
▪ domain knowledge
• mental representation of the current program
• ...

Figure 2.1. Key elements of program comprehension models [89]

Theoretical PC

Theoretical PC research deals with PC from psychology of programming point of view and tries to answer questions related to the process of understanding programs, including: what are those strategies used by programmers when comprehending programs? Which ones are the most useful? What kind of cognitive structures programmers build/have when comprehending programs? What kind of external representations are more helpful in the process of understanding?

PC is a process in which a programmer builds his or her own mental representation of the program. Understanding programs is a process that involves different elements as shown in Figure 2.1 [89]. *External representation* means how the target program is represented to the programmer. *Assimilation process* and *cognitive structure* are internal to the programmer. Cognitive structures include the programmer's knowledge base (his/her prior knowledge and the domain knowledge related to the target program) and the mental representation he/she has built of the target program. An assimilation process is the process of building a mental representation of the target program using the knowledge base and the given representation of the program. In the assimilation process, *top-down*, *bottom-up* or *integrated* strategies of building a mental representation may be used.

In a top-down strategy, the assimilation process starts by utilizing the knowledge about the application domain and proceeds to more detailed levels in the code to verify the hypothesis formed based upon the domain. In a bottom-up strategy, the assimilation process starts at a lower level of abstractions with individual code statements and proceeds to a higher level by grouping these statements. The final mental representation of the target program is constructed by repeating this process of chunking lower levels successively to higher levels. In an integrated strategy the programmer switches between the top-down and bottom-up models whenever he/she finds it necessary in order to build his/her mental representation effectively. In PC literature, integrated strategy is also referred to as combined, hybrid, opportunistic or mixed strategy.

Several PC models have been presented which differ in issues like, what assimilation strategies they recommend, what is the effect of programming paradigm on forming program and domain knowledge, etc. For more information on these models, see, for example, the following reviews on the topic: [17, 22, 69, 89, 94, 98, 99]. We will get back to some of these models in Section 3 when discussing the theoretical background of our method.

Practical PC

Practical studies on PC and the techniques and tools they develop have been influenced by the models introduced by the theoretical studies. According to Storey [94], the characteristics that influence cognitive strategies used by programmers, influence the requirements for supporting tools as well. As an example, top-down and bottom-up strategies introduced in PC 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" [94]. 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 [74].

Based on their functionality, PC tools can be divided into one of the following categories: extraction, analysis and presentation. Extraction tools perform the tasks related to parsing and data gathering. Analysis tools carry out static and/or dynamic analyses to facilitate different activities including clustering, concept assignment, feature identification, transformations, domain analysis, slicing and metrics calculations. Presentation tools comprise code editors, browsers, hypertext, and visualizations. Some tools may have multiple functionalities and be capable of carrying out different tasks from each category [94].

Knowledge-based techniques are widely adopted in practical PC tools. The basic idea is to store stereotypical pieces of code – which are called *plans*, *schemas*, *chunks*, *clichés* or *idioms* in different studies – 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 theoretical PC, there are three main techniques to perform matching: *top-down*, *bottom-up* and *hybrid* technique. Top-down techniques use the goal of the program to select the right plans from the knowledge base. This speeds up the process of selecting the right 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. For example, PROUST [41], as a tool that uses the top-down strategy of analysis, matches functional goals against pieces of code using programming plans. It gets top-level functional goals as an input and outlines how the goals are implemented in a program, but cannot identify these goals for an arbitrary program. Moreover, as top-down approaches process plans connected to the goal of the target program, they cannot perform partial plan recognition. The main concern with bottom-up techniques, on the other hand, is efficiency. As a 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. This is especially true when the target program is a real life program with thousands of lines of code. PAT, a Program Analysis Tool [36] that recognizes concepts based on pattern matching, is an example of a bottom-up analyzer. It identifies abstract concepts by analyzing semantic information such as control flow dependencies among sub-concepts. The technique is based on an Event Base and a Plan Base, where basic events are generated from code. Plans define the relation between events and they trigger a new event which corresponds to the intention of the plan. A plan needs to be defined for each implementation variant and thus the number of plans grows quickly. Hybrid techniques (e.g., [74], as discussed below) use the combination of the two techniques. In the following, we discuss some of knowledge-based techniques in more detail.

Quilici [74] developed a hybrid top-down, bottom-up technique to find operations and objects written in C language and to replace them with C++ code. In order to recognize plans, Quilici defines recognition rules that list components for each plan and describe constraints for these components. Organized in a plan library, plans have different relationships with each other: they are indexed by other plans, they have specialization relationship with other plans, and they have a list of other plans that they imply. Using these relationships, a particular program construct can activate a plan to be investigated against the code under examination. In the same way, a detected plan may suggest related indexed plans for examination. Plan indexing limits the search-space and thus speeds up finding the right plans. Specialization relationship makes it possible to first match general plans and then search for specialized version of those plans. Furthermore, using a list of implied plans, it is possible to realize existence of other plans by identifying a plan, even though those plans themselves have not been analyzed yet. Quilici defines plans as lists of attributes that characterize each plan and are represented by frames. Target programs, as well as plans are represented by an abstract syntax tree (AST) with frames as its nodes. Frames represent all types of programming objects that need to be identified and replaced, including primitive operations such as addition or more complex structures like loops. Plan recognition is performed in a depth-first manner based on specialization relationship between plans.

Kozaczynski et al. [53] developed a method for automatic recognition of programming concepts, a term they use for programming plans. The authors define abstract concepts as language-independent ideas of computation and problem solving methods and divide them into the following three classes: 1) programming concepts include general coding strategies, data structures and algorithms, 2) architectural concepts are related to architectural components such as databases, networks and operating systems, and 3) domain concepts are implementations of application or business logic. Representation of a given program is created by parsing it into an AST followed by several semantic analyses, including definition-used chain analysis and control-dependency relation analysis. Abstract concepts, concept recognition rules (information that describe what the concepts are and how they can be recognized based on lower-level concepts, i.e., sub-concepts) and constraints on and among the sub-concepts are organized in a concept classification hierarchy, called a concept model. The recognition is carried out by building abstract concepts on top of the AST nodes of the target program in a top-down mode. Evaluation of an abstract concept may be triggered by each AST node. The concept recognition rules related to the triggered abstract concept is used to compare it to the triggering part of the AST. The user interface makes it possible to browse the results of the recognition process.

Wills [102] uses the term *clichés* for commonly used computational structures. The target code is represented as annotated flow graphs by GRASPER, the system that implements the technique, and clichés are encoded as an attributed graph grammar. The problem of recognizing clichés is thus transformed into the problem of parsing a flow graph of the given source code based on the graph grammar, which is NP-complete. The cliché library used in the approach is manually constructed from textbooks and other sources. The result of the recognition is a hierarchy of Clichés and the relationships between them as identified from the analyzed program.

Among others, *fuzzy reasoning* technique [12, 13] was introduced to improve the performance of the knowledge-based PC techniques and address their problem of scalability and inefficiency. Instead of matching all the statements and plans of the target program against the plans in the knowledge base, these approaches first identify candidate chunks in the target code using data dependency analysis and a set of heuristics, as described in [11]. These chunks are then abstracted and mapped into higher level concepts that are used to retrieve a set of similar program plans from a plan library. The retrieved plans are ranked using a fuzzy reasoning technique and the plan identified as the most similar to the candidate chunk is used to perform the costly more detailed matching for automated program understanding.

Our SDM draws on the concept of schemas, which is central in many PC models. Another relevance of our method to PC research is that our method matches the schemas detected from the given program against the schemas stored in a knowledge base in a bottom-up manner, just like knowledge-based practical PC methods do. We will get back to this in Chapter 3.

A number of other techniques and tools are introduced to facilitate PC in software engineering activities. Many of these techniques deal with concept location, that is, finding fragments of the code that implement the domain concept that a programmer is looking for in order to, for example, perform a change request. As discussed in [20], concept location approaches are broadly divided into dynamic techniques, which are based on analyzing execution traces and their mapping to source code (see, for example [24, 26]) and static techniques, which analyze program dependencies and textual information within source code (e.g., [28, 61]). Most concept location approaches are interactive and iterative [28], where the process is initiated by the programmer by formulating a domain concept as a query. Information retrieval methods, such as Latent Semantic Indexing (LSI) are used to map the query to software components (see, e.g., [61]). The programmer then evaluates the results of the query and if necessary, makes more detailed queries. Hybrid approaches use both static and dynamic analyses to address the limitations of these techniques by using static information to filter the execution traces (see, e.g., [59, 72]). For example, in [72], LSI information retrieval technique is combined with a dynamic technique (scenario-based probabilistic ranking) to improve the effectiveness and precision of feature location. Poshyvanyk et al. introduced a method combining formal concept analysis and LSI [71, 73], that is able to reduce the programmers' effort by producing more relevant search results.

2.2.2 Clone Detection

Clones are code duplicates that result from copying and pasting code fragments for code reuse purposes, either directly or with minor modifications. Cloning is commonly practiced by software developers because it is an easy way to develop software [54]. Studies suggest that up to 20% of software systems are implemented as clones [62, 81]. Cloning is also an endemic problem in large, industrial systems [6, 23]. It makes software maintenance more complicated and increases maintenance costs: code duplication may duplicate errors and changes to the original code must be also performed to the duplicated code [54]. Clones can be a substantial problem in software development and maintenance because inconsistent changes (e.g., bug fixes) to clones can lead to unexpected behavior [44].

Clone detection (CD) improves the quality of the source code and eliminates harmful consequences of cloning. In addition, CD has great potential in the maintenance and re-engineering of legacy systems [62] and can benefit many other software engineering tasks. CD techniques can be applied in activities that involve a comparison analysis between two different versions of a system. For example, they can be applied in origin analysis, where the problem is to analyze two versions of a system to understand where, how and why structural changes have occurred [33]. Furthermore, CD techniques can be applied in evolution analysis, that is, mapping two or more different versions of a software for finding a relation between them in order to understand their evolution behavior [82] (for more information on application of CD in other domains see [80]). From our point of view, however, the most relevant application of CD is in program comprehension field: identifying functionality of a cloned fragment helps us understand other parts of the software that include the same fragment.

Clones are defined as code fragments that are similar. Here, *similarity* is either based on semantic or program text. Detecting semantic similarity in an undecidable problem in general and therefore most approaches and surveys have focused on program text similarity [7, 97]. Program text similarity is defined in terms of text, tokens, syntax, code metric vectors or data and control dependencies [97]. Different clone taxonomies have been presented (see, e.g., [3, 45, 46, 62]). Most research literature classify clones into three types [97]. Tiark et al. [97] present a refined categorization, as follows: 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.

Several techniques for detecting clones have been introduced including: textual approach (text-based comparison between code fragments, see, e.g., [23, 60]), lexical approach (the source code is transformed into a sequence of tokens and these are compared, see, for example, [2, 5]), metrics-based approaches (the comparison is based on the metrics collected from the source code, as, for example, in [51, 62]), tree-based approaches (clones are found by comparing the subtrees of the abstract syntax tree of a program, see, e.g., [6, 104]) and program dependency graphs (the program is represented as program dependency graphs and isomorphic subgraphs are reported as clones, as, e.g., in [50, 54]). Roy et al. [81] and Bellon et al. [7] compare and evaluate a number of different clone detection techniques and tools in their surveys.

The problem of AR is close to the CD area. The purpose of AR is to look for similar patterns of algorithmic code in the source code, in the same way that CD techniques look for similar code fragments. Some of the techniques that we use in AR are also used in CD (e.g., analyzing language constructs and software metrics). Recognizing algorithms from source code can be compared with and considered as the type-3 and semantic clones, where according to the recent surveys [7, 97] the current techniques and tools perform poorly. There are also other similarities between the techniques we use for AR problem and recent trends in CD. In their recent study on the state-of-the-art CD tools, Tiark et al. [97] suggest that decision trees should be used to distinguish real clones from false positives. In particular, they use several different metrics and apply supervised classification (a decision tree constructed by the C4.5 algorithm) to identify distinguishing characteristics and the most useful combinations of metrics that can indicate real type-3 clones.

To summarize, the main difference between AR and CD is that in AR, we look for the implementations of a predefined set of algorithms whereas in CD, clones are unknown code fragments. On the other hand, as in CD the goal is to find similar or almost similar pieces of code, it is not necessary to know what algorithms the detected clones implement. CD methods can utilize all kinds of identifiers that can provide any information in the process. These identifiers may include comments, relation between the code and other documents, etc. For example, comments may often be cloned along with the piece of code that programmers copy and paste. We do not use these types of identifies in AR.

2.2.3 Program Similarity Evaluation Techniques

The problem in program similarity evaluation research is to find the degree of similarity between computer programs. The main motivation for these studies is to detect plagiarism and prevent students from copying each other's work.

Based on how programs are analyzed, these techniques can be divided into two categories: *attribute-counting* techniques (see, for example, [34, 42, 56, 78]) and *structure-based* techniques (see, e.g., [40, 47, 67, 103]). Attribute-counting techniques use distinguishing characteristics to find the similarity between the two programs, whereas structure-based techniques focus on examining the structure of the code. Attribute-counting methods have been criticized as being sensitive to even textual modification of the code, but structure-based methods are generally regarded more tolerant to modifications imposed by students to make two programs look different [67]. Structure-based methods can be further divided into string matching based systems and tree matching based systems.

Program similarity evaluation techniques widely use software metrics (such as Halstead's metrics) as a measure of similarity. The relevance of these techniques to our method is that, as we will discuss in Chapter 5, our method also utilizes these metrics.

2.2.4 Reverse Engineering Techniques

Chikofsky and Cross define reverse engineering as the process of creating higher-level abstractions from source code [15]. This involves analyzing the target system and identifying its components and interrelationships between them. More specifically, reverse engineering techniques are used to understand a system in order to recover its high-level design plans, create high-level documentation for it, rebuild it, extend its functionality, fix its faults, enhance its functions and so forth. By extracting the desired information out of complex systems, reverse engineering techniques provide software maintainers a way to understand these systems, thus making maintenance tasks easier. Understanding a program in this sense refers to extracting information about the structure of the program, including control and data flow and data structures, rather than understanding its functionality. Different reports that can be generated by carrying out these analyses indeed help maintainers gain a better understanding of a program and enable them to modify the program in a much more efficient way, but do not provide them with direct and concise information about what the program does or what algorithm is in question. Reverse engineering techniques have been criticized for the fact that they are not able to perform the task of PC and deriving abstract specifications from source code automatically, but they rather generate documentation that can help humans to complete these tasks [75].

However, reverse engineering field provides useful techniques and methods for program analysis. The research fields discussed above use widely these techniques and in this sense are closely related to reverse engineering. As an example, many studies on concept location discussed in Section 2.2.1 are actually published in reverse engineering forums (see, e.g., [61]), as well as several studies on clone detection discussed in Section 2.2.2 (see [2, 3, 54]).

2.2.5 Roles of Variables

Roles of Variables (RoV) is a research field that explores the patterns how variables are used and their values are updated in programs. We use RoV as distinguishing factors to recognize algorithms and thus, RoV research is close to our research. Concerning theoretical PC, studies on using RoV in elementary programming courses show that roles provide a conceptual framework for novices that helps them comprehend and construct programs better [14, 55, 86]. Utilizing roles in teaching 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" [55]) 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" [55]). We will discuss RoV and their relation to PC and AR in more detail in the next chapter.

3. Program Comprehension and Roles of Variables, a Theoretical Background

In this chapter, we first present a theoretical background based on program comprehension (PC) models for the algorithmic schemas and beacons part of our method. In Section 3.2, we discuss the concept of roles of variables and in Section 3.3, we explain how roles can be utilized as beacons. Beacons are important elements of PC models and we will discuss how the concept connects roles of variables with PC.

3.1 Schemas and Beacons

Schemas consist of generic conceptual knowledge that abstract detailed knowledge of programming structures. Détienne defines schemas as formalized knowledge structures [21]. Through programming experience, programmers create and extend schemas. When dealing with new tasks with similar schemas, programmers use their stored schemas to understand and solve these tasks that differ in lower level of abstraction and implementation details. Schemas may contain other schemas in a hierarchical manner. Schemas and the process of schema creation are the focus of a number of theoretical PC studies. These studies show that programmers use schemas when working on programming tasks (see, for example, [64, 79, 91, 92]). Possession of schemas is a key factor that turns novices into experts [21, 91]. According to Détienne, the studies of Soloway and Ehrlich are among the most important reports on the topic [21]. Soloway and Ehrlich define schemas (which they call *plans*) as "generic program fragments that represent stereotypic action sequences in programming" [91]. To investigate the differences between experts and novices, they used *plan-like* and *unplan-like* programs. A plan-like program is a program that uses stereotypical plans and conforms to rules of programming discourse, unlike an unplan-like program (we will get back to this in Section 3.3). Soloway and Ehrlich showed that the novices perform poorly due to their lack of programming plans and discourse rules. Moreover, they showed that the experts perform significantly better with plan-like programs than with unplan-like programs, because plan-like programs follow the patterns that they have developed and used when solving the tasks, while unplan-like programs do not.

Beacons are key statements or features that indicate the existence of a particular structure or operation in code. A particular structure or operation could have different related beacons that indicate its occurrence more or less strongly. Moreover, the same beacon may indicate different structures or operations [10]. While novices do not use beacons heavily, experts rely on beacons and use them as important elements in understanding programs [18, 101]. Beacons provides a link between the source code and the process of verifying the hypotheses driven from the source code and thus help programmers to accept or reject their hypotheses about the code. As an example, existence of a swap operation, especially inside a pair of loops, indicates sorting of array elements [10]. Soloway and Ehrlich's model [91] use the term *critical lines* for the same meaning: the statements that carry important information about program plans and can be considered as the key representatives of the plans that help experts recognize them. As can be noted, the concepts of schema and beacon are closely connected.

The idea of algorithmic schemas and beacons of our method comes from the corresponding concepts introduced by PC models. Our SDM uses a set of predefined high-level algorithmic schemas stored in its knowledge base. In these schemas, all details, such as the type of loops, conditionals, irrelevant assignment statements, etc., are abstracted out. These correspond to schemas that experts have developed. When recognizing an algorithm, the method extracts the schemas of the same abstract level from the given program and compares it with those from its knowledge base to find a match, just like an expert deals with a given program. That is, abstracted stereotypical implementations of algorithms are used to automatically recognize new implementation instances that differ in implementation details.

For each supported algorithm, the knowledge base includes the related schemas, subschemas and beacons. These consist of, for example, loops, recursion, specific operations, etc. The method makes the final decision by putting these separately recognized elements together and examining their relationships in terms of nesting, execution order, etc., again, just like experts do. To give an example, we can repeat the example of Brook [10] presented above: a swap operation inside a pair of nested loops forms a schema that indicates sorting of array elements.

Algorithm-specific beacons are utilized to distinguish between the algorithms that have similar algorithmic schemas. For example, the above example of swap operation within two nested loops can indicate an implementation of Bubble sort, but also an implementation of a variation of Insertion sort that performs a swap operation (instead of a shift operation) in the inner loop. In these cases, patterns that implement features specific to the way each algorithm works are extracted from the source code. These patterns are utilized as beacons to recognize borderline cases. We will discuss this in Chapter 5.

3.2 Roles of Variables

Roles of Variables (RoV) constitute an essential part of our method in Algorithm Recognition (AR). In this section, we first discuss RoV, the concept, history and original application. In the next section, we explain the relationship between RoV and PC and outline how RoV can be utilized as beacons in PC and AR.

The concept of RoV was first introduced by Sajaniemi [85]. The idea behind RoV is that each variable used in a program plays a particular role that is related to the way it is used. RoV are specific patterns how variables are used in source code and how their values are updated. 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 argue, RoV are a part of programming knowledge that have remained tacit [84]. 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. Although RoV 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 this thesis, we have extended the application of RoV by applying them in the problem of AR.

Roles are cognitive concepts [8, 30], implying that human inspectors may have a different interpretation of the role of a single variable. However,

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.

Table 3.1. The roles of variables and their descriptions

as Bishop and Johnson [9] and Gerdt [31] describe, roles can be analyzed automatically using data flow analysis and machine learning techniques.

As reported in [85], 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 [87], a total of 11 roles are recognized. These roles are presented in Table 3.1[1]. Note that the three last roles shown in the table are related to data structures.

3.2.1 An Example

Figure 3.1 shows a typical implementation of Selection sort in Java. There are five variables in the code with the following roles. A loop counter, that is, a variable of an integer type used to control the iterations of a loop is a typical example of a stepper. In the figure, variables *i* and *j* have the stepper role. Variable *min* stores the position of the smallest element found

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

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

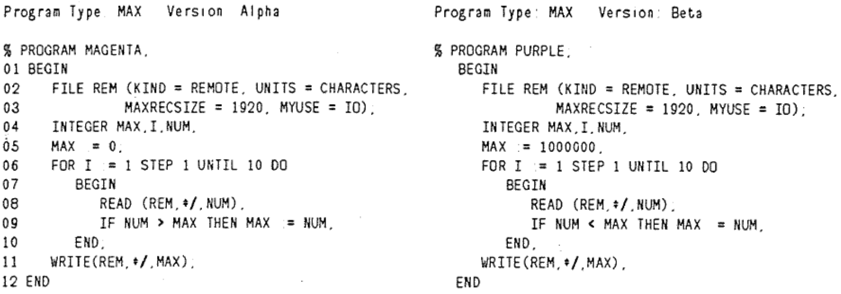
Figure 3.1. An example of stepper, temporary, organizer and most-wanted holder roles in a typical implementation of Selection sort

so far from the array and thus has the most-wanted holder role. A typical example of the temporary role is a variable used in a swap operation. Variable *temp* in the figure demonstrates the temporary role. Finally, data structure *numbers* is an array that has the organizer role.

3.3 The Link Between RoV and PC

RoV were introduced as a concept to help novices learn programming. Although some work on RoV has been linked to theoretical PC research (e.g., Kuittinen and Sajaniemi's study [55] draws on Pennington's work [70]), the author is not aware of any studies about further explicit connection between the two, nor further application of RoV to theoretical PC. Automatic role detection tools, such as [9], [29] and [31] can correspondingly be considered as related to practical PC research field. In this section, we discuss how RoV can serve as beacons in PC.

In the previous section, we presented the definition of critical lines and plan-like/unplan-like programs as introduced by Soloway and Ehrlich [91]. Figure 3.2 shows the two programs in Algol language which Soloway and Ehrlich used in their study on PC [91]. The two programs are essentially identical except for lines 5 and 9. The Alpha program (on the left side of the figure) is a maximum search plan and the Beta program (on the right side) is a minimum search plan. In the study, these programs were shown to expert programmers (41 subjects) three times (each time for 20 seconds). On the first trial, the programmers were asked to recall the program lines verbatim as much as they could. On the second and third



```
Program Type  MAX   Version  Alpha

% PROGRAM MAGENTA,
01 BEGIN
02    FILE REM (KIND = REMOTE, UNITS = CHARACTERS,
03             MAXRECSIZE = 1920, MYUSE = IO);
04    INTEGER MAX,I,NUM,
05    MAX  = 0;
06    FOR I  = 1 STEP 1 UNTIL 10 DO
07       BEGIN
08          READ (REM,*/,NUM),
09          IF NUM > MAX THEN MAX  := NUM,
10       END;
11    WRITE(REM,*/,MAX);
12 END
```

```
Program Type: MAX   Version: Beta

% PROGRAM PURPLE,
   BEGIN
      FILE REM (KIND = REMOTE, UNITS = CHARACTERS,
               MAXRECSIZE = 1920, MYUSE = IO);
      INTEGER MAX,I,NUM,
      MAX := 1000000,
      FOR I := 1 STEP 1 UNTIL 10 DO
         BEGIN
            READ (REM,*/,NUM),
            IF NUM < MAX THEN MAX  = NUM,
         END,
      WRITE(REM,*/,MAX),
   END
```

Figure 3.2. Plan-like and unplan-like programs used in Soloway and Ehrlich PC study [91]

trials, the programmers were asked to correct or complete their recall of the previous trial. The corrections/additions were made using different color pencil each time which made it possible to track the changes carried out on each trial. The programmers were expected to recall the Alpha program earlier, since it is a plan-like program. In the Beta program, the variable name (*max*) does not reflect its function, which is a minimum search function. Therefore, the program violates the discourse rule of using proper variables names and thus is considered as an unplan-like program. In their study, Soloway and Ehrlich focused on lines 5 and 9, as they are the critical lines of these programs. The results showed that the programmers recalled significantly more critical lines earlier from the Alpha program than the Beta program. The conclusion was that plan-like programs help programmers in the PC task and that critical lines are important in the process.

Roughly speaking, line 9 in Figure 3.2 and lines 4 and 5 in Figure 3.1 are identical. They both make a comparison between the currently encountered number and the minimum/maximum value of an array of numbers encountered so far. They then store the value of the current number into the variable holding the minimum/maximum value, if it is smaller/larger than the current value of that variable. As illustrated in Figure 3.1, variable *min* in line 5 has the most-wanted holder role. Therefore, since line 9 in Figure 3.2 is a critical line, most-wanted holder role can also be considered as a critical line (or a beacon) in a search plan. In addition, as discussed above, Brooks [10] regards the presence of a swap operation as a beacon in sorting functions. Swap operations typically include a temporary role and thus this role can be regarded as part of the beacon in the example of Figure 3.1. As we will discuss in this thesis, RoV have an important part in our method in automatic recognition of algorithms. Specifically, for example, the presence of the most-wanted holder role is a strong indicator (i.e., beacon) in recognizing Selection sort algorithm implementations.

As discussed in Chapter 2, in a study on effects of teaching RoV in elementary programming courses Kuittinen and Sajaniemi [55] found that “the teaching of roles seems to assist in the adoption of programming strategies related to deep program structures, i.e., use of variables”. This is a clear indication of applicability of RoV in PC. Furthermore, since 11 roles can cover all variables in novice-level programs [85], as a tool to be used in PC, RoV are inclusive and comprehensive as well.

From the above discussion, we hypothesize that RoV can be used in PC tasks as beacons. In the case of AR, we will show it in this thesis. As RoV should first be learned before they can be utilized as beacons, one can argue that roles may place a burden on programmers in PC tasks instead of helping them. However, as Sajaniemi and Navarro argue [84], RoV are tacit knowledge of experts. Thus, for experts, roles are somehow already familiar and do not require a huge effort to be learned. In case of novices, it can be logically concluded from the Sajaniemi’s and Navarro’s argument, that novices will (tacitly) adopt RoV, just like other programming skills, as they gain more experience in programming and become experts. It should be noted that, as discussed above, the exact same difference between experts and novices applies with regard to other elements of PC models as well, such as schemas, beacons, general programming knowledge and other elements of programmers’ knowledge base.

4. Decision Tree Classifiers and the C4.5 Algorithm

In this chapter, we first discuss the important issues about decision tree classifiers in general. This is followed by a brief discussion on the C4.5 decision tree classifier algorithm [77]. The issues discussed in this chapter are intended to help the reader understand how decision trees work. Readers who are familiar with these topics may skip this chapter.

4.1 Decision Tree Classifiers in General

Decision tree classifiers, also called *classification trees* or simply *decision trees*, are used to classify different instances of a data set into appropriate classes. Decision trees belong to *supervised machine learning classification* methods [52]. In these methods, first a set of known instances, called a *training set*, is introduced to the system. The system classifies each instance of the set, associates each class with the attributes of each instance and learns to what class each instance belongs. Based on what the trained system has learned in the *learning phase*, it is able to classify instances of a previously unseen set (i.e., the *testing* or *evaluating set*) in the *testing* or *evaluating phase* [96].

Unsupervised machine learning methods use unlabeled instances. Unsupervised algorithms, for example, clustering algorithms, examine a given data set to find regularities between the instances and to group them into meaningful clusters [39].

In a training set used to build a decision tree classifier, each instance consists of a group of attributes that describe that instance. One of the attributes is the class of the instance. In the learning phase, the task is to find a function that maps from other attributes to the class attribute. In the testing phase, the task is to assign a correct class to each instance of the testing set. The mapping function found in the learning phase is used to carry out the task in the testing phase.

A decision tree consists of internal nodes (including the root node) and leaves. Each internal node contains a test that results in splitting the data set into subsets based on the outcome of the test. Decision trees are constructed using the divide and conquer principle. Starting from the root node, based on the selected attribute, the instances of a training set are divided into two branches. This is repeated recursively for each internal node only for those instances that have reached that node. Each leaf is labeled with the corresponding class. The outcome of each test at each internal node is shown on each arc from that internal node to its children. If the internal nodes of a decision tree use a single attribute of the input instance to determine which child to visit next, the tree is called *univariate tree*. In *multivariate tree*, more than one attribute is tested in internal nodes [68].

When a new instance of a set is given to a decision tree, each of its attributes is tested at the corresponding internal nodes, starting from the root. Depending on the outcome of the test in each internal node, the appropriate child node of that internal node is visited next. This is continued until a leaf is encountered and a class is assigned to the instance. There are different issues associated with decision trees and their performance. In what follows, we present an overview on some of these issues.

Finding the best attribute. Different attributes of an instance have different values in how well they are able to split the data. In tree *induction* (the process of constructing a tree from the training set [68]), it is important to select attributes that can discriminate between different classes of data in the best possible way. The attribute that best distinguishes between the samples of the training data will be located in the root of the tree [52]. This selection process is then repeated to select the best distinguishing attribute for the internal nodes in a recursive manner. In the literature, this is often referred to as finding the *best split* [68]. The best split improves the accuracy of the decision tree and helps to keep its size right. To find such an attribute, all the attributes are examined using some *goodness measure*. These goodness measures are basically statistical tests and include information gain, distance measures and Gini index, to name a few [52, 68]. The explanation of all these measures is out of the scope of this thesis, but information gain is discussed in more detail in relation to the C4.5 algorithm in Appendix A.

Finding the right size. After being built, decision trees need to be simplified as they are often unnecessarily complex. Complexity is associated with *overfitting*, which in turn causes *generalization* problem. Overfitted trees adopt the structure of the learning data in such a detailed level that they become very specific to that data and cannot classify instances of an unseen data well.

It has been claimed that the quality of a decision tree depends more on the right size than the right split [68]. There can be many different sizes of a decision tree that are correct over the same training set, but the smaller size is preferred. A simpler decision tree is more likely to correctly recognize more instances of a testing set, because it can capture the structure of the problem and the relationship between the class of an instance and its attributes more effectively [76]. In addition to higher accuracy, smaller trees are more comprehensible as well [49]. Choosing the best discriminating attributes helps keep the size of a tree small. Because the problem of finding the smallest decision tree that is consistent with the training set is NP-complete [52, 77], selecting the right tests is very important in generating near-optimal trees.

In his survey on automatic construction of decision trees, Murthy [68] lists several methods for obtaining right sized trees. The most widely used method is *pruning*. In pruning, first the complete tree is built. Here, the complete tree means the tree where no splitting will improve the accuracy of the tree on the training data. In the next step, those subtrees with only little impact on the accuracy of the tree are removed. There are many variations of pruning methods, and it has been shown in different studies that there is no single best pruning method that is superior to the others [52, 68]. Another method is called *stopping* or *prepruning*, where the instances of the data set are not subdivided any further at some point. An interesting approach to pruning is to combine the tree building phase and the pruning phase. In this approach, if it appears that a node will be removed in the pruning phase, it will not be expanded in the building phase in the first place. This will result in saving a noticeable amount of time [52].

There are several other issues related to decision trees, such as how to deal with missing attributes, how to measure the quality of decision trees, etc., that are out of the scope of this thesis. To sum up, decision trees are powerful, simple and easily interpretable classifiers. Because of these properties decision trees are used in many different fields, including statistics, pattern recognition, decision theory, signal processing, machine learning and artificial neural networks [68].

4.2 The C4.5 Decision Tree Classifier

We chose the C4.5 algorithm [77] to build the decision tree, because it is a widely used and the most well-known algorithm for doing so, and has a good combination of error rate and speed [58]. The C4.5 algorithm preserves the advantages of its predecessor, the ID3 algorithm [76], but is further developed in many regards. It provides an accurate, readable and comprehensible model about the structure of data and the relationship between the attributes and this structure. More details on how the C4.5 algorithm deals with the important issues related to building decision trees presented in the previous section can be found in Appendix A.

5. Overall Process and Common Characteristics

We developed two different methods for recognizing algorithms: the *schema detection* (SDM) and the *classification* (CLM). Moreover, we developed a method that uses a combination of these two. This section gives an overview of these methods and describes the processes that are commonly used for recognizing different algorithms in general, including detecting algorithmic schemas, evaluating the estimated accuracy of the classification, constructing a decision tree classifier and using the constructed decision tree in recognizing previously unseen instances. This discussion is generic and does not detail how these processes work for particular algorithms. This is the topic of the next chapter. Instead, the common characteristics that are used for all types of algorithms are listed in detail.

5.1 Overall Process

Constructing a decision tree for classifying previously unknown instances typically consists of three main phases: constructing a tree using a set of training data instances, evaluating the estimated accuracy of the classification and using the tree to classify the instances of an unseen data set. These phases are illustrated in Figures 5.1 and 5.2: the first two phases in 5.1 (Steps 3 and 4), and the third phase in 5.2 (Step 3). These figures show the combination of SDM and CLM, which we abbreviate as CSC.

Figure 5.1 includes four steps represented by rectangles with white background. The process starts with detecting schemas and the related beacons from an input program. This step identifies algorithmic schemas and extracts the code that implements the target algorithm from the given program so that only this code is further processed and the irrelevant application data processing code is left out of the process. In order to identify schemas, we compute the schema-related beacons in this step as well (we will get back to this in Chapter 6). If this step detects the schemas successfully, its output is the implementation part of the algorithm from the original program, otherwise the same original input program.

Next, the characteristics and other beacons are computed and stored in a database as a vector representation of the analyzed program. These vectors are labeled by their correct type (denoted by the dashed arrow in the figure) and given to the C4.5 algorithm as learning data, based on which a decision tree is constructed (Step 3). In the final step, the estimated accuracy of the classification is evaluated using leave-one-out cross-validation technique.

It should be noted that Steps 3 and 4 in Figure 5.1 are independent from each other. This means that before we build a classification tree, we can evaluate the estimated accuracy 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 5.2 shows how the constructed decision tree is used to classify the instances of a previously unseen data set. The steps with white background are identical with those with the same name in Figure 5.1, with the differences pertaining to the steps represented by gray rectangles. The step denoted by 0 starts the process. It is a preprocessing step performed as a part of input data validation, where input programs are automatically tested using an automatic assessment system that gives feedback about the correctness of the program in terms of black-box testing. If a program does not pass the tests in this step, it will not be further analyzed. Correctly working implementations are further processed in order to detect their schemas and extract and store their characteristics and beacons as discussed above.

Step 3, "Recognize unseen instances", gets the classification tree constructed in Figure 5.1 and the unlabeled vector representations as an input. It assigns a type to each instance by traversing the classification tree according to the characteristics and beacons of that instance.

The CSC, as illustrated in Figure 5.1, is described in more detail in Publication VI and Publication VII. Publication II discusses the CLM, that is, Steps 2, 3 and 4 of the figure in more detail without the SDM. A more detailed description of the SDM can be found in Publication V. Finally, Publication IV discusses the process depicted in Figure 5.2 in more detail without the SDM.

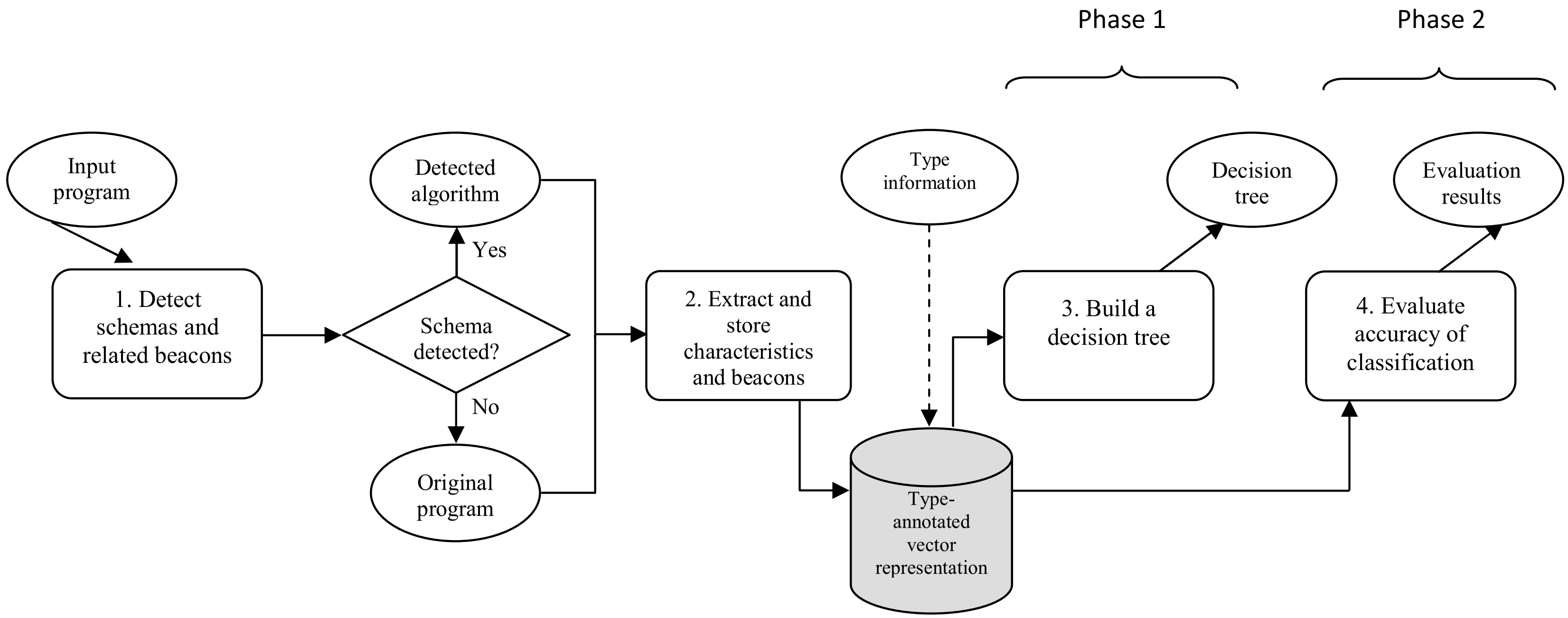

Phase 1
Phase 2
Input program
1. Detect schemas and related beacons
Detected algorithm
Yes
Schema detected?
No
Original program
2. Extract and store characteristics and beacons
Type information
Type-annotated vector representation
3. Build a decision tree
Decision tree
4. Evaluate accuracy of classification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

Figure 5.1. The process of building a decision tree and evaluating the estimated accuracy of the classification. Phase 1 and Phase 2 denote the first two phases in constructing a decision tree for classifying unknown instances, see text

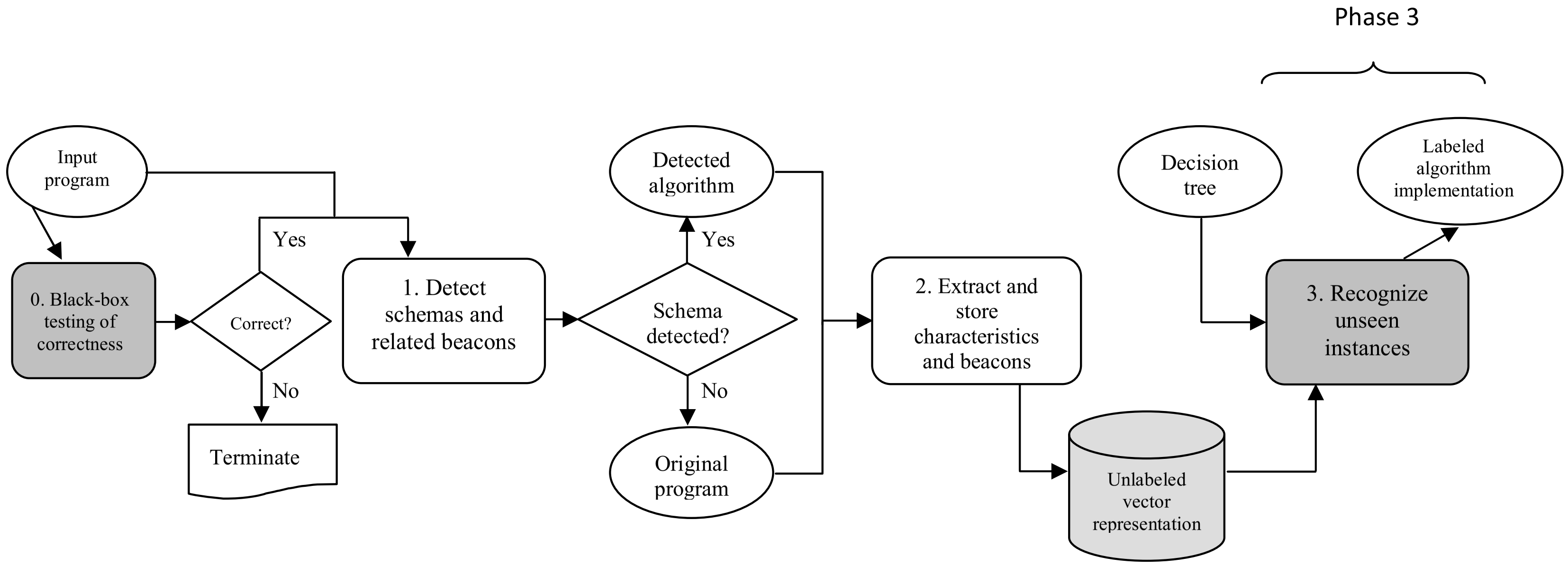

Phase 3
Input program
0. Black-box testing of correctness
Correct?
Yes
No
Terminate
1. Detect schemas and related beacons
Schema detected?
Yes
Detected algorithm
No
Original program
2. Extract and store characteristics and beacons
Unlabeled vector representation
Decision tree
3. Recognize unseen instances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

Figure 5.2. An overview of the process of recognizing previously unseen algorithm implementations. Phase 3 indicates the final phase in a typical construction of a decision tree for classifying unknown instances, see text

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) [63].
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

Table 5.1. The numerical, truth value and structural characteristics

5.2 Common Characteristics

We define *characteristics* as the shared features of all algorithm implementations. *Beacons*, on the other hand, are algorithm-specific features that distinguish a particular algorithm from others.

Table 5.1 shows the characteristics that we compute from given programs. We divided the characteristics into three groups: *numerical characteristics*, *truth value characteristics* and *structural characteristics*[1]. The numerical characteristics are commonly used software metrics that denote features of the code expressed as positive integers. The first six characteristics shown in the table are Halstead's metrics [35]. Structural characteristics allow us to identify language constructs and different patterns as well as algorithm-specific beacons.

The characteristics of Table 5.1 were selected based on manual analyses of many different types of sorting algorithms in the early stages of our work, and as a result of literature reviews especially on program similarity evaluation techniques discussed in Chapter 2. These characteristics include

[1] In Publication I and Publication II, the last two groups are named as "descriptive characteristics" and "other characteristics".

various metrics, such as size metrics, control-flow metrics (the cyclomatic complexity) and data-flow metrics (roles of variables), and thus represent different program aspects. We posited a hypothesis that these characteristics, along with the algorithm-specific beacons (that will be discussed in Chapter 6) would be sufficient to recognize different sorting algorithm implementations. As we developed our method further to cover variations of sorting algorithms as well as other fields of algorithms, we discerned several other useful beacons that, in addition to these characteristics, are computed from the given code and used in the process.

We implemented a tool named Aari (an Automatic Algorithm Recognition Instrument) that computes all the schemas, characteristics and beacons for programs written in Java. The characteristics and related beacons are stored in a database and thus, each algorithm is represented by an n-dimensional vector in the database where n is the number of characteristics and beacons. We call these characteristic and beacon vector representations the *technical definitions* of the corresponding algorithm implementations.

In our method, roles of variables can be considered as both characteristics and beacons. Roles are characteristics, because they are detected for all variables in all given programs and in this sense, are common features of all algorithms. Roles are also beacons in the sense that, as we will discuss in Chapter 6, existence of particular roles in implementations of particular types of algorithms is investigated as algorithm-specific features to distinguish these algorithms from others. It should also be noted that we use roles as truth value characteristics, as we are interested to know whether or not a particular role appears in a given algorithm implementation. Roles may also be used as numerical characteristics (e.g., how many variables have a particular role in an input program). This would provide us an additional distinguishing feature when differentiating between algorithm implementations that use different numbers of a particular role.

5.2.1 Computing Characteristics

In this subsection, we explain how some numerical characteristics are computed.

Salt [88] and Miller et al. [66] discuss strategies for calculation of Halstead's metrics for Pascal and Ada programs. We compute operators as consisting of all arithmetic, relational and logical operators as well as keywords and method calls. Likewise, identifiers (variables and all other identifiers other than keywords), constants, literals and type specifiers are computed as operands. These are computed by scanning source code.

The cyclomatic complexity [63] is a measure of how many paths there are through a program. It is defined with reference to the control flow graph of the program and is calculated as $CC = E - N + P$, where E represents the number of edges of the graph, N is the number of nodes of the graph, and P the number of connected components. As a software metric, the cyclomatic complexity can be computed as $CC =$ "The number of decision points (i.e., if statements or conditional loops)" + 1 [63].

We use the following strategy for computing dependency information between variables. Variable i is directly dependent on variable j, if i gets its value directly from j. If there is a third variable k on which j is directly or indirectly dependent, i also becomes indirectly dependent on k. A variable can be both directly and indirectly dependent on another one.

The way how the other characteristics of Table 5.1 can be computed is straightforward. We will discuss in the next section, how roles of variables are detected from a program.

5.3 The Tool for Detecting Roles of Variables

A tool developed by Bishop and Johnson for automatic detection of roles of variables [9] is integrated into Aari. The tool detects roles using program analysis techniques, particularly program slicing and data flow analysis. A set of example programs was used to analyze how each role can be defined in terms of the way a variable is assigned and used. Based on comparison of these assignments and usage conditions with the roles, a set of conditions, as shown in Table 5.2, were identified based on which role assignments can be checked. To detect roles, all occurrences of each variable in the program are captured first. The outcome of this analysis is the program slice for each variable. This is followed by data flow analysis for each program slice. The tool then compares the assignments and usage conditions of each variable of the target program with those predefined conditions. If the user has provided a role for a variable, the tool checks whether the corresponding conditions for the provided role are met by the corresponding variable. If so, the tool confirms that the role provided by the user is correct. Otherwise, the tool prints the role it believes to be correct and justifies its decision by giving an appropriate message. If there is no role suggested by the user, the tool simply prints the role it

Rule	Description
A	Variable is assigned in a loop.
B	Variable is used in its assignment loop.
C	Variable is used conditionally in its assignment loop.
D	Variable is used directly for its assigning loop condition.
E	Variable is used indirectly for its assigning loop condition.
F	Variable is assigned in “for” loop statement.
G	Variable is used directly in the program.
H	Variable is assigned in a branch for which it is part of the condition.
I	Variable appears directly on both sides of assignment statement.
J	Variable appears indirectly on both sides of assignment statement.
K	Variable is directly toggled within a loop.
L	Variable is indirectly toggled within loop.
M	Variable is incremented/decremented within a loop.
N	Variable is used outside of loop in which it is assigned.
O	Variable is assigned in loop before it is used in that loop.
P	Variable is used conditionally for a loop outside of its assignment loop.
Q	Variable appears in array organizing type statement.
R	Variable is of type array.
S	Variable is assigned within a loop with a combination of other variables, values and operators.
T	Variable is assigned with the output from a method call.
U	Variable is assigned with a value resulting from instantiation of a new object or directly with boolean value.

Table 5.2. The rules based on which roles of variables are detected. See [9]

considers the most appropriate for the variable in question. For details on the assignments, usage conditions and how the role detector works see [9].

Bishop and Johnson developed their role detector for educational purposes. Therefore, the tool allows users to provide a role for a variable and check if the tool agrees with them. Although providing a role for a variable is optional, special tags along with the name of the variable must be provided for each variable in a program, otherwise the tool will not consider the variable. During our project, we improved the tool in this and other regards to make the process of detecting roles fully automatic and make it more suitable for our purpose.

In addition, we tuned the tool up in order to improve its performance. For example, in some implementations that used a Do-While loop, the tool did not detect the conditions for a one-way flag role correctly. We fixed the code so that the role was recognized correctly in these situations. As another example, a temporary role typically appears in swap operations, which in turn is commonly used in sorting algorithms. In programs where a swap operation was performed in a separate method, the temporary role was sometimes falsely recognized as a fixed value by the role detector. To solve the problem, we automatically removed the method calls to swap operations in a preprocessing step, and inlined the corresponding swap method body in the target programs. As the result, temporary roles were detected much more accurately. We developed a way to provide the aforementioned required special tags automatically and we improved the tool in many ways in order to get detected roles directly out from the tool for further processing. We also carried out several minor detail modifications.

6. Schemas and Beacons for an Analyzed Set of Algorithms

To show the performance of our method, we applied the method to various algorithms, including sorting, searching, heap, basic tree traversal and graph algorithms. We analyzed a set of implementations of these algorithms to identify their algorithmic schemas (which we also call programming schemas or just schemas) and discern their algorithm-specific beacons. To make it easier to follow the discussion, the pseudo-code examples for these algorithms are presented in Appendix B. These schemas and beacons are the topics of this chapter. We will discuss the data sets and empirical studies in Chapter 7.

6.1 Algorithmic Schemas

6.1.1 Schemas for Sorting Algorithms

We present the schemas for Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort algorithms. These algorithms form two clearly different groups, the first group consisting of Bubble sort, Insertion sort and Selection sort, and the second one of Quicksort and Mergesort. From these, especially the first group has very similar internal structures. This similarity indicates that we should be able to differentiate between closely related algorithms. To complicate the matter further, we discuss the schemas of two student-implemented variations: Insertion sort WS (Insertion With Swap is a variation of Insertion sort where instead of shifting the elements in the inner loop, they are swapped) and Selection sort WILS (Selection With Inner Loop Swap, 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 swapping it once in the outer loop. See Figure 6.1 for this difference). We will get back to these variations in Chapter 7.

```
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. 6.1a)

```
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. 6.1b)

Figure 6.1. Fig. 6.1a shows a typical implementation of Selection sort WILS that uses swap in the inner loop instead of the outer loop. Fig. 6.1b illustrates a typical implementation of a standard Selection sort

We defined the schemas illustrated in Figures 6.2 and 6.3 for the non-recursive (Bubble sort, Insertion sort, Selection sort, Insertion sort WS and Selection sort WILS) and recursive (Quicksort and Mergesort) sorting algorithms respectively[1]. The nesting relationship between the loops and blocks are depicted by the indentations.

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. In Publication III, 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 elements compared in each pass are adjacent. The schemas illustrated in Figures 6.2 and 6.3 further abstract these implementational definitions.

As Figure 6.2 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 (this is the way Insertion sort algorithms and their variations are commonly implemented). 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, as discussed above, in Bubble sort implementations, the two compared elements in the inner loop in each pass are adjacent, whereas

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

this is not the case in Selection sort WILS implementations. We will discuss beacons for sorting algorithms in Subsection 6.2.1.

For more details on the schemas of these sorting algorithms see Publication V and Publication VI.

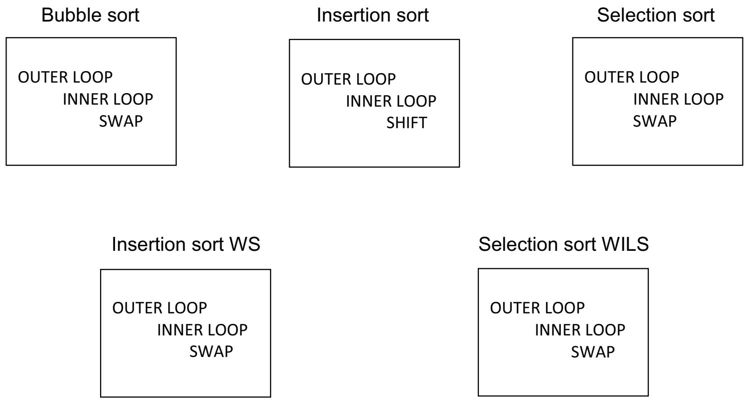


Figure 6.2. Algorithmic schemas for the non-recursive analyzed sorting algorithms

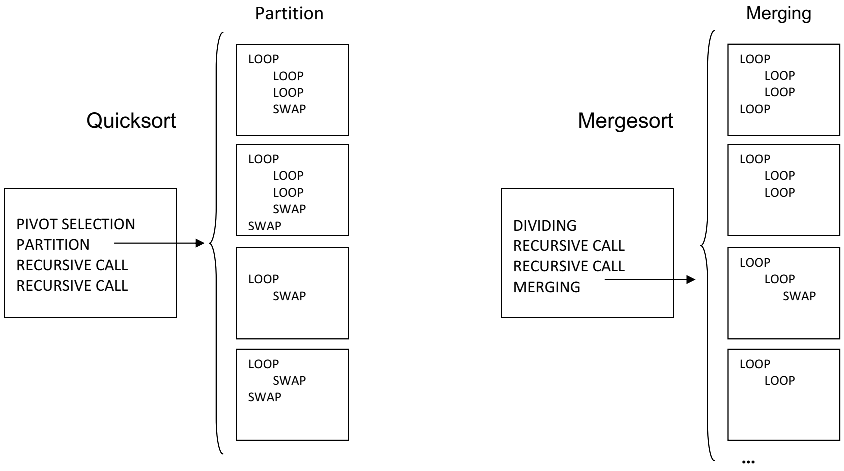
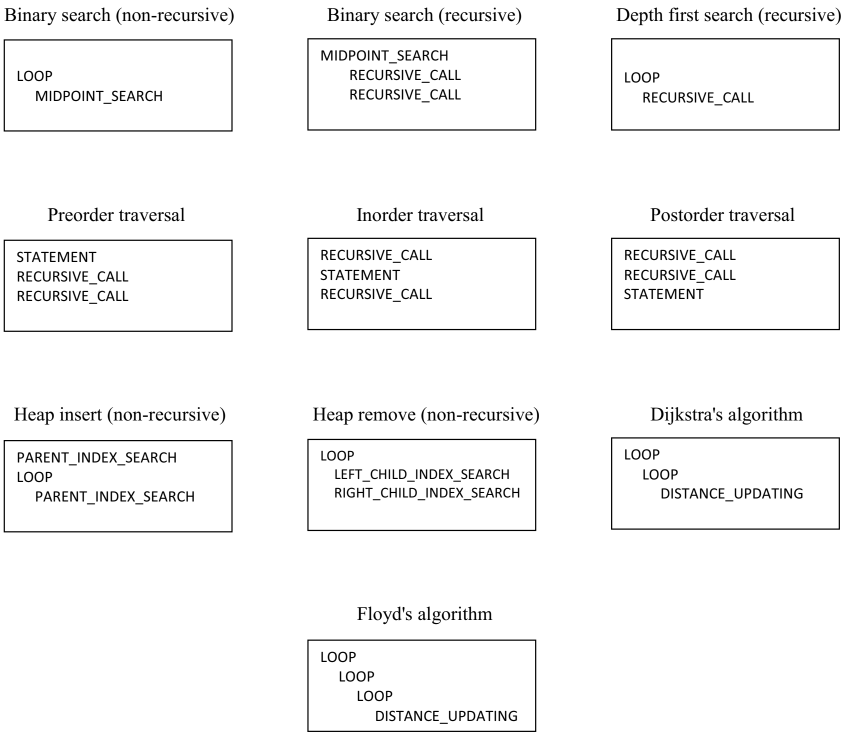


Figure 6.3. Algorithmic schemas for Quicksort and Mergesort. The three dots shown in the Mergesort schemas indicate that merging may have other schemas as well

6.1.2 Schemas for Searching, Heap, Basic Tree Traversal and Graph Algorithms

In a different study, we analyzed implementations of several other algorithms including searching, heap, basic tree traversal and graph algorithms for their schemas. These schemas are illustrated in Figure 6.4.

Many of the algorithms shown in Figure 6.4 have well-established recursive as well as non-recursive versions. For these algorithms, the analyzed version is indicated in the parentheses after their name. Furthermore, in-



Binary search (non-recursive)

```
LOOP
   MIDPOINT_SEARCH
```

Binary search (recursive)

```
MIDPOINT_SEARCH
     RECURSIVE_CALL
     RECURSIVE_CALL
```

Depth first search (recursive)

```
LOOP
   RECURSIVE_CALL
```

Preorder traversal

```
STATEMENT
RECURSIVE_CALL
RECURSIVE_CALL
```

Inorder traversal

```
RECURSIVE_CALL
STATEMENT
RECURSIVE_CALL
```

Postorder traversal

```
RECURSIVE_CALL
RECURSIVE_CALL
STATEMENT
```

Heap insert (non-recursive)

```
PARENT_INDEX_SEARCH
LOOP
   PARENT_INDEX_SEARCH
```

Heap remove (non-recursive)

```
LOOP
  LEFT_CHILD_INDEX_SEARCH
  RIGHT_CHILD_INDEX_SEARCH
```

Dijkstra's algorithm

```
LOOP
   LOOP
      DISTANCE_UPDATING
```

Floyd's algorithm

```
LOOP
   LOOP
      LOOP
         DISTANCE_UPDATING
```

Figure 6.4. The schemas for the analyzed algorithms

dentations indicate the nesting relationship between the loops and blocks. Also note that the schemas of Figure 6.4 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 $LEFT\_CHILD\_INDEX\_SEARCH$ operation once before the loop and again 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 of Figure 6.4, but considered them in the implementation of Aari system. These schemas and the way they are computed are discussed in more detail in Publication VII.

6.1.3 Detecting Schemas

The schemas of Figures 6.2, 6.3 and 6.4 are stored in the knowledge base of the system as illustrated in the figures. The target program is analyzed to find the schemas of the same abstract level. Schemas and subschemas are examined and their elements, including loops, recursion, specific operations, etc., are analyzed. Execution order and nesting relationships between these elements are also investigated. When searching for the schemas and subschemas, details such as type of loops or irrelevant assignments are not taken into consideration. For example, when searching for loops, While loops and For loops are treated the same. As another example, when a swap operation is searched, three consecutive assignment statements that usually constitute a swap operation are examined, ignoring possible assignment statements that may appear before or after these statements. After scanning the program and detecting the schemas and subschemas, they are matched against those from the knowledge base to identify the algorithm in question.

When two or more algorithms have similar schemas at the abstract level illustrated in Figures 6.2, 6.3 and 6.4, we need to investigate these similar schemas at lower level to be able to distinguish between them. Our technique to do so is to examine beacons. For example, as discussed above, the schemas of Bubble sort, Insertion sort WS and Selection sort WILS, which are similar as shown in Figure 6.2, are identified using the following algorithm-specific beacons: in implementations of Insertion sort WS, 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. Likewise, in Bubble sort implementations, the two compared elements in the inner loop in each pass are adjacent, whereas this is not the case in Selection sort WILS implementations. These beacons are recognized by examining how the loop counter variables of the outer and inner loop are initialized, how their values are changed and how the elements of an array are compared within the inner loop. Algorithm-specific beacons are discussed in the next section.

6.2 Beacons

6.2.1 Beacons for Sorting Algorithms

We discerned a set of beacons specific to the sorting algorithms listed in Subsection 6.1.1. These beacons can be utilized to separate implementations of these algorithms from each other, as well as to distinguish between implementations of these algorithms and implementations of algorithms from other fields. They help both in the schema detection process as well as in building a classification tree along with the characteristics discussed in the previous chapter. In the following, we present a list of the beacons that are selected by the C4.5 algorithm for constructing a classification tree for recognizing the sorting algorithms and their variations. Furthermore, the algorithm that each beacon primarily indicates is presented, along with a brief explanation on how each beacon is detected from source code. A complete list of all the computed beacons and their description can be found in Publication VI.

- *MVH* (Most-Wanted Holder): whether the implementation of the algorithm includes a variable appearing in MWH role. MWH mainly indicates the implementations of Selection sort. Existence of a MWH role in code is examined by going through all the roles detected by the role analyzer.

- *One_way_flag*: whether the implementation includes a variable appearing in one-way flag role. This mainly indicates the implementations of Bubble sort WF (Bubble sort With Flag, an optimized version which terminates if no swap is performed in the inner loop). Like for MWH, we can examine the existence of a one_way_flag role from all the roles detected by the role analyzer.

- *Swap_inner_loop*: whether a swap operation is performed in the inner loop of the two nested loops. This mainly indicates the implementations of Bubble sort, Bubble sort WF, Insertion sort WS and Selection sort WILS. This beacon is analyzed by detecting a swap operation and examining that it is located within the inner loop.

- *OIID* (Outer loop Incrementing Inner Decrementing): whether from the two nested loops, the outer loop is incrementing and the inner decrementing. This mainly indicates the implementations of Insertion sort and Insertion sort WS. We can investigate the existence of this beacon using the structural characteristic “Loop counter information”, as described in Table 5.1.

- *IITO* (Inner loop counter Initialized To Outer loop counter): whether from the two nested loops, the inner loop counter is initialized to the value of the outer loop counter. This mainly indicates the implementations of Insertion sort and Insertion sort WS. Existence of this beacon is examined using the structural characteristic “Dependency information” between variables, as described in Table 5.1.

- *Shift_inner_loop*: whether a shift operation is used in the inner loop of the two nested loops. This mainly indicates the implementations of Insertion sort. In order to detect this beacon, we look for a shift operation (i.e., moving the elements of an array to the right until the insertion point is reached) that takes place within the inner loop.

- *Efficient_pivot*: whether the implementation includes efficient pivot selection. This mainly indicates the implementations of Quicksort EP (Quicksort with Efficient Pivot selection, which uses more efficient pivot selection strategy than simply from the left or right end of the given array). This beacon is analyzed by examining that the right-hand side of the pivot’s assignment statement includes investigating the middle index or the median of the first, last and middle items of the given array.

6.2.2 Beacons for Searching, Heap, Basic Tree Traversal and Graph Algorithms

For the analyzed searching, heap, basic tree traversal and graph algorithms we found the following set of beacons that are used by the corresponding classification trees to identify implementations of these algorithms. For a list of all the computed beacons see Publication VII.

- *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. To examine this beacon we analyze the right-hand side of the assignment statement within a loop to make sure that it includes searching for the midpoint of an array, for example, $mid = (low + high)/2$.

- *MPBR* (MidPoint Before Recursion): whether the implementation includes searching midpoint before two recursive calls. This mainly indicates implementations of recursive binary search algorithm. This is investigated like for the MPSL beacon, but looking for it to occur before two recursive calls rather than within a loop.

- *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. We examine that the two recursive calls occur one after the other; in inorder traversal algorithm, there exist an statement between the two recursive calls.

- *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. Detecting this beacon includes identifying computing the index of the parent of a given node with index i, which is $i/2$.

- *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. This beacon is detected by analyzing $LEFT\_CHILD\_INDEX\_SEARCH$ and $RIGHT\_CHILD\_INDEX\_SEARCH$, which for a node with index i are $2i$ and $2i+1$, correspondingly. 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[2].

- *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. Existence of this beacon is examined by analyzing the occurrence of the operation $DISTANCE\_UPDATING$ within three nested loops. More specifically, we investigate 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.

Some of these beacons may seem generic and likely to occur in implemen-

[2] 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 for detecting the beacon and the corresponding schema as well.

tations of the other algorithms as well. For example, one could say that searching midpoint of an array (in the MPSL and MPBR beacons found in non-recursive and recursive binary search algorithms) may also exist in implementations of Mergesort (in the code for dividing the array into two halves) and Quicksort (for selecting the middle index of the array as a pivot). Likewise, it could be argued that "two sequential recursive calls" (the beacon TSRC found in preorder and postorder traversal algorithms) are also present in Mergesort and Quicksort implementations. However, it should be noted that these beacons are closely related to the schemas illustrated in Figure 6.4 and their values are set to true only when these schemas are detected. On the other hand, these schemas depict implementations of the corresponding algorithms at a very high-level of abstraction and do not show the code related to the details that could be utilized to detect them. In the implementation of Aari system, various checks have been done to prevent the values of the aforementioned beacons to be falsely set to true in the case of the other algorithms. These checks are essential in order to identify true value of those beacons that may seem to be shared by several algorithms, as discussed above. For example, in the case of the MPSL beacon, we examine that the algorithm is not recursive and that the midpoint search occurs within a loop. For the MPBR beacon, the relationship between the block of the midpoint search and the block of the recursive call is checked. Similarly, for the TSRC beacon, by investigating the code fragment before and after the recursive calls, we make sure that the schemas of Mergesort and Quicksort (as shown in Figure 6.3) are not in question before setting the value of this beacon to true.

7. Empirical studies and Results

In this chapter, we summarize the empirical studies conducted to evaluate the performance of our methods throughout their development processes. For each empirical study, we only present brief highlights of the results, referring the reader to the publication where the results are published. We first present an overview of the data sets used in the empirical studies, as well as a brief description of the objectives and contributions of the empirical studies and the way they are related to each other.

The layout of the tables that are used for presenting the results of the empirical studies in this chapter are not necessarily identical to those that present the same results in the publications. We have changed the appearance of some tables in order to make them consistent and give a logical and understandable summary here.

7.1 An Overview of the Data Sets and Empirical studies

7.1.1 The Data Sets

We collected three different data sets for our empirical studies (MS, SUB, and MIX). These data sets are shown in the second column of Table 7.1. The first data set was collected from various learning resources including textbooks and the Web, and a few instances were from students' submissions. It mainly consists of the implementations of five basis sorting algorithms, as indicated in Table 7.1. The category "Others" includes the implementations of other sorting algorithms, such as Heapsort, Shellsort and the hybrid implementations of, for example, Quicksort-Insertion sort, as well as the implementations of algorithms from other fields, such as binary search, etc. The second data set was collected from genuine students' sorting algorithm implementations in a first year data structures and algo-

First data set

Algorithm	MS	MS_1	MS_2	MS_3
Bubble sort	41	41	41	Bubble: 26 Bubble WF: 15
Insertion sort	52	52	Insertion: 43 Insertion WS: 9	Insertion: 43 Insertion WS: 9
Selection sort	43	43	Selection: 43 Selection WILS: 0	Selection: 43 Selection WILS: 0
Mergesort	34	34	34	34
Quicksort	39	39	39	Quicksort: 22 Quicksort EP: 17
Others	78	-	-	
Total	287	209	209	209

Second data set

Algorithm	SUB	SUB_1	SUB_2
Bubble sort	29	29	Bubble: 14 Bubble WF: 15
Insertion sort	17	17	17
Insertion WS	10	10	10
Selection sort	36	36	36
Selection WILS	13	13	13
Mergesort	20	20	20
Quicksort	34	34	Quicksort: 15 Quicksort EP: 19
Others	33	-	-
Total	192	159	159

Third data set

Algorithm	MIX	Abbreviation
Non-recu. BinSearch	36	NBS
Recursive BinSearch	13	RBS
Depth First Search	15	DFS
Inorder Traversal	23	InT
Preorder Traversal	24	PreT
Postorder Traversal	22	PostT
Heap Insertion	22	HeapI
Heap Remove	21	HeapR
Dijkstra's algorithm	23	Dijkstra
Floyd's algorithm	23	Floyd
Total	222	

Table 7.1. The data sets and their subsets used in different empirical studies. In the third data set, depth first search algorithm is recursive and heap insertion and remove are non-recursive. Insertion WS: Insertion With Swap, Selection sort WILS: Selection With Inner Loop Swap, Bubble sort WF: Bubble sort With Flag, Quicksort EP: Quicksort with Efficient Pivot selection. MS: Multi-Source algorithm implementations (collected from textbooks and websites), MS_1: Multi-Source sorting algorithm implementations collected from MS, MS_2: Multi-Source sorting algorithm implementations including Insertion WS and Selection WILS variations collected from MS_1, MS_3: Multi-Source sorting algorithm implementations including Insertion WS, Selection WILS, Bubble WF and Quicksort EP variations collected from MS_2, SUB: Submissions (authentic students' submissions), SUB_1: Submissions sorting algorithm implementations collected from SUB, SUB_2: Submissions sorting algorithm implementations including Bubble WF and Quicksort EP variations collected from SUB_1, MIX: algorithms from different fields (collected from textbooks and websites)

rithms course. "Others" consists of the implementations of other standard algorithms (Shellsort and Heapsort), the implementations of less-known inefficient sorting algorithms (such as Bozo sort, Bogosort and Gnome sort) and the implementations of student-made inefficient algorithms (see Publication III for more details on these variations). Finally, we collected the third data set from textbooks and websites.

Table 7.1 also shows several subsets of these data sets we used in different empirical studies (the third, fourth and fifth columns). These subsets are formed by further analyzing their corresponding main data sets. There are two reasons for using these subsets: first, to select a group of algorithms we want to examine, for example, MS_1 (for the abbreviations, see the caption of the table) selects only the implementations of the five sorting algorithms from the data set MS; second, to differentiate between the variations of the algorithms that we need to analyze. These variations are Insertion WS (Insertion With Swap, is a variation of Insertion sort that swaps the elements in the inner loop, instead of shifting them) and Selection sort WILS (Selection With Inner Loop Swap, 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 swapping it once in the outer loop). In addition, we distinguish between two optimized sorting algorithms and their basic versions: Quicksort EP (Quicksort with Efficient Pivot selection, which uses more efficient pivot than simply from the left or right end of the given array) and Bubble sort WF (Bubble sort With Flag, an optimized version which uses a boolean value to indicate whether a swap operation is performed in the inner loop, and terminates if not). The aim is to identify these variations and versions in students' work and give useful feedback on them. For more details, see Publication VI. As an example, the subset MS_3 further distinguishes these variations and optimized versions from the data set MS, as Table 7.1 illustrates.

We gathered these three main data sets and formed the related subsets presented in Table 7.1 during our research in order to evaluate the performance of our methods, since no data set exists or is publicly available for this purpose.

More details on the first, second and third data sets can be found in Publication I, Publication III and Publication VII respectively.

7.1.2 The Publications and Empirical studies

Table 7.2 outlines the topic of the publications and summarizes the related empirical studies. For each empirical study, the data set and the main contribution is presented. In the following sections, we present the main results of each empirical study and explain how it improves the previous one (where applicable). The results of the empirical study discussed in

Pub.	Data	Description
I	MS	Analysis of sorting algorithm implementations and manual selection of the best characteristics and beacons. Developing a method to recognize implementations of a testing set based on the selected characteristics and beacons. Using a hold-out method to evaluate the performance of the method.
II	MS_1	Using the C4.5 algorithm to automatically select the best characteristics and beacons and build a decision tree to classify sorting algorithms. Evaluating the estimated accuracy of the classification by leave-one-out cross-validation.
III	SUB	Categorizing student-implemented sorting algorithms and identifying their inefficient variations in order to develop a tool that can automatically recognize these variations and give feedback to students on their problematic solutions.
IV	MS_1+ SUB	Using the data set MS_1 to construct a classification tree (as described in Publication II) and recognizing students' implementations of sorting algorithms (SUB) by the tree.
V	MS_2+ SUB_1	Developing a method to recognize algorithms by detecting algorithmic schemas from source code. Applying the method to sorting algorithms and their variations and evaluating the performance of the method using a combined data set. Using "Others" from MS and SUB data sets to test how the method performs on the other algorithm implementations.
VI	MS_3+ SUB_2	Combining the SDM and CLM to enhance the reliability and performance. Developing techniques to identify the optimized versions of the sorting algorithms in addition to their variations, in order to give useful feedback. Evaluating the estimated accuracy of the classification using leave-one-out cross-validation.
VII	MIX	Extending the SDM and CLM to searching, tree traversal, heap and graph algorithms. Defining a set of schemas and beacons for recognizing these algorithms and evaluating the performance of the SDM and CLM.
UP	MIX+ MS_1+ SUB	Using a combined learning data (MIX+MS_1) to construct a classification tree, evaluating its performance using leave-one-out cross-validation technique and recognizing previously unseen students' sorting implementations in SUB data set using the tree (like in Publication IV).

Table 7.2. A summary of the publications and the related empirical studies, their objectives/contributions, the data set(s) they use and the way they are related to each other. UP: unpublished

Publication VII are not reported in detail in that paper due to the space limitation. We discuss them at the end of Section 7.5. Moreover, the final empirical study denoted by UP in Table 7.2 is not included in the publications and thus will be discussed in more detail in Section 7.6.

To evaluate the classification performance, we use the following widely used metrics: True Positive (TP, implementations that are correctly assigned to a class), False Positive (FP, implementations that are incorrectly assigned to a class) and False Negative (FN, implementations that belong to a class but not assigned to it). Moreover, based on these metrics, we discuss the results in terms of precision (proportion of correctly recognized positive case implementations from all implementations recognized as positive cases, i.e., precision = TP/(TP+FP)), recall or true positive rate (proportion of correctly recognized positive case implementations from all implementations that should have been recognized as positive cases, i.e., recall = TP/(TP+FN)) and False Negative Rate (proportion of incorrectly rejected implementations from all implementations that should have been recognized as positive cases, i.e., FNR = FN/(TP+FN).

True Negative (TN) cases are instances that are correctly rejected for each algorithm class. Thus, the proportion of TN cases compared to TP, FP and FN cases is excessively large. This is considered as a problematic situation in literature [95]. Moreover, in some applications of classification, the number of TN cases may remain unknown (e.g., when identifying web documents based on queries provided to a web search engine) [95]. Instead, for example, precision and recall are widely used since their values, as discussed above, do not depend on TN cases. In the results presented in this chapter, TN cases are not as informative as the aforementioned metrics and therefore we do not discuss them here.

In addition, we use the *confusion matrix* to discuss the incorrectly identified implementations in more detail. The confusion matrix is an $N \times N$ matrix, where each instance I_{ij} indicates the instance that belongs to class I_i, but is classified as class I_j [96]. The instances located on the diagonal are classified correctly.

For the explanations of the abbreviations of the beacons discussed in this chapter see Section 6.2.

7.2 Manual Analysis and the Classification Tree Constructed by the C4.5 Algorithm

7.2.1 Manual Analysis

We analyzed the implementations of 70 sorting algorithms of the MS data set for Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort, as shown in Table 7.1. We discerned a set of characteristics and beacons (see Section 5.2) and based on our judgment, decided how to use them to help us distinguish between implementations of these algorithms. We used the numerical characteristics to filter out the implementations of the testing set that have greater or smaller values of these characteristics than those of the analyzed 70 implementations. Moreover, we used the beacons to differentiate between those implementations that pass this filter. We implemented a tool that automatically computed all the characteristics and beacons and assigned a type for a given implementation based on the logic discussed above. We tested the performance of the method using 217 implementations of sorting and other algorithms (see Section 7.1). 86% of the implementations of the testing set were recognized correctly. More details on this study can be found in Publication I.

7.2.2 The Classification Tree Constructed by the C4.5 Algorithm

It is very difficult to manually select the best distinguishing factors, even in a seemingly simple task of classifying the five sorting algorithms. This is illustrated by our next study explained in Publication II. In this empirical study, we automatized the process of constructing a classification tree using the C4.5 algorithm[1]. Since our objective was to examine what kind of classification tree the C4.5 algorithm builds for the five sorting algorithms and how accurately this tree performs, as our learning data we used MS_1 data set that includes only the implementations of these algorithms. The C4.5 algorithm selects the best splits and builds a more optimal and understandable classification tree, shown is Figure 7.1.

Evaluated by leave-one-out cross-validation method, the performance estimate of the classification is 98,1% (i.e., 205 implementations of all the total 209 implementations of the data set are classified correctly). In Publication II, we defined FP as the cases where an algorithm implementation not belonging to the members of the target set of the five sorting algorithms, is incorrectly recognized as one of them. As the result of this definition, since all the implementations of the data set belong to one of the five sorting algorithms, FP cases did not occur in the empirical study (see Table 7.3 from Publication II, where false positives are shown as FP´ to signify this definition). However, if we use the definition of FP cases discussed in Subsection 7.1.2, that is, if we consider as FP cases also those implementations of the target set that are falsely recognized as another

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

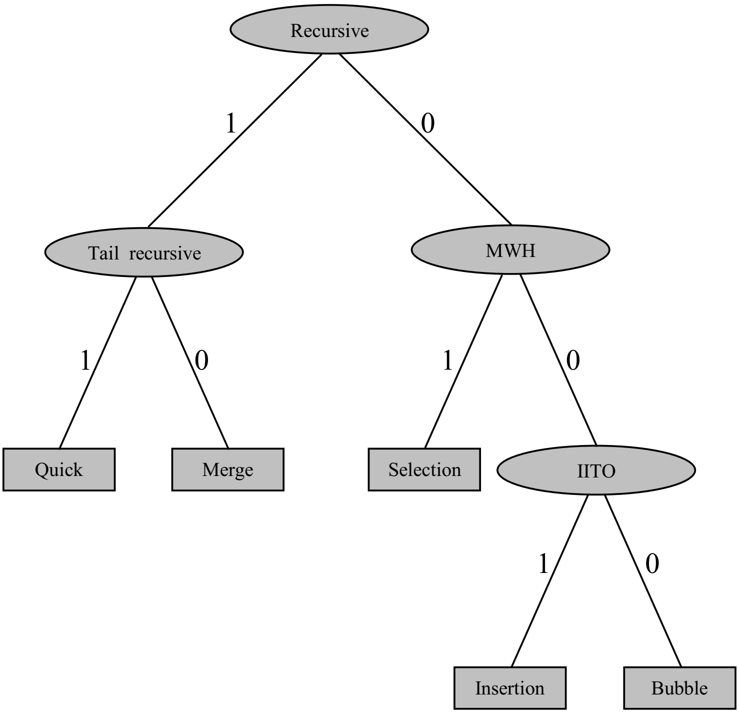


Figure 7.1. The classification tree constructed by the C4.5 algorithm for recognizing sorting algorithm implementations

class of the target set, then the value of FP cases would be 4, since four Insertion sort implementations are falsely recognized as Bubble sort. Thus, the results presented by the commonly used performance measures would be as shown in Table 7.4. Moreover, as Table 7.3 shows, in Publication II, we defined a TN case as correctly rejecting an implementation which does not belong to any member of the target set. This definition resulted in the value of the TN cases to be zero in the empirical study as well (shown as TN´ in Table 7.3). For more details, see Publication II.

Note that the materials used for evaluating the accuracy of the classification presented in this section and Section 7.2.1 (reported in Publication I) are different. In Publication II, we have presented a comparison between the accuracy of these classifications achieved by using the same material.

7.3 Students' Sorting Algorithm Implementations, a Categorization and Automatic Recognition

We collected the authentic students' sorting algorithm implementations in a first year data structures and algorithms course (the data set denoted by SUB in Table 7.1). We used it for two purposes: 1) examining what

Algorithm	TP	FP´	FN	TN´	FNR	Recall	TNR	FPR	Total
Bubble	41	0	0	0	0	1	0	0	41
Insertion	48	0	4	0	0.077	0.923	0	0	52
Selection	43	0	0	0	0	1.0	0	0	43
Merge	34	0	0	0	0	1	0	0	34
Quick	39	0	0	0	0	1	0	0	39
Total	205	0	4	0	-	-	-	-	209

Table 7.3. The value of different metrics indicating the estimated classification accuracy using leave-one-out cross-validation (from Publication II). TNR = TN´ / (TN´ + FP´), FPR = FP´ / (TN´ + FP´). See the text for the definition of FP´ and TN´

Algorithm	TP	FP	FN	FNR	Recall	Precision	Total
Bubble	41	4	0	0	1	0.911	41
Insertion	48	0	4	0.077	0.923	1	52
Selection	43	0	0	0	1	1	43
Merge	34	0	0	0	1	1	34
Quick	39	0	0	0	1	1	39
Total	205	4	4	-	-	-	209

Table 7.4. Different metrics used for evaluating the estimated performance of the classification using leave-one-out cross-validation

variations of sorting algorithms students implement in order to discover their problematic solutions and insufficient understandings. This helps us further develop our method to detect these variations and give automatic feedback on them, and 2) testing Aari system that uses the classification tree of Figure 7.1 with a previously unseen data set.

7.3.1 Categorizing the Variations

Manual categorization of students' submissions revealed that they implement many inefficient variations. Two of these variations are Insertion sort WS and Selection sort WILS. We explained these variations in Subsection 7.1.1. Based on these results, we developed techniques to recognize these variations (both with the SDM, as well as the CLM). The submissions were collected in two rounds: at the beginning of the course before the students received any instruction on sorting algorithms, and after taking a lecture on sorting algorithms. Table 7.5 summarizes the types of the submissions analyzed manually separately for the first round, second round and in total. More details on this study, such as what types of algorithms the category "Others" includes, can be found in Publication III. The main contribution of this article is manual analysis of students' sorting algorithm implementations in order to identify and categorize their problematic solutions.

Algorithm	Round 1 (%)	Round 2 (%)	Total (%)
Bubble sort	25 (22)	4 (5)	29 (15)
Insertion sort	8 (7)	9 (11)	17 (9)
Selection sort	30 (27)	6 (8)	36 (19)
Mergesort	4 (4)	16 (20)	20 (10)
Quicksort	2 (2)	32 (40)	34 (18)
Inefficient variations	20 (18)	3 (4)	23 (12)
Others	23 (21)	10 (13)	33 (17)
Total	112	80	192

Table 7.5. Sorting algorithms implemented by the students in the first and second round. The values are computed manually. The percentages show the results with respect to the number of total submitted implementations in the corresponding round. For example, there are 25 implementations of Bubble sort algorithm in the first round, that is, 22% of all the implementations in this round (which is 112)

Algorithm	Round 1	Correct (%)	Round 2	Correct (%)	Total	Correct (%)	False (%)
Bubble sort	25	24 (96)	4	4 (100)	29	28 (97)	1 (3)
Insertion sort	8	8 (100)	9	8 (89)	17	16 (94)	1 (6)
Selection sort	30	30 (100)	6	6 (100)	36	36 (100)	0 (0)
Mergesort	4	1 (25)	16	11 (69)	20	12 (60)	8 (40)
Quicksort	2	2 (100)	32	32 (100)	34	34 (100)	0 (0)
Inefficient var.	20	11 (55)	3	2 (67)	23	13 (57)	10 (43)
Others	23	3 (13)	10	2 (20)	33	5 (15)	28 (85)
Total	112	79 (71)	80	65 (81)	192	144 (75)	48 (25)

Table 7.6. Students' sorting algorithm implementations (a new unseen data set) recognized automatically by Aari system. For the first and second round, the percentages show the results with respect to the number of the implementations of each algorithm in the corresponding round. For example, the number of the implementations of Bubble sort algorithm in the first round is 25, from which 24 implementations are correctly recognized, that is, 96 percent of the 25 implementations

7.3.2 Automatic Recognition

We tested Aari system that used the classification tree illustrated in Figure 7.1, with student's sorting algorithm implementations as a previously unseen data set (i.e., a data set that have not been used in constructing the classification tree). The results of this automatic recognition, summarized in Table 7.6, show that Aari performs very good with implementations of those types of sorting algorithms that it has been trained to recognize. The implementations of these algorithms are recognized with an average accuracy of about 90%. When considering all the implementations, Aari achieved an overall accuracy of 71% and 81% for the first and second round respectively. Note that in Table 7.6, the implementations of inefficient variations are considered as recognized correctly if they are recognized as the standard corresponding algorithms, that is, as Selection sort and Insertion sort. For further details, see Publication IV.

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

Table 7.7. The results of detecting algorithmic schemas for sorting algorithms and their variations

7.4 Using the SDM and CLM for Recognizing Sorting Algorithms and Their Variations

7.4.1 The SDM

We developed another method for algorithm recognition that is based on the concept of algorithmic schemas. The theoretical background of the method is discussed in Chapter 3. The CLM considers the whole of the given program as the algorithmic code and computes the characteristics from the whole program. The aim of the SDM is to search for the fragments of code that implement the algorithm in question and select them for further analysis, leaving the irrelevant code out of the process.

We applied the method to the five basic sorting algorithm implementations discussed above and their two variations that we found by analyzing the students' implementations, namely Insertion WS and Selection WILS. The schemas for these algorithms are presented in Subsection 6.1.1, Figures 6.2 and 6.3, and the beacons in Subsection 6.2.1. To differentiate between the algorithms and variations with the same schemas, we use algorithm-specific beacons. We used the data sets MS_2 and SUB_1 (see Table 7.1) to evaluate the performance of the method. Table 7.7 summarizes the results. In addition, we used the implementations of other algorithms from MS and SUB data sets (denoted by "Others" in Table 7.1) to test how many implementations would be falsely recognized as one of the sorting algorithms. From the 111 implementations of other algorithms (78 implementations from MS data set and 33 from SUB data set) 10 implementations (i.e., 9 percent) were falsely detected as including one of the specified algorithmic schemas.

More details on this study, such as schemas and results, can be found in Publication V.

Algorithm	TP	FP	FN	FNR	Recall	Precision	Total
Bubble sort	38	4	2	0.050	0.950	0.905	40
Insertion sort	60	0	0	0	1	1	60
Selection sort	79	0	0	0	1	1	79
Mergesort	52	1	2	0.037	0.963	0.981	54
Quicksort	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 sort WF	30	1	0	0	1	0.968	30
Quicksort EP	36	1	0	0	1	0.973	36
Total	357	11	11	-	-	-	368

Table 7.8. The estimated classification accuracy achieved by leave-one-out cross-validation method for recognizing sorting algorithms and their variations using the CSC

7.4.2 The CSC

In our next study, we combined the SDM and CLM and developed a method that first detects the algorithmic schemas from the given program and then computes the characteristics and beacons only from the code related to these schemas, rather than from the whole program. Because the irrelevant code is not considered for further analysis, this improves the reliability of the CLM. A discussion on how the CSC works is presented in Chapter 5 and particularly Section 5.1.

We applied this CSC to the five sorting algorithms and their variations. In addition, we developed techniques for detecting optimized versions of two sorting algorithms, Bubble WF and Quicksort EP (see Subsection 7.1.1 for the explanation), and considered these versions in our empirical study as well. This would allow to provide useful feedback to students on their implementations. For this empirical study, we used the data sets MS_3 and SUB_2. We evaluated the performance estimate of the classification using leave-one-out cross-validation method. The estimated classification accuracy is 97.0%, that is, from the 368 instances of the data sets, the number of correctly classified instances is 357 and the number of incorrectly classified instances is 11 (i.e., 3.0%). Table 7.8 summarizes the results.

For more details, especially the constructed classification tree, see Publication VI.

7.5 Using the CSC for Recognizing Algorithms from Other Fields

Having good results from applying the CSC to sorting algorithms and their variations, we extended the methods to cover algorithms from other fields: searching, heap, basic tree traversal and graph algorithms. We collected

Algorithm	NBS	RBS	DFS	InT	PreT	PostT	HeapI	HeapR	Dijkstra	Floyd
NBS	35	0	0	1	0	0	0	0	0	0
RBS	0	13	0	0	0	0	0	0	0	0
DFS	0	0	14	0	0	0	0	0	1	0
InT	0	0	0	23	0	0	0	0	0	0
PreT	0	0	0	1	23	0	0	0	0	0
PostT	0	0	0	0	0	22	0	0	0	0
HeapI	0	0	1	0	0	0	21	0	0	0
HeapR	0	0	0	0	0	0	0	21	0	0
Dijkstra	0	0	1	0	0	0	0	0	22	0
Floyd	0	0	0	0	0	0	0	0	1	22

Table 7.9. The confusion matrix that shows how each implementation is recognized using leave-one-out cross-validation; the row headings indicate the actual type of each algorithm and the column headings indicate what type it was recognized as. See MIX data set in Table 7.1 for the explanations of the abbreviations

222 implementations of 10 different algorithms (the data set denoted by MIX in Table 7.1). We defined the schemas and beacons related to these algorithms. These are discussed in Subsections 6.1.2 and Subsection 6.2.2 respectively.

We evaluated both the performance of the SDM and CLM. The schemas were detected with an average accuracy of 94,1%. The estimated accuracy of the classification measured by leave-one-out cross-validation method was 97,3%. Figures 7.2 shows how accurately the implementations of the data set are recognized by the SDM. Figures 7.3 illustrates the results for the CLM. Details, including the corresponding decision tree, can be found in Publication VII.

In addition to these, we present the confusion matrix of Table 7.9 to discuss the correctly and falsely classified implementations in more detail. In the table, the implementations positioned on the diagonal are recognized correctly. As an example, all the implementations of recursive binary search (RBS) are recognize correctly, while one implementation of non-recursive binary search (NBS) is falsely recognized as inorder tree traversal algorithm (InT).

Finally, we use the performance evaluation measures shown in Table 7.10 to discuss the results in further details. As can be seen from the table, larger numbers of false positive and false negative cases indicate the poorer value of precision and recall. For the definition of these evaluation measures, see Subsection 7.1.2.

The results presented in this section suggest that the proposed method for algorithm recognition can be extended to other fields of algorithm with high accuracy.

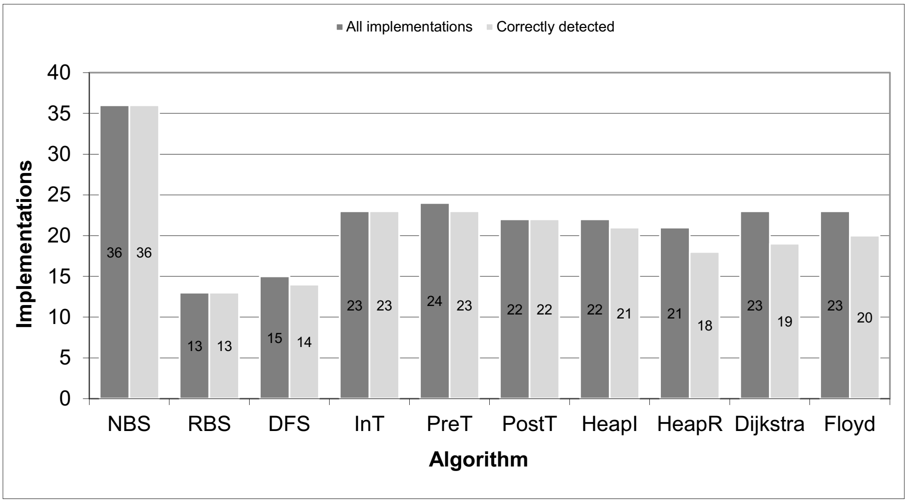


Figure 7.2. The results of detecting the algorithmic schemas for the implementations of the data set MIX (see Table 7.1)

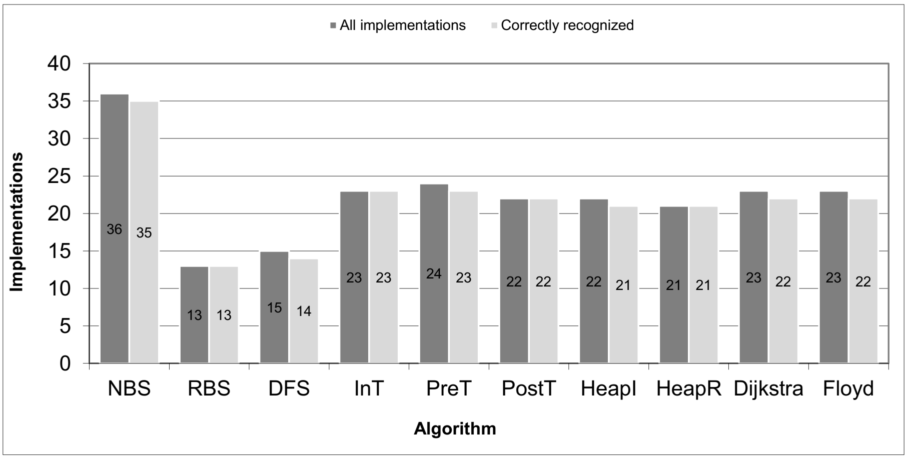


Figure 7.3. Correctly identified implementations of the data set MIX resulted from evaluating the performance estimate of the classification by leave-one-out cross-validation

Algorithm	TP	FP	FN	FNR	Recall	Precision	Total
Non-recursive BinSearch	35	0	1	0.028	0.972	1	36
Recursive BinSearch	13	0	0	0	1	1	13
Depth first search	14	2	1	0.067	0.933	0.875	15
Inorder tree traversal	23	2	0	0	1	0.920	23
Preorder tree traversal	23	0	1	0.042	0.958	1	24
Postorder tree traversal	22	0	0	0	1	1	22
Heap insertion	21	0	1	0.045	0.955	1	22
Heap remove	21	0	0	0	1	1	21
Dijkstra’s algorithm	22	2	1	0.043	0.957	0.917	23
Floyd’s algorithm	22	0	1	0.043	0.957	1	23
Total	216	6	6	-	-	-	222

Table 7.10. The value of the performance evaluation measures for classifying the instances of the MIX data set evaluated by leave-one-out cross-validation method

7.6 Building a Classification Tree of Sorting and Other Algorithms for Recognizing Students' Sorting Algorithm Implementations

In this section, we describe an empirical study on using the implementations of sorting, searching, heap, tree traversal and graph algorithms (i.e., the combined data set MIX + MS_1) to build a classification tree and evaluating the performance estimate of this classification using leave-one-out cross-validation method. This would further show the generalizability of the method within the domain of basic algorithms covered in computer science education. Moreover, in our final empirical study, we used the classification tree to recognize the authentic students' sorting algorithm implementations (the data set SUB). A similar empirical study is carried out and reported in Publication IV with a simpler classification tree constructed only by the implementations of the five sorting algorithms (see Subsection 7.3.2). The purpose of this empirical study is to examine how accurately the student implementations will be recognized with a more complex classification tree. This would also show the ability of the method to be extended. This empirical study is not reported in the publications and thus the results will be covered in a greater detail.

7.6.1 The Decision Tree and Classification Accuracy

Figure 7.4 shows the constructed classification tree, with the results of evaluating the accuracy of the classification presented in Table 7.11. The last column of the table indicates the algorithms as which the falsely classified implementations are recognized. Table 7.12 shows more detailed results presented by commonly used performance evaluation measures, including recall and precision. As can be seen, the classification accuracy remains high, even though a number of algorithms from different fields are covered.

7.6.2 Recognizing the Students' Implementations

Finally, we used the classification tree of Figure 7.4 to recognize the authentic student-implemented sorting algorithms (the data set denoted by SUB in Table 7.1). We performed the evaluation for each round separately. The process in this empirical study corresponds to the steps illustrated in Figure 5.2.

Table 7.13 summarizes the results for each round and in total. For the

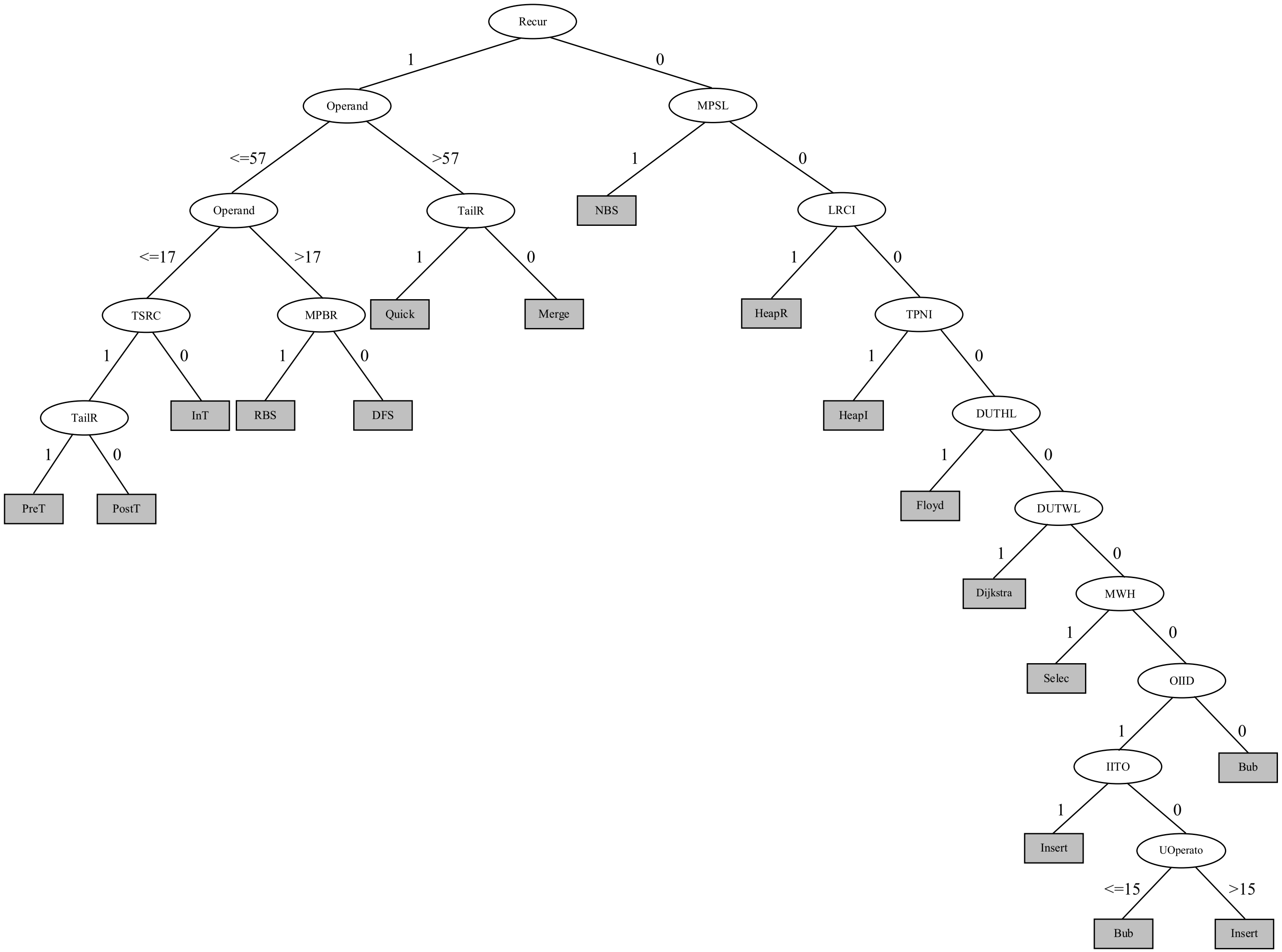


Figure 7.4. The classification tree constructed for recognizing sorting, searching, heap, basic tree traversal and graph algorithms using the data sets MIX and MS_1. Recur: recursive, TailR: tail recursive, UOperato: number of unique operators, Bub: Bubble sort, Insert: Insertion sort, Selec: Selection sort. For the abbreviations of the beacons and other algorithms, see Subsection 6.2.2 and Table 7.1 respectively

Algorithm	Correct (%)	False (%)	Total	Falsely recognized as
Bubble sort	41 (100)	0 (0)	41	-
Insertion sort	51 (98,1)	1 (1,9)	52	Bubble sort
Selection sort	42 (97,7)	1 (2,3)	43	Bubble sort
Mergesort	34 (100)	0 (0)	34	-
Quicksort	38 (97,4)	1 (2,6)	39	Depth first search
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	Mergesort
Inorder tree traversal	23 (100)	0 (0)	23	-
Preorder tree traversal	23 (95,8)	1 (4,2)	24	Inorder tree traversal
Postorder tree traversal	22 (100)	0 (0)	22	-
Heap insertion	21 (95,5)	1 (4,5)	22	Depth first search
Heap remove	19 (90,5)	2 (9,5)	21	Heap insertion
Dijkstra's algorithm	21 (91,3)	2 (8,7)	23	Bubble, Selection
Floyd's algorithm	22 (95,7)	1 (4,3)	23	Dijkstra's algorithm
Total	420 (97,4)	11 (2,6)	431	-

Table 7.11. The results of evaluating the estimated classification accuracy. The last column indicates the algorithms as which the falsely classified implementations are recognized

Algorithm	TP	FP	FN	FNR	Recall	Precision	Total
Bubble sort	41	3	0	0	1	0.932	41
Insertion sort	51	0	1	0.019	0.981	1	52
Selection sort	42	1	1	0.023	0.977	0.977	43
Mergesort	34	1	0	0	1	0.971	34
Quicksort	38	0	1	0.026	0.974	1	39
Non-recursive BinSearch	36	0	0	0	1	1	36
Recursive BinSearch	13	0	0	0	1	1	13
Depth first search	14	2	1	0.067	0.933	0.875	15
Inorder tree traversal	23	1	0	0	1	0.958	23
Preorder tree traversal	23	0	1	0.042	0.958	1	24
Postorder tree traversal	22	0	0	0	1	1	22
Heap insertion	21	2	1	0.045	0.955	0.913	22
Heap remove	19	0	2	0.095	0.905	1	21
Dijkstra's algorithm	21	1	2	0.087	0.913	0.955	23
Floyd's algorithm	22	0	1	0.043	0.957	1	23
Total	420	11	11	-	-	-	431

Table 7.12. Different measures used for evaluating the estimated performance of the classification using leave-one-out cross-validation

Algorithm	Round 1	Correct (%)	Round 2	Correct (%)	Total	Correct (%)	False (%)
Bubble sort	25	24 (96)	4	4 (100)	29	28 (97)	1 (3)
Insertion sort	8	7 (88)	9	9 (100)	17	16 (94)	1 (6)
Selection sort	30	30 (100)	6	6 (100)	36	36 (100)	0 (0)
Mergesort	4	1 (25)	16	14 (88)	20	15 (75)	5 (25)
Quicksort	2	2 (100)	32	32 (100)	34	34 (100)	0 (0)
Inefficient var.	20	10 (50)	3	2 (67)	23	12 (52)	11 (48)
Others	23	4 (17)	10	2 (20)	33	6 (18)	27 (82)
Total	112	78 (70)	80	69 (86)	192	147 (77)	45 (23)

Table 7.13. The results of recognizing the student-implemented sorting algorithms (as a new unseen data set)

first and second round, the percentages show the results with respect to the number of the implementations of each algorithm in the corresponding round. As an example, the number of the implementations of Insertion sort in the first round is 8, from which 7 implementations are correctly recognized, that is, 88 percent of the 8 implementations. Again, a comparison between these results and the results shown in Table 7.6 (reported in Publication IV) shows that although in this empirical study, the classification tree is much more complex and is constructed using the implementations from various fields of algorithms, the estimated accuracy of the classification remains practically the same.

It should be noted that since the system is not trained to recognize the algorithms of the category “Others”, these algorithms cannot be identified. If “Others” includes algorithms that should be classified by the system as belonging to a class but are not, they would be considered as false negative cases. Similarly, if some algorithms from “Others” are falsely recognized as belonging to a class, they would be false positive cases.

8. Discussion and Conclusions

This chapter first discusses the issues related to the proposed method, such as its applications. This is followed by a summary of what has been done in terms of the research questions posed in Section 1.2 and directions for future work. A brief discussion on validity issues concludes this thesis.

8.1 Discussion

This thesis introduces techniques developed for recognizing basic algorithms and their variations within the scope of computer science education. Several techniques are developed and evaluated including 1) manual analysis of the implementations of a learning data in order to identify a set of characteristics and beacons to differentiate between algorithms, 2) applying machine learning methods to select the best discriminators to distinguish between the algorithms and build an automatic classification method, and evaluating the estimated performance of the classification, 3) analyzing authentic student-implemented algorithms in order to discern the variations students implement (for sorting algorithms), 4) developing a method based on schemas for identifying the sorting algorithms and their variations, 5) developing a method that combines the SDM and CLM in order to achieve a more reliable performance, and 6) extending the methods and developing the similar techniques for other fields of algorithms in order to demonstrate their potential.

8.1.1 Applications of the Method

The proposed method could support a teacher in assessing students' submissions by examining whether students have implemented the required algorithm. Although the results are not 100% accurate - due to the statistical nature of the method - assessing a major part of the submissions is of a great help in the burden of marking students' assignments especially in large courses. This would allow the teacher to focus only on the solutions which do not conform to the specification, rather than assessing all the submissions. In this context, false positive cases are more difficult to track and thus more serious problems. If an implementation is incorrectly classified as, for example, a Quicksort (a false positive case), this will not be discovered since the teacher accepts the positive cases and does not inspect them. However, when there are a large number of submissions to be assessed, even teachers make errors and cannot assess all the submissions with 100% accuracy. Moreover, a teacher can take random samples from the positive cases and evaluate the accuracy of the system with regard to the false positive cases within a certain period of time. This will give an indication of the benefit of the system (as saving a teacher's time) compared to the accepted incorrect cases. This application of the method would allow the teacher to give better personal comments to the students and pick up interesting examples to discuss with them.

On the other hand, students could use the informative feedback that Aari system can provide on their implementations to gain a better understanding of their code. We have shown that students make problematic implementation choices in the case of sorting algorithms and we have developed and discussed techniques that can identify these kind of implementation choices. We can thus give automatic feedback on problematic solutions and reasonably assume that these types of feedback could be justified and beneficial. However, in order to use Aari to give feedback on a comprehensive set of algorithms, we need to do similar studies to examine student-implemented variations in other fields of algorithms. Moreover, we need to evaluate our method in an educational setting and investigate how students use the system and how useful they find it. A big concern remains the accuracy of the system. For the algorithms that Aari has been train to recognize, it performs well (as reported in Publication IV). However, for an open task of implementing sorting algorithms, Aari achieved the average accuracy of 75%. Therefore, since making a summative assessment on whether something is right or wrong might have a negative effect, we should give feedback in form of suggesting something that the student should look at. In short, before using Aari as a feedback providing system, we need to situate it in the educational setting and assess how it could improve education.

The method has potential to be extended to be applied in software engineering related tasks as well. As an example, the task in clone detection is to identify similar pieces of code. This is what our method, with the appropriate further developments, would be capable of performing for the implementations that are stored in its knowledge base. However, since these activities involve dealing with large-scale software (unlike implementations in computer science education), the performance of the method in this context should be evaluated with empirical tests.

8.1.2 Our Methods and Other Research Fields

The methods introduced in this thesis for algorithm recognition (AR), which is considered as a subfield of practical program comprehension (PC), draw on some techniques and concepts used in other research fields discussed in Chapter 2. For example, the methods use software metrics that are also used in program similarity evaluation techniques, or they use the concept of programming schemas introduced by studies on theoretical PC. We could benefit from other experiences and results of the related studies as well. As an example, concept and feature location approaches of PC field have achieved good results by combining dynamic and static techniques. These hybrid techniques use program dependencies and textual information gained by static analysis of source code to filter the execution traces, which are often very large and contain a lot of noise [59]. In order to increase the accuracy (i.e., decrease false positive and false negative cases) this possibility should be explored in AR as well. Moreover, we can examine the applicability of control flow and data flow metrics used in metrics-based clone detection approaches (e.g., [62]) to see whether they provide useful information for AR.

Our SDM applies the results from theoretical PC in practical PC. Our CLM, on the other hand, utilized machine learning techniques to recognize unseen algorithm implementations. A number of PC tools use plans in their knowledge base to facilitate the process of comprehension. These tools, however, do not use supervised machine learning widely. Our method brings supervised learning into the process of PC. The method teaches a supervised learner (i.e., a classifier) with a set of learning data instances, how to identify each algorithm based on its features. The learner is then able to apply this knowledge to recognize new and previously unseen instances. This concept of using learning in PC tools is also discussed by Gerdt and Sajaniemi in the context of developing a role detection tool based on machine learning techniques [32]. Moreover, by applying the concept of beacons from theoretical PC, we have introduced features that can be utilized to characterize and detect source code fragments in, for example, clone detection tools. An important part of these beacons are roles of variables, for which we have introduced a new application area in this thesis.

8.2 Research Questions Revisited and Future Work

We posed eight questions in Section 1.2, from which the first one was:

1. *How could we automatically recognize basic algorithms and their variations from source code?*

We have answered this question throughout this thesis by presenting the CLM and SDM, as well as the characteristics and beacons. With respect to this question, we conclude that our method works with high accuracy for the algorithms that it has been trained to recognize. However, the accuracy of the method decreases with previously unknown algorithms that the method has no mechanism to deal with. For example, implementations of Shell sort might be falsely recognized as, say Bubble sort. To tackle this, we need to add the appropriate mechanism to the method so that it can deal with other algorithms that are covered in data structures and algorithms courses.

The second and third questions concerned the algorithmic characteristics, beacons and schemas as follows:

2. *Can algorithmic characteristics and beacons be utilized in AR process and how?*

3. *Can programming schemas facilitate automatic AR? How can we implement a method based on schemas?*

To answer these questions, we introduced a set of characteristics and beacons, and developed schemas for a number of algorithms, as discussed in Chapters 5 and 6. In Chapter 7, we presented the evaluation results of the SDM showing that the method performs with high accuracy. With respect to the schemas, an important question is the level of abstraction they are defined at. If the level of abstraction is too high, there exists a threat that in programs with a bigger size, false implementations would be identified as acceptable (i.e., false positive cases increase). On the other hand, a too low level of abstraction may result in identifying correct implementations as not acceptable (false negative cases).

The research question with regard to RoV was:

4. *How applicable and useful RoV are in recognizing basic algorithms?*

The usefulness of RoV in introductory programming education has been investigated in many studies and the results show that using RoV can increase students' skills in comprehending and constructing programs (see, e.g., [14, 86], as well as [93] for information from a teacher's point of view). According to our empirical studies, RoV are very useful beacons in the AR task as well. As an example, the *most-wanted holder* role distinguishes the implementations of Selection sort from the implementations of other sorting algorithms that we analyzed.

In our empirical studies, we used the tool developed by Bishop and Johnson [9] for detecting RoV. The tool detected the roles in the empirical studies on sorting algorithms very accurately. However, when analyzing searching, heap, basic tree traversal and graph algorithms, the tool did not detect the roles of the variables in the implementations of these algorithms accurately enough. Perhaps with a more accurate tool, RoV could have played a distinguishing role as beacons for these algorithms as well. As an example, the low index in a binary search (e.g., $low = middle + 1$) has a *follower* role ([83]) that could be a useful beacon for differentiating between the implementations of binary search algorithm from other implementations. Developing a role detection tool that performs with high precision is a challenging task, but as Gerdt and Sajaniemi describe in [31], good results are achieved by using data flow analysis and applying machine learning techniques to determine data flow characteristics for roles. Unfortunately, we did not get access to this tool. One interesting idea would be to go to a lower-level of abstraction and use directly the data flow characteristics that define the roles, instead of the roles themselves, and evaluate which one would characterize the functionality of an algorithm better.

With respect to the applicability of machine learning techniques, we posed the following research question:

5. *Can machine learning methods, and in particular the C4.5 algorithm,*

be used in AR problem and how accurate it is?

Converting implementations of algorithms to characteristic and beacon vectors and using these vectors as technical definitions of algorithms allows us to utilize machine learning techniques. Using the C4.5 decision tree classifier in our CLM and the estimated performance of the classification illustrate the applicability of the C4.5 algorithm as a supervised machine learning classification technique. Quality of learning and testing data sets impacts the performance of machine learning methods. In order to have representative and unbiased data sets, we have collected them from different sources, including textbooks, the Web resources and students' implementations.

The sixth question concerned the CSC:

6. *How can we combine the SDM and CLM to get more reliable results?*

The CSC was discussed in Chapters 5 and its performance evaluation results were presented in Chapter 7. By selecting the code fragment that implements the algorithm in question for further analysis, the CSC achieves more reliable performance, as the irrelevant code does not affect the value of the computed characteristics and beacons.

The seventh and eight research questions were the following:

7. *How can we classify students' implementations of sorting algorithms? What kind of variations of well-known sorting algorithms students use?*

8. *How accurately Aari can recognize student-implemented sorting algorithms and their variations?*

The results of categorizing sorting algorithms are presented in Subsection 7.3.1. The purpose of the categorization is to discover what types of problematic solutions students use and to develop a method to automatically identify these solutions. This allows to give useful feedback that make students rethink their solutions (which remains for future work). We found that students have many misconceptions related to sorting algorithms. They include unnecessary swaps in their Insertion and Selection sort implementations which makes the code more complicated and inefficient. In this context, the term misconception is used to indicate problematic understandings or failure to fully understand the basic principles behind some well-known algorithms. Similar studies should be done for other fields of algorithms as well. Moreover, to get a better insight into students' misconceptions, we need to ask them about how they describe their own code and what they think about their solutions. We discussed the schemas and beacons developed to recognize these variations in Subsections 6.1.1 and 6.2.1.

Finally, Subsection 7.3.2 discusses the performance of Aari system on students' implementations. In this respect, we conclude that Aari performs accurately with the algorithms that it has been trained to recognize. However, as expected, if implementations of algorithms that Aari has no mechanism to deal with are involved, the accuracy decreases. As also discussed above, covering other fields of algorithms will address this problem. In addition, Aari needs to be further evaluated with authentic students' implementations from other algorithm fields as well.

In this thesis, we have examined the application of the presented techniques on programs written in Java. As, for example, length of a program in terms of lines of code varies from a programming language to another, a direction for future work could be to examine the techniques on materials written in other languages and compare the results.

8.3 Validity

In this section we discuss possible issues with internal and external validity involved in our research. We reflect on the possible related threats and discuss how we have addressed them.

8.3.1 Internal Validity

As roles of variables play an important role in our method, it is important to use a tool that detects roles accurately. Poor performance of such tool is a threat to the internal validity of our research. To eliminate this threat, we used the most accurate role detector available to us. We did several improvements to the tool to improve its accuracy (see Section 5.3). As the result, for those roles that are used in recognizing sorting algorithms, the tool worked very accurately. As discussed above, for other algorithms, the tool did not perform accurately enough. A more accurate role detector is needed in order to utilize the roles in recognizing other fields of algorithms.

Another potential threat is the manual categorization of the students' implementations of sorting algorithms reported in Publication III. According to Knuth, “the classification of sorting methods into various families such as “insertion,” “exchange,” “selection,” etc., is not always clear-cut.” [48]. When it is about novices' implementations, the categorization is even more difficult. In addition to the implementations of standard sorting algorithms, there were many other types of implementations in the data such as inefficient variations, which we categorized as selection sort with inner loop swap and insertion sort with swap (discussed in Subsection 7.1.1). Therefore, it is possible that some implementations have been categorized wrongly. Parts of the results of our research as well as parts of the techniques developed in Aari system are based on this categorization, and thus internal validity may be threatened by the incorrectly categorized algorithm implementations. To rule out or reduce this threat, the categorization and especially the unclear and borderline cases were iteratively reconsidered and discussed by three researchers as follows. In the beginning, one of the researchers analyzed the implementations and tentatively classified them into the appropriate categories. After this, he and another researcher reviewed and discussed the categories and reconsidered especially the implementations with no obvious and clear category. As a result of this discussion, the categorization was specified and the borderline cases were reclassified if necessary. All the researchers discussed the resulted categorization and unclear implementations one more time and agreed on the final categorization. The researchers reached a consensus on the problematic cases. The implementations gathered from textbooks and Websites were naturally much more clear and reliable and thus do not pose a serious threat.

It should also be emphasized that we selected the metrics used in the techniques presented in this thesis based on literature review. Before selecting them, we could not find any empirical evidence on their application to algorithm recognition problem, although several of them are widely applied and evaluated in program similarity evaluation techniques. It is possible that there exist other metrics that can be applied to the problem with good results. The applicability of different metrics in terms of accuracy could be an interesting direction of future work.

In addition, there always exists a potential risk that a program may have bugs that produce incorrect results and cause the program to perform in an unexpected way. Aari system is no exception. By doing different tests and comparing the results, we have tried to reduce this risk and improve the quality of the system.

8.3.2 External Validity

External validity is related to the extent to which the results of a study are generalizable. In our study, this refers to whether the proposed methods can be applied to different fields of algorithms with the same accuracy. In the beginning of our research, we demonstrated our methods and their performance in the case of basic sorting algorithms. We then extended our methods to the variations of these sorting algorithms as well as to searching, heap, basic tree traversal and graph algorithms with practically the same results. This indicates that the methods and results are generalizable.

In addition, we collected the implementations of our data sets randomly, that is, with no preferences (to, e.g., a particular source or alike). This makes the implementations representative and implies that, for each analyzed algorithm field, the methods are highly probable to be generalized to other implementations of that algorithm field with the same results.

Bibliography

[1] Kirsti Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

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

[3] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 98–107. IEEE Computer Society Washington, DC, USA, 2000.

[4] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceedings of the 10th European Software Engineering Conference, Lisbon, Portugal, 5–9 September*, pages 156–165. ACM, New York, NY, USA, 2005.

[5] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth McMaster, Andrew Turpin, and Stan Jarzabek. 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*, pages 513–516. ACM New York, NY, USA, 2007.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th IEEE International Conference on Software Maintenance, Bethesda, Maryland, USA, 16–19 March, 1998*, pages 368–377. IEEE Computer Society Washington, DC, USA, 1998.

[7] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[8] Mordechai Ben-Ari and Jorma Sajaniemi. Roles of variables as seen by CS educators. *SIGCSE Bulletin*, 36(3):52–56, 2004. ISSN 0097-8418.

[9] 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.

[10] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[11] Ilene Burnstein and Katherine Roberson. Automated chunking to support program comprehension. In *Proceedings of the 5th International Workshop on Program Comprehension (IWPC '97)*, pages 40–49. IEEE Computer Society Washington, DC, USA, 1997.

[12] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and debugging tool. In *9th International Conference on Tools with Artificial Intelligence (ICTAI '97)*, pages 102–109. IEEE Computer Society Washington, DC, USA, 1997.

[13] Irene Burnstein and Floyd Saner. An application of fuzzy reasoning to support automated program comprehension. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC '99), Pittsburgh, Pennsylvania, USA, 5–7 May*, pages 66–73. IEEE Computer Society Washington, DC, USA, 1999.

[14] Pauli Byckling and Jorma Sajaniemi. Roles of variables and programming skills improvement. *SIGCSE Bulletin*, 38(1):413–417, 2006. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/1124706.1121470.

[15] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, USA, third edition, 2009.

[17] Cynthia L. Corritore and Susan Wiedenbeck. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, 54(1):1–23, 2001.

[18] Martha E. Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *Proceedings of the 14th Annual Workshop on the Psychology of Programming Interest Group (PPIG '02), Brunel University, London, UK.*, pages 58–73, 2002.

[19] Nell Dale, Daniel T. Joyce, and Chip Weems. *Object-Oriented Data Structures Using Java*. Jones and Bartlett Publishers, Sudbury, MA, USA, first edition, 2002.

[20] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, To appear.

[21] Françoise 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.

[22] Françoise Détienne. What model(s) for program understanding? In *Conference on Using Complex Information Systems (UCIS '96)*, Poitiers, France, 1996.

[23] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99), 1999*, pages 109–118. IEEE Computer Society Washington, DC, USA, 1999.

[24] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, 2006.

[25] Stephen 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. ISBN 1-58113-751-6. doi: http://doi.acm.org/10.1145/949344.949390.

[26] Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05), 2005*, pages 337–346. EEE Computer Society Washington, DC, USA, 2005.

[27] William H. Ford and William R. Topp. *Data Structures with Java*. Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[28] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. 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*, pages 351–360. EEE Computer Society Washington, DC, USA, 2009.

[29] Petri Gerdt. A system for the automatic detection of variable roles. *Licenciate Thesis, Department of Computer Science, University of Joensuu, Finland*, 2007.

[30] Petri Gerdt and Jorma Sajaniemi. 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*, pages 86–93. The University of Warwick, UK, 2004.

[31] Petri Gerdt and Jorma 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. ISBN 1-59593-055-8. doi: http://doi.acm.org/10.1145/1140124.1140172.

[32] Petri Gerdt and Jorma Sajaniemi. Introducing learning into automatic program comprehension. In *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group (PPIG '07), Joensuu, Finland, July 2007*, pages 101–115. International Proceedings Series 7, University of Joensuu, Department of Computer Science and Statistics, 2007.

[33] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[34] Sam Grier. A tool that detects plagiarism in pascal programs. In *Proceedings of the twelfth SIGCSE technical symposium on Computer science education (SIGCSE '81)*, pages 15–20. ACM New York, NY, USA, 1981.

[35] Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc, New York, NY, USA, 1977.

[36] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, 1990.

[37] Colin Higgins, Pavlos Symeonidis, and Athanasios 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. ISBN 1-58113-499-1. doi: http://doi.acm.org/10.1145/544414.544431.

[38] Petri Ihantola, Ville Karavirta, Otto Seppälä, and Tuukka 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)*, pages 86–93. ACM New York, NY, USA, 2010.

[39] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys (CSUR)*, 31(3):264–323, 1999.

[40] Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho. A source code linearization technique for detecting plagiarized programs. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '07)*, pages 73–77. ACM New York, NY, USA, 2007.

[41] W. Lewis Johnson and Elliot 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.

[42] Edward L. Jones. Metrics based plagiarism monitoring. In *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges (CCSC '01)*, pages 253–261. Consortium for Computing Sciences in Colleges , USA, 2001.

[43] Mike Joy, Nathan Griffiths, and Russell Boyatt. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3):1–28, 2005.

[44] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceeding Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 485–495. IEEE Computer Society Washington, DC, USA, 2009.

[45] Cory Kapser and Michael W. Godfrey. A taxonomy of clones in source code: The re-engineers most wanted list. In *Proceedings of IWDSC'03*, 2003.

[46] Cory Kapser and Michael W. Godfrey. Toward a taxonomy of clones in source code: A case stud. In *Evolution of large scale industrial software architectures*, pages 67–78, 2003.

[47] Young-Chul Kim and Jaeyoung Choi. A program plagiarism evaluation system. In *Proceedings of the 2005 international conference on Computational Science and Its Applications (ICCSA '05)*, pages 10–19. Springer-Verlag Berlin, Heidelberg, 2005.

[48] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, New Jersey, USA, second edition, 1998.

[49] Ron Kohavi and John Ross Quinlan. Decision tree discovery. In *In Handbook of Data Mining and Knowledge Discovery*, pages 267–276. University Press, 1999.

[50] Raghavan Komondoor and Susan Horwitz. 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*, pages 40–56. Springer, 2001.

[51] Kostas A Kontogiannis, R DeMori, Ettore M Merlo, M Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3(1–2):77–108, 1996.

[52] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica, An International Journal of Computing and Informatics*, 31(3):249–268, 2007.

[53] Wojtek Kozaczynski, Jim Ning, and Tom Sarver. Program concept recognition. In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference (KBSE '92). Los Alamitos, CA, 20-23 September, 1992*, pages 216–225. IEEE Computer Society Press, 1992.

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

[55] Marja Kuittinen and Jorma Sajaniemi. 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)*, pages 57–61, 2004.

[56] Ronald J. Leach. Using metrics to evaluate student programs. *ACM SIGCSE Bulletin*, 27(2):41–43, 1995.

[57] Anany Levitin. *Introduction to The design and analysis of algorithms*. Pearson Education Inc., Boston, MA, USA, third edition, 2012.

[58] Tjen-Sien Lim, Wei-Yin Loh, and Yu-Shan Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, 40(3):203–228, 2000.

[59] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. 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*, pages 234–243. ACM New York, NY, USA, 2007.

[60] Andrian Marcus and Jonathan I. Maletic. 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*, pages 107–114. IEEE, Washington, DC, USA, 2001.

[61] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. 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*, pages 214–223. EEE Computer Society Washington, DC, USA, 2004.

[62] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 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*, pages 244–254. IEEE Computer Society Washington, DC, USA, 1996.

[63] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2:308–320, 1976.

[64] Katherine B. McKeithen, Judith S. Reitman, Henry H. Rueter, and Stephen C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13(3):307–325, 1981.

[65] Robert Metzger and Zhaofang Wen. *Automatic Algorithm Recognition and Replacement*. The MIT Press, USA, 2000.

[66] Dennis M Miller, Robert S Maness, James William Howatt, and Wade H Shaw. A software science counting strategy for the full ada language. *ACM SIGPLAN Notices*, 22(5):32–41, 1987.

[67] Maxim Mozgovoy. *Enhancing Computer-Aided Plagiarism Detection*. Doctoral dissertation, University of Joensuu, 2007.

[68] Sreerama K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2(4): 345–389, 1998.

[69] Michael O'Brien. Software comprehension – a review and research direction. *Technical Report no. UL-CSIS-03-3. University of Limerick*, pages 176–185, 2003.

[70] Nancy Pennington. Comprehension strategies in programming. *Empirical studies of programmers: second workshop*, pages 100–113, 1987.

[71] Denys Poshyvanyk and Andrian Marcus. 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*, pages 37–48. EEE Computer Society Washington, DC, USA, 2007.

[72] Denys Poshyvanyk and Andrian Marcus. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[73] Denys Poshyvanyk, Malcom Gethers, and Andrian Marcus. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4), 2012.

[74] Alex Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.

[75] Alex Quilici. Reverse engineering of legacy systems: a path toward success. In *Proceedings of the 17th international conference on Software engineering, Seattle, Washington, USA, 24–28 April*, pages 333–336. ACM, New York, NY, USA, 1995.

[76] John Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1): 81–106, 1986.

[77] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, USA, 1993. ISBN 0-321-32136-7.

[78] Michael J. Rees. Automatic assessment aids for pascal programs. *ACM SIGPLAN Notices*, 17(10):33–42, 1982.

[79] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 13: 389–414, 1989.

[80] Chanchal K. Roy and James R. Cordy. A survey on software clone detection research. *Queen's Technical Report:541, 115 pp.*, 2007.

[81] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[82] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution using clone detection. In *Proceedings of the 4th ECOOP '03 International Workshop on Object-Oriented Reengineering (WOOR '03), Darmstadt, Germany, July 2003.*, pages 71–75. USENIX Association Berkeley, CA, USA, 2003.

[83] Jorma 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.

[84] Jorma Sajaniemei and Raquel Navarro Prieto. 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.*, 2005.

[85] Jorma 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.

[86] Jorma Sajaniemi and Marja Kuittinen. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1):59–82, 2005.

[87] Jorma Sajaniemi, Mordechai Ben-Ari, Pauli Byckling, Petri Gerdt, and Yevgeniya Kulikova. Roles of variables in three programming paradigms. *Computer Science Education*, 16(4):261–279, 2006.

[88] Norman F Salt. Defining software science counting strategies. *ACM SIGPLAN Notices*, 17(3):58–67, 1982.

[89] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James 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. New York, NY, USA, 2010.

[90] Robert Sedgewick. *Algorithms in Java, Parts 1-4*. Pearson Education Inc., Boston, MA, USA, third edition, 2002.

[91] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

[92] Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and processes in the comprehension of computer programs. In M. Farr M. Chi, R. Glaser, editor, *The Nature of Expertise*, pages 129–152. Lawrence Erlbaum Associates, 1988.

[93] Juha Sorva, Ville Karavirta, and Ari Korhonen. Roles of variables in teaching. *Journal of Information Technology Education*, 6:407–423, 2007. URL `http://jite.org/documents/Vol6/JITEv6p407-423Sorva280.pdf`.

[94] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3): 187–208, 2006.

[95] Hanna Suominen, Tapio Pahikkala, and Tapio Salakoski. Critical points in assessing learning performance via cross-validation. In Timo Honkela, Matti Pöllä, Mari-Sanna Paukkeri, and Olli Simula, editors, *Proceedings of the 2nd International and Interdiciplinary Conference on Adaptive Knowledge Representation and Reasoning (AKRR 2008)*, pages 9–22, Espoo, Finland, 2008. Helsinki University of Technology. ISBN 978-951-22-9525-8.

[96] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, USA, 2006. ISBN 0-321-32136-7.

[97] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Control archive*, 19(2):295–331, 2011.

[98] Anneliese von Mayrhauser and A. Marie Vans. Program understanding – a survey. *Technical Report CS-94-120, Department of Computer Science, Colorado State University*, 1994.

[99] Anneliese von Mayrhauser and A. Marie Vans. Program understanding: Models and experiments. *In: M. Yovits and M. Zelkovitz (eds.), Advances in Computers*, 40:1–37, 1995.

[100] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, 1988.

[101] Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697–709, 1986.

[102] Linda M. Wills. Using attributed flow graph parsing to recognize clichés in programs. In *Proceedings of the 5th International Workshop on Graph Grammars and Their Application to Computer Science*, pages 170–184. Springer-Verlag London, UK, 1996.

[103] Michael J. Wise. Yap3: improved detection of similarities in computer program and other texts. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education (SIGCSE '96)*, pages 130–134. ACM New York, NY, USA, 1996.

[104] Wuu Yang. Identifying syntactic differences between two programs. *Software–Practice and Experience*, 21(7):739–755, 1991.

A. The C4.5 Decision Tree Classifier

The C4.5 algorithm is a widely used and the most well-known algorithm for building decision trees [58]. In the following, we explain how the C4.5 algorithm deals with the important issues related to building decision trees, such as finding the best attributes to construct a decision tee and finding the right size for a tree. The discussion in this section is based on the book about the C4.5 algorithm written by its inventor [77].

A.1 Finding the Best Attribute

The earlier version of the C4.5 algorithm used *information gain* to evaluate the tests and find the best split. As described below, a more accurate criterion called *information gain ratio* was adopted later.

Information gain (also called *mutual information*) is based on entropy, a measure used in information theory. Entropy indicates the average information needed to identify instances of a set. Let S be a set of instances, c be the number of different classes in S and n_i be the number of instances in S that belong to class i. Entropy can be defined as follows:

(1) $entropy(S) = -\sum_{i=1}^{c} n_i \times \log_2 n_i$

The information gain is the difference between the entropy of the set S before the split and the entropy of the set S after the split that follows some test T. Therefore, the information gain can be computed by the following formula:

(2) $gain(T) = entropy(S) - \sum_{j=1}^{k} \frac{|S_j|}{|S|} \times entropy(S_j)$

Here, k is the number of outcomes of the test T (i.e., the set of values of the attribute T), and S_j indicates the number of instances in S, where T has value j. $gain(T)$ measures the information gained by splitting the set S according to the test T. To perform the split, the C4.5 algorithm, like its predecessor ID3, selects the test that gives the maximum information gain. Thus, the decision tree is generated so that those internal nodes that give the largest information gain are expanded.

Although the information gain was used as the criterion in the ID3 for many years with good results, Quinlan, the inventor of the ID3 and C4.5 algorithms developed a criterion called information gain ratio to fix the deficiency of information gain: information gain favors the tests that result in many outcomes. This causes problems when the outcomes of this kind of tests have no value with regard to the classification, for example, because of the small number of instances associated with each outcome. His solution to correct the issue is to adjust the gain of these kinds of tests. The information gain ratio is the ratio of the information gain to the split information. It gives the information that is obtained by the ratio of the information relevant to the classification produced by the split, to the information that is provided by the split itself. Thus, the information gain ratio can be formally defined as

(3) $gain\ ratio(T) = gain(T)/split\ entropy(T)$

$split\ entropy(T)$ is computed by the following formula:

(4) $split\ entropy(T) = -\sum_{j=1}^{k} \frac{|S_j|}{|S|} \times \log_2 \frac{|S_j|}{|S|}$

The denominator in Formula 3 grows rapidly if a test results in many outcomes. However, if the test is trivial (for example, each outcome of the split contains only one instance), the numerator would be small. Thus the overall information gain ratio would remain small. This will eliminate the chances of these kinds of tests to become selected.

In the case of unknown attribute values, information gain is computed as follows. Let p_1 denote the probability that the value of the attribute A tested in test T is known. Correspondingly, let p_2 denote the probability that the value of the same attribute in the same test is unknown. The information gain is

(5) $gain(T) = p_1 \times (entropy(S) - \sum_{j=1}^{k} \frac{|S_j|}{|S|} \times entropy(S_j)) + p_2 \times 0$

The value of zero in the end of Formula 5 reflects the fact that if the value of the attribute is missing, clearly no information can be gained for the corresponding instance from the split in question. If we suppose that the value of A is known in fraction F of the instances in the set S, we get the following simpler formula for computing information gain for unknown attribute values:

(6) $gain(T) = F \times (entropy(S) - \sum_{j=1}^{k} \frac{|S_j|}{|S|} \times entropy(S_j))$

Formula 6 is the same as Formula 2 multiplied by the fraction of the instances that have the value of the corresponding attribute available. The effect of missing attribute values in computing the information gain ratio can be taken into consideration in the similar way, using Formula 4.

A.2 Finding the Right Size

The issue of finding the right size in the C4.5 algorithm is handled by pruning the tree after it has been constructed. The tree is built using the divide and conquer principle without evaluating any split at the building phase. This results in an overfitted tree, which is then pruned to become simpler: those parts of the tree that are not important in terms of the accuracy are removed. This approach includes an extra computation for building the parts of the tree that will be eliminated later in the pruning phase. However, this is well justified by the more accurate and reliable final result [77].

In the C4.5 algorithm, pruning includes either replacing subtrees with leaves or with one of its branches. Pruning is error-based, that is, the replacement is carried out if it results in a lower predicted error rate. The process starts from the bottom of the tree and proceeds by investigating each non-leaf subtree. To predict the error rate, the C4.5 algorithm uses a sophisticated pruning heuristic which is based on computing the probability of appearance of misclassified instances in a leaf relative to all instances covered by that leaf.

B. Pseudo-Code for Sorting, Searching, Heap, Basic Tree Traversal and Graph Algorithms

The pseudo-code examples for the algorithms discussed in this thesis are listed below.

B.1 Sorting Algorithms

Bubble Sort

Bubble sort (algorithm 1) is adapted from [16].

Algorithm 1 BUBBLE-SORT(A)

```
for $i = 1$ to $A.length - 1$ do
    for $j = A.length$ downto $i + 1$ do
        if $A[j] < A[j - 1]$ then
            swap $A[j]$ $and$ $A[j - 1]$
        end if
    end for
end for
```

Insertion Sort

Insertion sort (algorithm 2) is adapted from [57].

Algorithm 2 INSERTION-SORT(A)

```
for $i = 1$ to $A.length - 1$ do
    $key = A[i]$
    $j = i - 1$
    while $j \geq 0$ and $A[j] > key$ do
        $A[j + 1] = A[j]$
        $j = j - 1$
    end while
    $A[j + 1] = key$
end for
```

Selection Sort

Selection sort (algorithm 3) is adapted from [57].

Algorithm 3 SELECTION-SORT(A)

```
for $i = 0$ to $A.length - 2$ do
  $min = i$
  for $j = i + 1$ to $A.length - 1$ do
    if $A[j] < A[min]$ then
      $min = j$
    end if
  end for
  swap $A[j]$ $and$ $A[min]$
end for
```

Mergesort

Mergesort (algorithms 4 and 5) is adapted from [16].

Algorithm 4 MERGESORT(A, p, r)

```
if $p < r$ then
  $q = \lfloor (p + r)/2 \rfloor$
  MERGESORT$(A, p, q)$
  MERGESORT$(A, q + 1, r)$
  MERGE$(A, p, q, r)$
end if
```

Algorithm 5 MERGE(A, p, q, r)

```
$n_1 = q - p + 1$; $n_2 = r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$ do
  $L[i] = A[p + i - 1]$
end for
for $j = 1$ to $n_2$ do
  $R[j] = A[q + j]$
end for
$L[n_1 + 1] = \infty$; $R[n_2 + 1] = \infty$
$i = 1$; $j = 1$
for $k = p$ to $r$ do
  if $L[i] \leq R[j]$ then
    $A[k] = L[i]$
    $i = i + 1$
  else
    $A[k] = R[i]$
    $j = j + 1$
  end if
end for
```

Quicksort

Quicksort (algorithms 6 and 7) is adapted from [16].

Algorithm 6 QUICKORT(A, p, r)

```
if $p < r$ then
  $q = PARTITION(A, p, r)$
  QUICKSORT$(A, p, q - 1)$
  QUICKSORT$(A, q + 1, r)$
end if
```

Algorithm 7 PARTITION(A, p, r)

```
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$ do
  if $A[i] \leq x$ then
    $i = i + 1$
    swap $A[i]$ $and$ $A[j]$
  end if
end for
swap $A[i + 1]$ $and$ $A[r]$
return $i + 1$
```

B.2 Binary Search Algorithms

Non-recursive Binary Search

Non-recursive binary search (algorithm 8) is adapted from [90].

Algorithm 8 NON-RECURSIVE-BINARY-SEARCH(A, l, r, v)

```
while $r \geq l$ do
  $m = (l + r)/2$
  if $v == A[m]$ then
    return $m$
  end if
  if $v < A[m]$ then
    $r = m - 1$
  else
    $l = m + 1$
  end if
end while
return $-1$
```

Recursive Binary Search

Recursive binary search (algorithm 9) is adapted from [90].

Algorithm 9 RECURSIVE-BINARY-SEARCH(A, l, r, v)

```
if $l > r$ then
  return $null$
end if
$m = (l + r)/2$
if $v == A[m]$ then
  return $A[m]$
end if
if $v < A[m]$ then
  return RECURSIVE-BINARY-SEARCH$(A, l, m - 1, v)$
else
  return RECURSIVE-BINARY-SEARCH$(A, m + 1, r, v)$
end if
```

B.3 Depth First Search Algorithm

Depth first search (algorithms 10 and 11) is adapted from [57].

Algorithm 10 DEPTH-FIRST-SEARCH(G)

```
mark each vertex in $V$ with $0$ as a mark of being “unvisited”
$count = 0$
for each vertex $v$ in $V$ do
  if $v$ is marked with $0$ then
    DFS-VISIT$(v)$
  end if
end for
```

Algorithm 11 DFS-VISIT(G, u)

```
$count = count + 1$; mark $v$ with $count$
for each vertex $w$ in $V$ adjacent to $v$ do
  if $w$ is marked with $0$ then
    DFS-VISIT$(w)$
  end if
end for
```

B.4 Tree Traversal Algorithms

Preorder Traversal

Preorder traversal (algorithm 12) is adapted from [90].

Algorithm 12 PREORDER-TRAVERSAL(x)

```
if $x == $ NIL then
    return
end if
print($x.key$)
PREORDER-TRAVERSAL($x.left$)
PREORDER-TRAVERSAL($x.right$)
```

Inorder Traversal

Inorder traversal (algorithm 13) is adapted from [16].

Algorithm 13 INORDER-TRAVERSAL(x)

```
if $x \neq$ NIL then
    INORDER-TRAVERSAL($x.left$)
    print($x.key$)
    INORDER-TRAVERSAL($x.right$)
end if
```

Postorder Traversal

Postorder traversal (algorithm 14) is adapted from [19].

Algorithm 14 POSTORDER-TRAVERSAL(x)

```
if $x \neq$ NIL then
    POSTORDER-TRAVERSAL($x.left$)
    POSTORDER-TRAVERSAL($x.right$)
    print($x.key$)
end if
```

B.5 Heap Algorithms

Heap Insertion

Heap insertion (algorithm 15) is adapted from [27].

Algorithm 15 HEAP-INSERTION$(H, last, item)$

```
$currPos = last$
$parentPos = (currPos - 1)/2$
**while** $currPos \neq 0$ **do**
  **if** $item > H[parentPos]$ **then**
    $H[currPos] = H[parentPos]$
    $currPos = parentPos$
    $parentPos = (currPos - 1)/2$
  **end if**
**end while**
$H[currPos] = item$
```

Heap Remove

Heap remove (algorithms 16 and 17) is adapted from [27].

Algorithm 16 HEAP-REMOVE$(H, last)$

```
$temp = H[0]$
$H[0] = H[last - 1]$
$H[last - 1] = temp$
ADJUST-HEAP$(H, 0, last - 1)$
**return** $temp$
```

Algorithm 17 ADJUST-HEAP$(H, first, last)$

```
$currPos = first$
$target = H[first]$
$childPos = 2 * currPos + 1$
**while** $childPos < last$ **do**
  **if** $childPos + 1 < last$ and $H[childPos + 1] > H[childPos]$ **then**
    $childPos = childPos + 1$
  **end if**
  **if** $H[childPos] > target$ **then**
    $H[currPos] = H[childPos]$
    $currPos = childPos$
    $childPos = 2 * currPos + 1$
  **end if**
**end while**
$H[currPos] = target$
```

B.6 Graph Algorithms

Dijkstra's Algorithm

Dijkstra's algorithm for the single-source shortest-paths problem (algorithm 18) is adapted from [57].

Algorithm 18 DIJKSTRA(G, s)

```
$Initialize(Q)$ //initialize priority queue to empty
for every vertex $v$ in $V$ do
  $d_v = \infty$
  $p_v =$ null
  $Insert(Q, v, d_v)$ //initialize vertex priority in the priority queue
end for
$d_s = 0$
$Decrease(Q, s, d_s)$ //update priority of $s$ with $d_s$
$V_T = \emptyset$
for $i = 0$ to $|V| - 1$ do
  $u^* = DeleteMin(Q)$ //delete the minimum priority element
  $V_T = V_T \cup \{u^*\}$
  for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
    if $d_{u^*} + w(u^*, u) < d_u$ then
      $d_u = d_{u^*} + w(u^*, u)$
      $p_u = u^*$
      $Decrease(Q, u, d_u)$
    end if
  end for
end for
```

Floyd's Algorithm

Floyd's algorithm for all-pairs shortest-paths problem (algorithm 19) is adapted from [57].

Algorithm 19 FLOYD($W[1..n, 1..n]$)

```
$D = W$
for $k = 1$ to $n$ do
  for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
      $D[i, j] = min\{D[i, j], D[i, k], D[k, j]\}$
    end for
  end for
end for
return $D$
```

Errata

Publication I

In Subsection 3.2 on page 1055, ‘(the implemented tool that performs the’ should be ‘(the implemented tool that performs the recognition), the first phase to be performed’.
In Subsection 5.4 on page 1061, ‘is less that’ should be ‘is less than’.
In Subsection 5.4 on page 1062, ‘column in blue’ should be ‘column in light gray’, ‘column in green’ should be ‘column in medium gray’ and ‘column in red’ should be ‘column in dark gray’.
In Subsection 5.4 on page 1063, ‘reveals than’ should be ‘reveals that’.

Publication II

In Subsection 1.1 on page 1847, ‘are much more’ should be ‘is much more’.
In Subsection 2.1 on page 1848, ‘sourced code’ should be ‘source code’.
In Subsection 6.3 on page 1858, ‘experience’ should be ‘experiment’.
In Subsection 7, ‘Simple additions’ should be ‘Simple addition’.

Publication III

In the caption of Figure 5, ‘in’ is redundant.

Publication V

In Sections 4, the second-to-last paragraph, ‘This make’ should be ‘This makes’.

In Sections 6, ‘111 program’ should be ‘111 programs’.

In Sections 6 and 7, ‘10 percent’ should be ‘9 percent’.

Aalto-DD 17/2013

ISBN 978-952-60-4989-2
ISBN 978-952-60-4990-8 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi