AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Ahmad Taherkhani

Recognizing Algorithms Using Roles of Variables, Language Constructs and Software Metrics: A Machine Learning Approach

Licentiate's Thesis submitted in partial fulfillment of the requirements for the degree of Licentiate of Science in Technology.

Espoo, November, 19, 2010

Supervisor: Professor Lauri Malmi
Instructor: Docent Ari Korhonen

AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

ABSTRACT OF
LICENTIATE'S THESIS

Author: Ahmad Taherkhani

Date: November, 19, 2010
Pages: 65 + 44

Title of the thesis:
Recognizing Algorithms Using Roles of Variables, Language Constructs and Software Metrics: A Machine Learning Approach

Language:
English

Professorship: Software Systems

Code: T-106

Supervisor: Professor Lauri Malmi

Instructor: Docent Ari Korhonen

Roles of Variables (RoV) are concepts that describe the behavior and usage of variables in computer programs. RoV have originally been introduced to help novices learn programming.

Algorithm Recognition (AR) is a subfield of program comprehension, where the problem is to identify algorithms from the source code. AR covers recognizing different algorithms that carry out different computational task, as well as different types of algorithms that perform the same task. The main application of AR is in automatic assessment of students' programs to verify that the program implements the required algorithm.

This thesis investigates the applicability and usefulness of RoV in AR. The idea is to analyze different implementations of basic algorithms to see whether RoV appear in those algorithms is such a way that they can be distinguished on this basis. In addition to RoV, other distinctive characteristics that can be used in the recognition process are also identified from the algorithms. These characteristics are related to language constructs and various software metrics. Based on the results of these analyses, a method for AR is introduced and a tool for automatic algorithm recognition is developed.

Two experiments on sorting algorithms are carried out to illustrate the feasibility of the AR method. In the first experiment, sorting algorithms are recognized using a manually constructed decision tree. The second experiment uses the C4.5 algorithm to construct the decision tree. The results of the experiments (86% and 97.1% correctly recognized algorithms respectively) show the applicability of RoV and the other characteristics in AR problem. Moreover, the performance of the automatically constructed decision tree demonstrates that machine learning techniques are also suitable for AR task.

Keywords: algorithm recognition, program comprehension, program understanding, roles of variables, static program analysis

Preface

This thesis is the result of studies and research at the LETECH research group, Department of Computer Science and Engineering at Aalto University and was partially funded by the Academy of Finland.

I would like to thank my supervisor, Professor Lauri Malmi, for providing me the opportunity to do this work. I highly appreciate his valuable input, feedback and suggestions during the project. I am sincerely thankful to him.

I am also grateful to my instructor, Docent Ari Korhonen, for his insightful comments, helpful ideas and great discussions.

Finally, I would like to express my greatest gratitude to my family; my wife, Elmira, my daughter Ava and my son Arad, for their love, understanding and encouragement, as well as my parents for their support over the years.

Otaniemi, November, 19, 2010

Ahmad Taherkhani

List of publications and the contributions of the author

This thesis is based on the following publications [P1] - [P3]

[P1] Using Roles of Variables in Algorithm Recognition. Ahmad Taherkhani, Lauri Malmi, and Ari Korhonen (To appear). In: *Proceedings of the 9th Koli Calling International Conference on Computing Education Research*. Joensuu, Finland, October 29 – November 1, 2009.

This paper discusses roles of variables, their application to algorithm recognition and introduces a method for algorithm recognition. It also presents an extensive literature overview.

The author of this thesis is the corresponding author of the paper. He carried out the literature survey, formulated the method and wrote most part of the paper.

[P2] Recognizing Algorithms Using Language Constructs, Software Metrics and Roles of Variables: An Experiment with Sorting Algorithms, Ahmad Taherkhani, Ari Korhonen and Lauri Malmi (2010) *The Computer Journal 2010; doi: 10.1093/comjnl/bxq049*.

This paper describes the application of the algorithm recognition method to five sorting algorithms: Insertion sort, Bubble sort, Selection sort, Quicksort and Mergesort. It presents an experiment conducted to evaluate the performance of the method and discusses the results.

The author of this thesis is the corresponding author of the paper and performed data collection and analysis. He was also responsible for writing the major part of the paper.

[P3] Using Decision Tree Classifiers in Source Code Analysis to Recognize Algorithms: An Experiment with Sorting Algorithms. Ahmad Taherkhani **In review:** The Computer Journal 2010.

This paper presents a method for automated construction of a decision tree which is used in algorithm recognition. The paper describes an experiment conducted to illustrate the performance of the method. The accuracy of the decision tree is evaluated using leave-one-out cross-validation technique and the results are presented in the paper.

The author of this thesis is the sole author of this paper.

Contents

Chapter 1

Introduction

Roles of variables (RoV) [45] 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. RoV are originally introduced and applied to improve novices' understanding of variables and their usage. Algorithms, on the other hand, are well-defined computational procedures that take some value as input and produces some value as output [11]. Algorithms consist of specific instructions that should be performed in a specific order to achieve a specific goal. Based on these definitions of RoV and algorithms, the starting point for the work presented in this thesis was to investigate whether an algorithm can be recognized based on the variables used in its implementation is source code. The idea is to make a connection between the notion of specific patterns of variable usage (RoV) and the variables used to implement specific instructions in some language (algorithms).

1.1 Motivation

To help teachers assess students' work, specially in large courses, number of different automatic assessment tools are developed, including Boss [27], CourseMarker [23] and Web-CAT [13]. These tools are capable of performing various functionalities, such as verifying the correctness of programs, checking the program structure and implementation style, run time efficiency, etc. (see the survey by Ala-Mutka [1] for an overview of this field). The existing tools, however, are not capable of identifying different types of algorithms that carry out the same functionalities. For example, an automatic assessment tool verifies the type and correctness of a sorting algorithm by executing it with a predefined set of numbers and checking that the output is a set of the same numbers in the predefined and expected order. But, the tool cannot easily and reliably distinguish between different sorting algorithms, and thus cannot help in grading the task where students are asked to implement a particular type of sorting algorithm, such as Quicksort (A simple approach to discover the used algo-

rithm by these kind of tools would be to check some intermediate states, but this is clumsy and unreliable as students may very well implement the basic algorithm in slightly different ways, for example, by taking the pivot item from the left or right end in Quicksort). This is the main motivation of the work presented in this thesis; developing a method that can automatically recognize different types of algorithms.

Algorithm Recognition (AR) 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 [34] (tuning of existing algorithms or replacing them with more efficient ones), clone recognition [3, 32] (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 [56] (a source to source translation approach, which involves the abstract understanding of what the source program does).

1.2 Research questions

Our main goal is to investigate the usefulness of RoV in recognizing basic algorithms. We want to discover how distinctive factors RoV are in identifying algorithmic patterns and how valuable they are in automatic AR process. Thus, our main research question is the following:

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

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

Although they may prove to be useful, it seems unlikely that RoV alone are able to distinguish between basic algorithms reliably and accurately enough (e.g., with at least 80% of correctly identified algorithms), due to the following reasons. First, algorithms that solve different problems or different types of algorithms that solve the same problem may use variables that appear in the same roles. For example, Insertion sort and Bubble sort algorithms use variables that plays exactly the same roles (e.g., *steppers* as loop counters and *temporary* in swap operation. See Table 4.1 for definition of RoV). In these cases, RoV cannot differentiate between the algorithms. Second, with regard to the RoV, some algorithms may have different implementations. For example, while most implementation of Mergesort algorithm do not use any variable playing temporary role (because of merging operation), however, this role appears in some implementations of Mergesort algorithm. RoV are not distinguishing enough in these cases neither. Finally, RoV are cognitive concepts [5, 17]. Two observers may assign different roles to a variable appearing in the same

context, or more than one role may be considered appropriate for the same variable. However, an automatic role detection tool deals with roles as technical concepts, not cognitive. This implies that automatic detection of RoV in a way that is in accordance with those assigned by a human during initial manual data analysis cannot always be guaranteed. This has also a weakening effect on RoV as distinguishing factors.

To tackle this problem, in addition to RoV, we have to use other factors in AR as well. This leads us to the second research question:

2. *What set of algorithmic characteristics are useful in AR process?*

We will analyze basic algorithms to find a set of characteristics that are distinguishing enough to be used in AR. We then use these characteristics as the complementary set of distinctive factors along with RoV. We will also evaluate the value of these characteristics from AR point on view. As the analysis is extended to cover more and more fields of basic algorithms, this set of characteristics may be subject to change.

One way to evaluate the value of RoV and other characteristics of algorithms in AR is using machine learning techniques. Thus, the final research question is:

3. *How suitable machine learning techniques are in AR problem?*

We will deal with this question by using the C4.5 algorithm for constructing a decision tree that can guide the AR process. We will also investigate the applicability of the C4.5 algorithm by evaluating the performance of the resulted decision tree.

1.3 Structure of the thesis

This thesis is structured as follows. Chapter 2 discusses Program Comprehension (PC) and AR as research fields, as well as their relationship. The challenges of AR are also presented in the chapter. Chapter 3 presents an overview on the previous work on PC and briefly discusses the related research fields. Chapter 4 gives a definition of RoV along with an example and highlights their connection to PC. Chapter 5 explains decision tree classifiers. The AR method is presented in Chapter 6, followed by two experiments discussed in Chapter 7 along with the results. Finally, Chapter 8 discusses related issues, presents some conclusions that can be drawn from the work mapping them into the research question explained above and gives some directions for future work.

Chapter 2

Program Comprehension and Algorithm Recognition

In this chapter, we discuss *Program Comprehension* (PC) and *Algorithm Recognition* (AR) and explain their relationship.

2.1 Program comprehension

PC is a concept that consists of all activities, aspects, methods, and research fields and techniques that can be linked to the process of understanding programs. Activities relate to different actions that can be utilized to understand a program, such as reading the code, executing the program, debugging, investigating the documentation, etc. Aspects pertain to how PC is carried out, by a human or automatically. Methods include dynamic, static or hybrid manners of performing PC and finally research fields and techniques cover those research lines and methodologies that facilitate PC, such as reverse engineering and software visualization.

We discuss different methods and aspects of PC in the following. A discussion on PC research fields and techniques is presented in Chapter 3, where an overview of related work is given. As PC activities are somehow self-evident and non-relevant, we leave them out of the discussion.

2.1.1 Program comprehension methods

There are mainly two types of analyses used in PC task: dynamic analysis and static analysis. In dynamic analysis, the program is executed by some input and the program output is used to understand the program. In static analysis, the program is not executed, but the code is investigated in order to extract such information that can help understanding the program. Dynamic analysis is commonly used in automatic assessment tools to check the

correctness of students' works, whereas most PC research fields utilize static analysis (see Chapter 3). It is also possible to use combination of the two analyses in PC tasks.

Dynamic analysis provide exact information about the code and thus is very important from PC point of view. However, the provided information is limited by the input, as the input determines which path of the code to execute. Static analysis provide more comprehensive overview of the code an makes it possible to investigate the code from various points of views.

2.1.2 Program comprehension aspects

PC research field can be divided into *human PC* (HPC) and *automated PC* (APC).

Human program comprehension

The focus of HPC is to discover how humans understand programs. HPC research deals with PC from psychology of programming point of view and tries to answer to the questions related to the process of understanding programs, including:

1. What are those strategies used by humans when comprehending programs? Which ones are the most useful?
2. What kind of cognitive structures humans build/have when comprehending programs?
3. What kind of external representations are more helpful in the process of understanding?

PC is a process in which a human 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 [49]. *External representation* means how the target program is represented to the programmer. *Assimilation process* and *cognitive structure* are the two elements 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. 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 assimilation process, *top-down*, *bottom-up* or *integrated* strategies of building mental representation may be used.

In top-down strategy, the assimilation process starts by utilizing the knowledge about the domain of the program and proceeds to a more detailed levels in the code to verify the hypothesis formed based upon the domain. In bottom-up strategy, the assimilation process starts at a lower level of abstractions with individual code statements and proceeds to higher level of abstractions by grouping code statements. The final mental representation of the target program is constructed by repeating this process of chunking lower levels successively

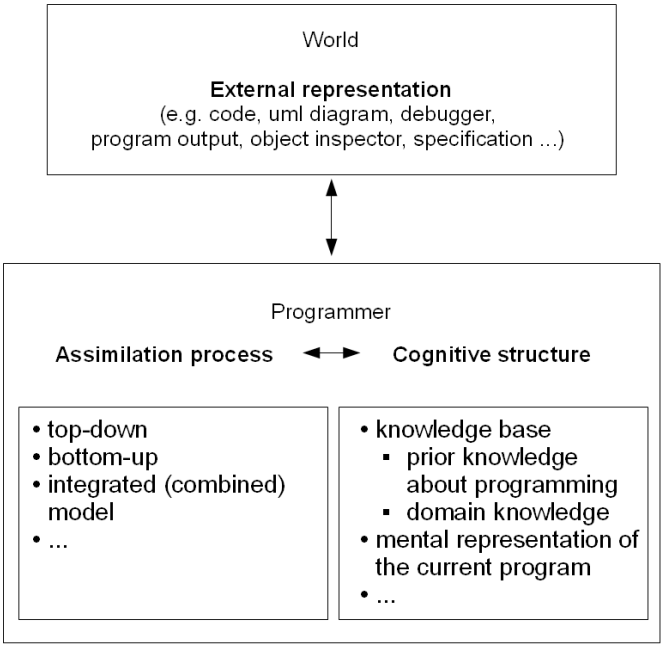

Figure 2.1: Key elements of program comprehension models [49]

to higher levels. In integrated strategy the programmer switches between the top-down and bottom-up models whenever he/she founds it necessary in order to build his/her mental representation effectively. In PC literature, integrated strategy is also referred to as combined, hybrid, opportunistic, as-needed or mixed strategy, sometimes interchangeably and sometimes with slightly different meaning (whose detailed discussion is out of the scope of this thesis).

Comprehending a program, regardless of the strategy recommended or suggested by PC models, involves elements that guide the comprehender and help him or her in the comprehension task. For example, in Soloway and Ehrlich's model [50]) model, *critical lines* are statements that carry important information about program plans and can be considered as the key representatives of the plans that help experts to recognize them. As another example, *beacons* are statements that indicate the existence of a particular structure or operation in the code. It has been shown that beacons play important roles in studying and understanding programs by experts [57]. Beacons are also described as providing the link between the source code and the process of verifying the hypotheses driven from the source code [7]. We will discuss these issues in a more detail in Chapter 4.

HPC has been studied for decades and several PC models have been suggested. The differences between PC models, as well as the difference between how they define novices and experts, are mainly related to the way they describe the cognitive structure and the assimilation process. We will explain some of the most popular PC models in Chapter 3

and discuss the differences between these models from various perspectives.

Automated program comprehension

APC is the aspect of PC that aims to automate the PC process by developing appropriate tools. The objective is, for example, to help humans in PC process by reducing the effort of reading the code.

APC research is closely related to HPC research. It uses the results of HPC research to develop more efficient tools that can assist programmers and maintainers in PC task. In her survey on theories, methods and tools in PC, Storey [52] notices that 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 in HPC 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" [52].

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 [52].

The knowledge extracted by PC tools from source code can be used for different purposes including restructuring programs, teaching novices, generating documentation from the code and finding the location of those parts of the code that can be reused [41].

2.2 Algorithm recognition

The task in AR is to identify algorithms from source code. Recognizing an algorithm involves discovering its functionality and comprehending the corresponding program code, thus AR can be regarded as a subfield of PC. 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. Like PC tools, AR tools can also 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. There exist different algorithms that perform the same computational task, such as sorting an array or finding the minimum spanning tree of a graph.

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

In addition to aforementioned variations that make AR a difficult and challenging task, there are also other issues that contribute to its complexity: in real-world programs, algorithms are not "pure algorithm code" as in textbook examples. They include calls to other functions, processing of application data and other activities related to the domain, which greatly increases the complexity of the recognition process. The implementation may include calls to other methods or the other functionalities may be inlined within the code [P2].

With respect to computational complexity, AR can be regarded to be comparable to many undecidable problems. As an example, it can be considered as similar to the problem of deciding the equivalency of syntactical definitions of programming languages, which is also known as the equivalency problem of context-free grammars, and as described in [22], is proven undecidable by Bar-Hillel et al. [2]. This problem is undecidable, because there exists no algorithm which can show in a finite amount of time, whether two given input set of syntactic rules are equivalent, that is, whether they define the same language. On the other hand, the problem of AR can be regarded to be a problem of deciding whether two given algorithms are equivalent, that is, whether they perform the same task or solve the same problem. In order to be able to decide whether two algorithms solve the same problem, the functionality of those algorithms must be understood first. This means that being able to tell whether two algorithms solve the same problem can be regarded equal to being able to tell what problem those two algorithms solve. Thus, AR and syntactical equivalence problem can be regarded to belong to the same category, and this implies that AR problem also can be considered as an undecidable problem. As we will describe in Chapter 6 when presenting the method, we approach the problem by converting it into the problem of extracting the characteristics of algorithms and examining algorithms as characteristic vectors. Furthermore, we limit the scope of our work to include a particular group of algorithms. In addition, we are not looking for a perfect matching, but aim at developing a method that provides statistically reasonable matching results.

Chapter 3

Related Work

PC research field has been mainly motivated by finding effective solutions to be used in software engineering tasks and by developing automated tools to facilitate understanding programs [52]. Figure 3.1 shows the research fields that are connected with PC. In the figure, the octagon with gray background depicts PC as a concept and the rounded rectangles with white background illustrates research fields that draw on PC literature or are otherwise related to PC.

In this section, we present a brief overview on the previous research on the two aspects of PC, HPC and APC, followed by a discussion on the related research fields depicted in Figure 3.1. We also analyze the relevance of each research field to the problem of AR.

3.1 Human program comprehension research

The goal of HPC research is to discover how a human understands program and what lies behind good and poor performance in the understanding task. HPC research is not an automatic process, and thus it is not directly relevant to AR. However, findings of HPC models can well be used in developing AR tools, as has been the case in APC tools. As also noted by Gerdt [16], there are several similar concepts, techniques and strategies between HPC and knowledge-based APC (an APC technique that will be discussed in Section 3.2). As an example, knowledge base of an APC tool comprises the prior knowledge of the system upon which the comprehension process is based. This corresponds to the knowledge base of a human in the comprehension process (see Figure 2.1). Likewise, the assimilation process presented in HPC models appears in APC tools as different techniques of matching plans from the target program against the plans in the database. Since HPC research line is started before APC, it can be concluded that APC draws on HPC, although this relationship is not explicitly highlighted in the APC literature.

In the study centered on PC literature and the inferences that can be made from PC studies and models for computing education, a vast body of PC literature was reviewed [49].

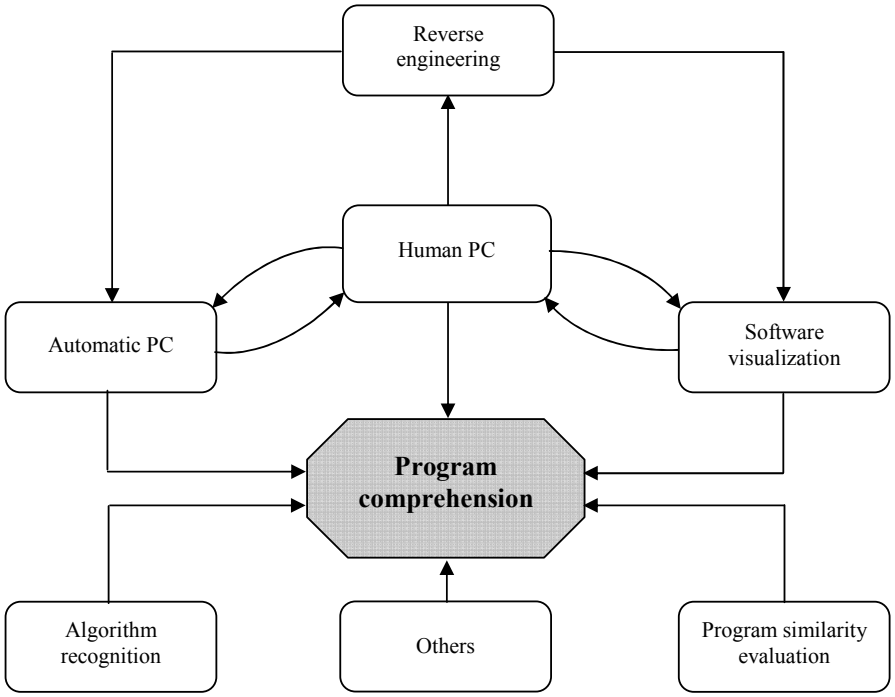

Figure 3.1: Different research fields that are connected to PC. Arrows illustrate the interrelationship between the fileds and should be read as "drawing on" or being somehow "related to" (from [P2])

Figure 3.2 shows all the considered PC models, from which the models with a gray background were selected for detailed analysis. These models were selected with the criterion that they represent main PC models and have had significant individual influence on the subsequent related research. We present a brief description of these models in the following (for more detailed discussion see [49]). In addition to a short overview for each model, the following description includes the elements shown in Figure 2.1, that is, external representation, assimilation process and cognitive structure. The selected models are presented in chronological order, starting from the earliest one.

Soloway and Ehrlich's model ([50]) is one of the earliest models of PC which is based on text comprehension research. In this model, program consist of plans: "generic program fragments that represent stereotypic action sequences in programming" ([50]). To form plans, rules of programming discourse are used, which are programming conventions. If typical plans and proper rules of programming discourse are used to write a program, the resulted program is readable and understandable (which the authors call a *plan-like* program). Using bad programming conventions and atypical plans result in a program that is difficult to comprehend (correspondingly called an *unplan-like* program). The model uses program code as external representation. To comprehend a program, a programmer reads the code and understands or assumes the plans used in the program. This is followed by a

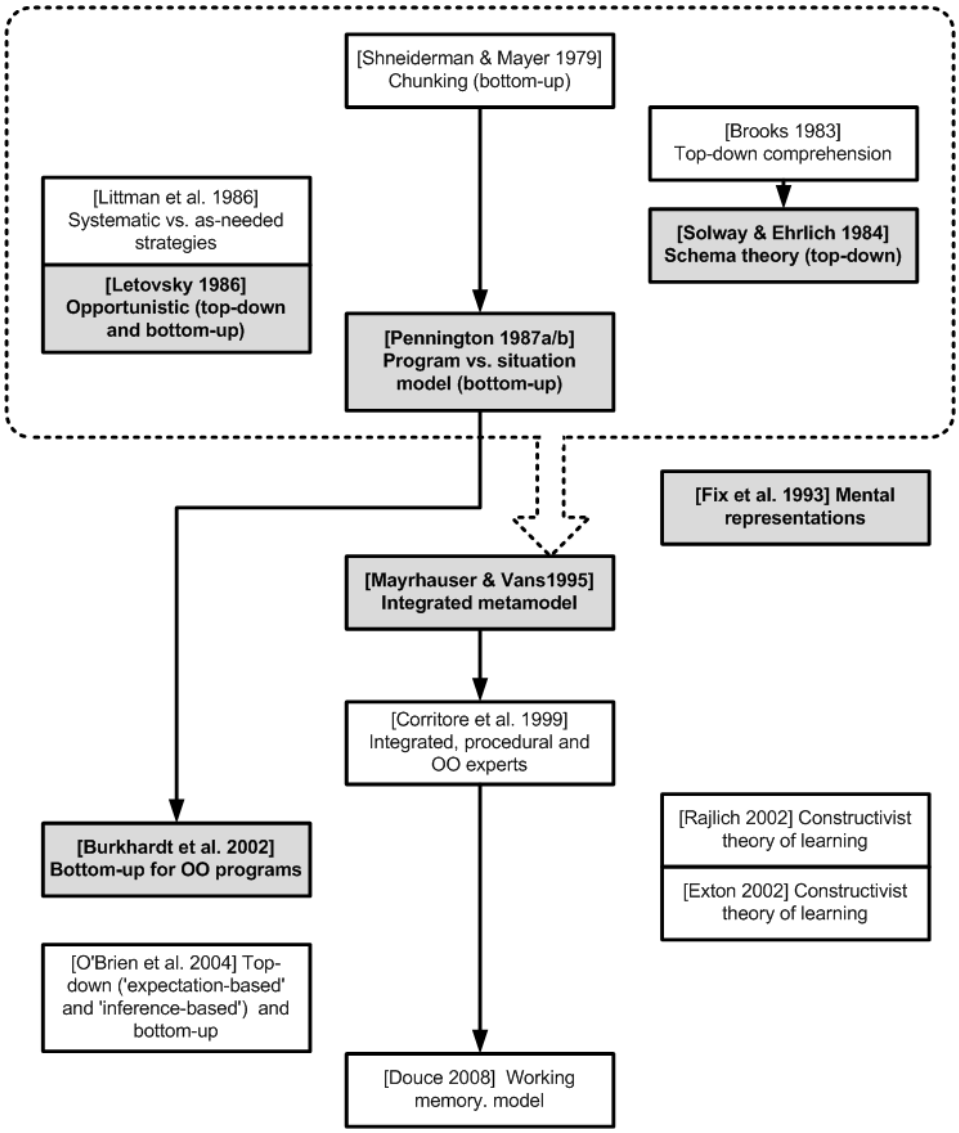

Figure 3.2: Most representative and individually influential PC models [49]. The models with a gray background are selected for detailed analysis. Arrows indicate direct antecedence or strong influence. The model of von Mayrhauser and Vans integrates the models surrounded by the dotted area

more detailed examination of the code, where the programmer verifies the assumed plans. Rules of programming discourse (e.g., using proper variable names), play an important role in this stage. The programmer also utilizes critical lines, which play the role of beacons in the comprehension process. Thus, the programmer's knowledge base consists of programming plans and rules of programming discourse. Based on an empirical study, Soloway and Ehrlich show that the difference between experts and novices is that experts have developed a more complete knowledge base, which is why they understand plan-like programs much more quickly and correctly than novices. When comprehending unplan-like programs, experts perform almost as poorly as novices. This is because unplan-like programs do not use typical plans and proper rules of programming discourse, and thus result in confusion and mislead experts when they try to recognize plans and verify their hypotheses.

Letovsky [31] proposed a PC model in which the assimilation process is opportunistic, that is, the programmer uses both top-down and bottom-up process in the comprehension task to achieve the best results. In this model, the comprehension is based on knowledge base. To comprehend programs, the programmer uses his/her knowledge base which consist of his/her background knowledge and expertise. During the comprehension task, the programmer builds a mental model which is his/her understanding of the target program so far, and which evolves as the comprehension proceeds. External representation comprises source code and program documentation. The empirical part of the Letovsky's study was carried out using six professional programmers with varying work experience. Thus, the author does not explicitly highlight the difference between experts and novices. However, it is evident that a comprehensive knowledge base is necessary to become an expert.

In **Pennington's model** [39, 40], which is a bottom-up model based on text comprehension, the mental model comprises a program model and a domain model (situation model). The program model is text based and comes from focusing on program's text structure and objects (control flow, etc.), while the domain model relates objects in the problem model to source-language entities, answering the question why the target program does what it does. These models are built sequentially, program model being built first. Cross-referencing combines the two model and links them and thus results in the best outcome. Programmers who construct either a program model or a domain model (i.e., novices) do not perform as well as experts who use cross-referencing. The external representations of Pennington's study were program code and some documentation about the domain. The prior knowledge of programmers consist of text structure knowledge and plan knowledge.

Fix, Wiedenbeck and Scholz's model [15] concentrates on the mental representation of computer programs. This model uses the aforementioned models of Soloway and Ehrlich [50], Letovsky [31] and Pennington [39, 40] to present the following five abstract features that distinguish novices form experts: hierarchical structure (decomposition of goals and sub-goals based upon Letovsky's model [31]), explicit mapping of code to program goals

(linking high-level goals to their code representation), foundation on recognition of recurring patterns (plan knowledge as introduced by Soloway and Ehrlich [50]), connection of knowledge (understanding how different parts of the program interact with each other), and grounding in the program text (ability to localize structures and operations in the code). The results of the experience conducted by the authors show that the mental representation that experts build from the code includes all the abstract features, while the mental representation of novices (although may appear to contain poorly developed elements of the features) does not. The external representation used in the study is program code presented on paper. The assimilation process consists of extracting information from the code based on the knowledge base, for example, knowledge about plans and the hierarchical structure of programs. Reading strategies can also influence the assimilation process.

Von Mayrhauser and Vans' model [54, 55] is an integrated model developed based upon the prior models of Soloway and Ehrlich [50], Letovsky [31], Pennington [39, 40] and others. The model includes four major components: "the top-down, situation and program models and the knowledge base. The first three reflect comprehension processes; the fourth is needed to successfully build the other three" [54]. According to the authors, programmers do not use only top-down or bottom-up comprehension strategy, but rather apply all the three model components, that is, program, situation and top-down (domain) models, and frequently switch between them to build their understanding. These components are applied at different levels of abstraction depending on code size. These results were found by conducting an experiment where programmers tried to understand a large scale software system using the program code as external representation. The authors also note that understanding of large scale programs requires significant domain information. The von Mayrhauser and Vans model is developed based on the performance of professional programmers in comprehension task, hence the performance of novices is not expressed explicitly. While the professional programmers used all the program, situation and top-down models and effectively switched between them to achieve the best understanding of the code, novices probably focus on one model and use less sophisticated assimilation process.

Burkhardt , Détienne and Wiedenbeck's model [8] is based on Pennington's model [39, 40] with consideration of object-oriented program features such as objects, relations between objects, message passing, larger program structures, etc. The authors studied how programmer expertise, nature of the task and development of comprehension over time affect PC. This effect was studied using two tasks: a read-to-recall task (documenting a program) and a read-to-reuse task (modifying a program). In the study, the external presentation of the program consisted of source code both in hard copy and on computer screen with some documentation. Like in Pennington's model, the assimilation process consisted of extracting information and building program and situation models. The cognitive structures of the model are also similar to those in Pennington's model, with additional elements

related to object-oriented programming taken into account. According to the authors, the difference between performance of novices and experts depends on the given task. In the read-to-recal task, novices focused on the program model more than the situation model, whereas in the read-to-reuse task, novices were able to develop a situation model over time. Therefore, with the read-to-reuse task the difference between experts and novices are smaller than with the read-to-recall task.

3.2 Automated program comprehension research

APC deals with understanding programs automatically and without reading the source code. Most APC techniques are knowledge-based, where the basic idea is to store stereotypical pieces of code – which are often called *plans*, but also *chunks*, *clichés* and *idioms* in different studies with slightly different meaning – in a database and match the target program against these pieces. Since the functionality of the plans in the database 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 HPC, there are three main techniques to perform matching: *top-down*, *bottom-up*, *hybrid* technique. Top-down techniques (see e.g., [24]) use the goal of the program to select the right plans from the knowledge base. This speeds up the process of selecting the plans and makes the matching more effective. However, the main disadvantage of these techniques is that they need the specification of the target program, which is not necessarily available in real life, especially in case of legacy systems. Most knowledge-based APC systems work bottom-up (see for example [21]), where the matching starts at the low-level of abstraction, with statements and small plans, and proceeds toward the higher-level of abstraction to discover the goal of the target program. The main concern with bottom-up techniques is efficiency. As the statement can be part of several different plans and the same plan can be part of different bigger plans, the process of matching statements and plans can get ineffective. This is especially true when the target program is a real life program with thousands of lines of code. Hybrid techniques use the combination of the two techniques. For example, in Quilici approach [41], plans are first recognized in a bottom-up manner. After this, general plans are suggested by the system to be matched against the program in a top-down manner. These general plans are proposed by using a well-organized plan library, where each plan is identified by an index, specialization and implication links to the other plans. By using the indexing facility, the system is able to quickly associate a piece of the source code with a plan in the knowledge base.

Among others, *fuzzy reasoning* technique [9] was introduced to improve the performance of the knowledge-based APC techniques and to address the problem of scalability and inefficiency related to them. Instead of matching all the statements and plans of the tar-

get program against the plans in the knowledge base, fuzzy reasoning technique first selects a set of more promising pieces of code, and performs the costly more detailed matching only between this set and the corresponding plans.

3.3 Reverse engineering techniques

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 complex 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 to gain a better understanding of a program enabling 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 [42].

Since providing abstract specifications and creating documentation from source code are the main outcomes of reverse engineering techniques, these techniques can be regarded as analysis methods of system structure rather than understanding its functionality. Thus their relevance to our work is not high.

3.4 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 and the main application for these tools have been detecting plagiarism in universities and preventing students from copying each other's works.

Based on how programs are analyzed, these techniques can be divided into two categories: *attribute-counting* techniques (see, for example, [19, 26]) and *structure-based* techniques (see, e.g., [35, 58]). Attribute-counting techniques use distinguishing characteristics of the subject program 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, whereas structure-based methods are generally regarded more tolerant to modifications imposed by

students to make two programs look different [35]. Structure-based methods can be further divided into string matching based systems and tree matching based systems.

Since the focus of the program similarity evaluation techniques is on the style and structure of a program rather than discovering its functionality, these techniques are not highly relevant to AR as such. However, as we will discuss in Chapter 6 when presenting our method, we used software metrics that are widely used in these techniques.

3.5 Others research fields

There are several other research methods related to HPC and APC.

Clone Detection (CD) techniques are used for locating clones in source code. Clone means the duplication of some piece of a source code, which is either intentionally copied by a programmer from somewhere else in the same software to be reused directly or with some small modifications, or is created by him or her without awareness of the existence of the same code elsewhere in the software. Since clones make the maintenance task more difficult, CD tools are valuable in software engineering tasks.

CD techniques are close to our research, as our purpose is to look for similar patterns of algorithmic code in the source code. However, there is an essential difference between AR and CD: in CD, the goal is to find similar or almost similar pieces of code within software and therefore, all kinds of identifiers that can provide any information in the process can be utilized. 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. However, our purpose, on the other hand, is to identify implementations of some predefined set of algorithms for human inspection that would support understanding the purpose of the code. We are not looking for the similarities such as comments an alike within the program, and thus, we cannot make use of them in the same way.

Traditionally, CD techniques are based on structural analyses such as structural organization, control and data flow as well as abstract syntax tree analysis (see for example [4]). Marcus and Maletic's technique [32] detect high-level clones by identifying the implementation of similar high-level concepts. Latent Semantic Indexing (LSI) is used as the information retrieval technique to statically analyze software and to identify semantic similarity (similar words). To detect clones, the method examines various files and documentations, as well as comments and identifiers.

Basit and Jarzabek [3] have developed a tool for detecting clones in a file (simple clone) and clones in different files (structural clones or design-level similarities). To detect simple clones, first the code is tokenized and then similarities in the token sequence are evaluated. After this, data mining techniques are used to find structural clones. This is carried out by

investigating the pattern of co-occurring simple clones in different files. The authors claim that the technique is capable of scaling up to handle big systems.

Software Visualization(SV) is a common name for techniques that help humans to better understand different aspects of software by using graphics and animation. SV tools are used in various purposes including education, software engineering tasks, etc. SV tools facilitate the process of PC by visualizing software at different level of abstraction. They also can contribute to PC by, for example, supporting different assimilation processes presented in PC models, namely top-down, bottom-up and integrated strategies [12]. Many studies in SV research field that aim to help novices draw on the PC literature (see, e.g., [37] and [48]). Although SV is not highly relevant to AR problem, SV techniques can be utilized to highlight the recognized algorithm from the source code. This further helps the user to locate the recognized algorithm in the source code and to understand its relation to the other parts of the code.

As an independent field, **Roles of variables** (RoV) can also be categorized as highly relevant to AR and HPC. With regard to AR, as we will describe in Chapter 6, RoV can be used as distinguishing factors to recognize algorithms. Concerning HPC, studies on using roles of variables in elementary programming courses have shown that roles of variables provide a conceptual framework for novices that helps them comprehend programs better [30]. Utilizing roles in teaching also help 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" [30]) 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" [30]). We will discuss RoV and their relation to PC in more detail in the next chapter.

Several other methods are also introduced for automatic or semi-automatic PC including *Program understanding based on constraints satisfaction* [59, 60], *Task oriented program understanding* [14], *Data-centered program understanding* [25], and *Understanding source code evolution* [38].

Chapter 4

Roles of Variables and Program Comprehension

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

4.1 Roles of variables

The concept of RoV was first introduced by Sajaniemi [45]. The idea behind them 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 argues, RoV are part of programming knowledge that have remained tacit. 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 can make it a valuable tool that can be used in teaching programming to novices by showing the different ways in which 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 with different purposes. In this work, we have extended the application of RoV by applying them in the problem of algorithm recognition.

Roles are cognitive concepts [5, 17], implying that human inspectors may have a different interpretation of the role of a single variable. However, as Gerdt [18] and Bishop and Johnson [6] describe in their work, roles can be analyzed automatically using data flow analysis and machine learning techniques.

As reported in [45], Sajaniemi identified nine roles that cover 99% of all variables

Role	Description
Stepper	A variable that systematically goes through a succession of values, for example, values stored in an array.
Temporary	A variable that holds a value for a short period of time appears in temporary role.
Most-wanted holder	A variable that holds a 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. The fixed value role can be thought as a final variable in Java which is immutable once it has been assigned a value.
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. For example, all Java data structures that implement Collection interface.
Walker	Can be assigned to a variable that is used for going through or traversing a data structure.

Table 4.1: The roles of variables and their descriptions (from [P1])

used in 109 novice-level procedural programs. Currently, based on a study on applying the roles in object-oriented, procedural and functional programming [46], a total of 11 roles are recognized. These roles are presented in Table 4.1 (see the RoV Home Page (`http://www.cs.joensuu.fi/~saja/var_roles/`) for a more comprehensive information on roles). Note that the three last roles shown in Table 4.1 are related to data structures.

4.1.1 An example

Figure 4.1 shows a typical implementation of Selection sort in Java. There are five variables in the algorithm with the following roles. A loop counter, that is, a variable of 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 roles. Variable *min* stores the position of the smallest element found 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

```
// i and j: steppers, min: most-wanted holder
// temp: temporary, numbers: organizer
01 for (int i = 0; i < numbers.length-1; i++){
02     int min = i;
03     for (int j = i+1; j < numbers.length; j++){
04            if (numbers[j] < numbers[min]){
05                   min = j;
06            }
07     }
08     int temp = numbers[min];
09     numbers[min] = numbers[i];
10     numbers[i] = temp;
11 }
```

Figure 4.1: An example of stepper, temporary, organizer and most-wanted holder roles in a Selection sort algorithm (from [P1])

figure demonstrates the temporary role. Finally, data structure *numbers* is an array that has the organizer role.

4.2 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 HPC research (e.g., Kuittinen and Sajaniemi's study [30] draws on Pennington's work [39]), no study about further explicit connection between the two, nor further application of RoV to HPC has been reported. Automatic role detection tools, such as [6], [16] and [18] can correspondingly be considered as related to APC research field. In this section, we first define *beacons*, which are important elements of PC models that help programmers in comprehension task. We then discuss how RoV can serve as beacons in PC.

PC models emphasize the role of beacons in PC. Beacons are parts of codes that help programmers to identify plans while reading the code. Wiedenbeck describes beacons as signs of presence of particular structure or operation in source code [57]. Brooks defines beacons as the link between the process of hypothesis verification and the source code [7]. As an example, the existence of a swap operation specially inside a pair of loops indicates sorting of array elements [7]. Soloway and Ehrlich [50]) emphasize the role of *critical lines* in verifying the hypothesis about plans. In the following, we explain critical lines and show that they are conceptually identical to beacons.

Critical lines are those lines of the programs that "carry the information that makes the programs plan-like or not" [50] (a plan-like program is "one that uses only typical programming plans and whose plans are composed so as to be consistent with rules of programming discourse" [50]). Figure 4.2 shows the two program in Algol language which

Soloway and Ehrlich used in their study on PC [50]. 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 trial, 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 trail. 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 agree with the search function, which is a minimum search function. Since, the variable name does not reflect its function, 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 these lines are 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 PC task and that critical lines are important in the process.

```
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 4.2: Plan-like and unplan-like programs used in Soloway and Ehrlich PC study [50]

Roughly speaking, line 9 in figure 4.2 and lines 4 and 5 in Figure 4.1 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 4.1, variable *min* in line 5 has the most-wanted holder role. Therefore, since line 9 in figure 4.2 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 [7] regards the presence of a swap operation as a beacon in sorting functions. Swap operations typically include temporary role and thus temporary role can be regarded as part of the beacon in the example of Fig-

ure 4.1. As we will discuss in this thesis, RoV constitute an essential part of our method in automatic AR. Specifically, presence of the most-wanted holder role and absence of the temporary role provide strong indicators (i.e., beacons) in recognizing Selection sort and Mergesort.

As discussed in the previous chapter, in a study on effects of teaching RoV in elementary programming courses Kuittinen and Sajaniemi [30] 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 [45], as a tool to be used in PC, RoV are inclusive and comprehensive as well.

From the above discussion, we conclude that RoV can be utilized in PC tasks as beacons. 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 argues [45], 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 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 in the previous chapter, the exact same difference between experts and novices applies with regard to other elements of PC models as well, such as beacons, critical lines, general programming knowledge and other elements of programmers' knowledge base (see Figure 2.1 and the discussion on different HPC models in Chapter 3).

Chapter 5

Decision Tree Classifiers

In this chapter, we first discuss the important issues about decision tree classifiers in general. After this, we explain the C4.5 decision tree classifier algorithm and discuss how these issues are dealt with in the C4.5 algorithm. The issues discussed in this chapter are intended to help the reader understand how decision trees, and in particular, the C4.5 algorithm work. Readers who are familiar with these topics may skip this chapter and proceed to the next one.

5.1 Decision tree classifiers in general

Decision tree classifiers, also called *classification trees* or simply *decision trees*, are used to classify different instances of a set into appropriate classes. Decision trees use *classification*, which is a subfield of machine learning methods. Machine learning methods can be divided into *supervised* and *unsupervised* learning. In supervised learning, to which classification also belongs, first a set of known instances, called *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*. In unsupervised machine learning method, there is no training set or previous learning. Instead, the input set is examined for finding regularities between the instances. From slightly different perspective, machine learning methods can be divided into *predictive* and *descriptive* methods. Predictive methods, such as classification, use some attributes to predict unknown values of other attributes, whereas descriptive methods, such as clustering, provide patterns to describe the data.

In a training set, each instance consists of a group of attributes that describe the 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.

Decision trees are often constructed using the divide and conquer principle. Instances of a training set are recursively divided to construct a tree, starting from the root going down in the tree toward the leaves. A decision tree consists of internal nodes (including the root node) and leaves. Each internal node contains a test that uses some attribute and results in splitting the data set into subsets based on the outcome of the test. 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* and the corresponding split is called a *binary split*. This implies that internal nodes have only two children. In *multivariate tree*, more than one attribute is tested in internal nodes [36].

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 of 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 the proper 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 and in the next section we explain them in more detail in connection with the C4.5 algorithm.

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 tree from the training set [36]), 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 [29]. 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* [36]. 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 [29, 36]. The explanation of all these measures is out of the scope of this thesis, but we will discuss information gain in more detail in relation to the C4.5 algorithm in the next section.

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 unseen data well.

It has been claimed that the quality of a decision tree depends more on the right size than the right split [36]. 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 [43]. In addition to higher accuracy, smaller trees are more comprehensible as well [28]. 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 ([29, 44]), selecting the right tests is very important in generating near-optimal trees.

In his survey on automatic construction of decision trees, Murthy [36] 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 method, and it has been shown in different studies that there is no single best pruning method that is superior to the others ([29, 36]). 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 [29].

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 [36].

5.2 C4.5 decision tree classifier

We chose the C4.5 algorithm 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 [29]. The C4.5 algorithm preserves the advantages of its predecessor, the ID3 algorithm, 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. In what follows, we explain how the C4.5 algorithm deals with the important issues related to building decision trees presented in the previous section. The

discussion in this section is based on the book about the C4.5 algorithm written by its inventor [44].

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 were 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.

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 [44].

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.

Chapter 6

Method

We have applied two methods in AR problem. The first method uses a manually constructed decision tree to recognize algorithms (we call it Manual Decision Tree method, i.e., MDT-method), while the second method uses a decision tree constructed by the C4.5 decision tree classifier to be used in the recognition process (which we call C4.5 Decision Tree method, i.e., CDT-method). Both methods consist of two phases. The first phase of the methods are essentially the same, with the difference mainly pertaining to the second phase, that is, the process of constructing the decision tree and recognizing algorithms. In this chapter we first describe the methods in terms of what is common between them and then present their differences.

6.1 Common phase of the methods

MDT-method and CDT-method are both based on static analysis of source code including *roles of variables* (RoV), language constructs and software metrics. We extract several characteristics related to statistics of language and metrics, which enable us to distinguish between different algorithms. These characteristics are converted into vectors, each vector describing the corresponding algorithm. Thus, the problem of AR is transformed to the problem of finding similarities and differences between characteristic vectors. The novelty of the methods is in using RoV as a characteristic of algorithms, more specifically, as beacons that indicate the type of algorithms. This issue was discussed in Chapter 4 and we will further elaborate on this later in Chapter 8.

We divided the characteristics of the algorithm into *numerical characteristics* (characteristics that can be expressed as positive integers) and *descriptive characteristics* (characteristics that describe some features of algorithms and are not presented as integers). These characteristics are shown in Table 6.1. Additionally, the last three characteristics in the table, which are related to the numerical and descriptive characteristics, are also computed. In this thesis, we will refer to these characteristics using the corresponding abbreviation

Numerical characteristics	**Description**
NAS	Number of assignment statements in the algorithm.
LoC	Lines of code.
MCC	McCabe complexity (i.e., cyclomatic complexity) [33].
N_1	Total number of operators in the algorithm.
N_2	Total number of operands in the algorithm.
n_1	Number of unique operators in the algorithm.
n_2	Number of unique operands in the algorithm.
N	Program length ($N = N_1 + N_2$).
n	Program vocabulary ($n = n_1 + n_2$).
NoV	Number of variables in the algorithm.
NoL	Number of loops. Supported loops are *for* loop, *while* loop and *do while* loop.
NoNL	Number of nested loops in the algorithm.
NoB	Number of blocks in the algorithm. A block refers to a sequence of statements wrapped in curly braces, for example, a method or a control structure (loops and conditionals).
Descriptive characteristics	**Description**
Recursive	Whether the algorithm uses recursion.
Tail recursive	Whether the algorithm is tail recursive.
Roles of variables	Roles of the variables of the algorithm.
Auxiliary array	Does the algorithm use an auxiliary array (for the algorithms that use arrays in their implementation).
Other characteristics	**Description**
Block/loop information	Information about blocks and loops, including starting and ending lines, length and interconnection between them (how they are positioned in relation to each other).
Loop counter information	Information about how the value of loop counters are initialized and updated. This is used to determine incrementing/decrementing loops (the value of the loop counter increases/decreases after each iteration).
Dependency information	Direct and indirect dependencies between variables (variable i is directly dependent on variable j, if i gets its value directly from j. If there is a third variable k on which j is directly or indirectly dependent, i also becomes indirectly dependent on k. A variable can be both directly and indirectly dependent on another one).

Table 6.1: The numerical, descriptive and other related characteristics computed from algorithms (from [P3])

shown in the first column of the table.

Characteristics N_1, N_2, n_1, n_2, N and n are Hasltead software complexity metrics [20] that are widely used in plagiarism detection tools (see, e.g., [19, 26]). The *Other characteristics* presented in the last three line of the table are used to identify different patterns in algorithms that can be utilized in the recognition process. We discuss these characteristics more in the next section when we explain the application of the method to sorting algorithms.

We implemented an Analyzer that computes all the characteristics of Table 6.1 for each algorithms and stores them in a database. Therefore, each algorithm is represented as a n-dimensional vector in the database where n is the number of characteristics. In order to be used in the process, descriptive characteristics are converted into numerical values. As an example, if an algorithm is recursive (or non-recursive), it has a value of 1 (or 0) in the corresponding database column.

Both MDT-method and CDT-method consist of two phases. In the first phase, different implementations of those algorithms supported by the tool are collected, analyzed, converted into characteristics vectors and stored in the database. In the second phase, a decision tree is constructed and the algorithm recognition is performed using this decision tree.

6.2 Description of the methods

After the first phase which is common between the MDT-method and the CDT-method, the methods use different ways to recognize algorithms.

Manual decision tree method

To perform the recognition, we manually construct a decision tree that includes a test on the characteristics in each node. The tests comprise both examination of the frequency of occurrences of the numerical characteristics in an algorithm, as well as investigation of the descriptive characteristics of the algorithm.

The algorithms stored in the database in the first phase are used to construct the manual decision tree and thus constitute the learning data. After converting the algorithms of the learning data into vectors of characteristics and storing them in the database and constructing the manual decision tree, the testing phase follows. In the testing phase, a set of previously unseen algorithms including both those algorithms of the same type as in the training set, as well as other types of algorithms or irrelevant code are analyzed. The Analyzer computes the numerical and descriptive characteristics of each test sample, converts it into characteristic vectors and stores it in the database. In the next step, the Analyzer retrieves the information of the algorithms of the learning data from the database

and calculates the minimum and maximum limits of the numerical characteristics from this information. If the value of a numerical characteristic of a test sample does not fit into the range of the minimum and maximum values of the corresponding characteristic calculated from the learning data, the test sample is labeled as *Unknown*. In addition, a message is shown to the user indicating the numerical characteristic that does not fit into the permitted range, as well as whether its value is smaller or greater than the corresponding numerical characteristic of the learning data. Finally, the information of the test sample is stored in the database and the recognition process for this test sample is ended. Otherwise, if the test sample passes this stage, the examination process continues with examining the descriptive characteristics according to the steps given by the corresponding decision tree up to a leaf node (a decision tree for recognizing sorting algorithms is shown in Figure 7.1 in the next chapter). Depending on the results of this examination, the target algorithm (i.e., the test sample) is labeled with the type of the algorithm indicated by the leaf of the decision tree to which the target algorithm has ended up. The information of the target algorithm is stored in the database in this case as well.

It is possible that a legitimate algorithm of the testing data is incorrectly labeled as Unknown, for example, due to a poor implementation style (which may lead to greater or smaller number of numerical characteristics). In this cases, a user can always examine the database manually to see whether the algorithm is incorrectly labeled as Unknown and if so, correct its type. This way the knowledge base of the system can be extended. Next time, the similar algorithms will be labeled correctly. For more detailed explanation of the MDT-method see [P2].

C4.5 decision tree method

In the CDT-method, a decision tree is constructed using the C4.5 algorithm based on the data set consisting of the algorithms of the supported types. Next, the performance of the decision tree is empirically evaluated using leave-one-out cross-validation technique. We will explain the process of constructing the decision tree and its evaluation in more detail in Chapter 7, where we describe the related experiment conducted for recognizing sorting algorithms.

6.3 Application to sorting algorithms

In this section we further elaborate on the method by explaining its application to five common sorting algorithms: Insertion sort, Bubble sort, Selection sort, Quicksort and Merge-sort. Sorting algorithms are suitable for this purpose for the following reasons. They are widely discussed topic in computing education. They are used as examples in programming courses as well as in courses on data structure and algorithms. Consequently, large number

Characteristic	Description
MWH	Whether the algorithm includes a most-wanted holder role.
TEMP	Whether the algorithm includes a temporary role.
In-place	Whether the algorithm needs extra memory.
OIID	(Outer Incrementing Inner Decrementing) Whether from the two nested loops the outer is incrementing and the inner is decrementing.
IITO	(Inner Initialized To Outer) Whether from the two nested loops the inner loop counter is initialized to the value of the outer loop counter.

Table 6.2: The descriptive characteristics specific to the sorting algorithms (from [P3])

of sorting algorithms are easily accessible from textbooks, the Web, students' work as well as in handouts of the related courses. Sorting algorithms are also easy to analyze as they are not too complex. Moreover, there are many different sorting algorithms that perform the same computational task. This feature makes discriminating sorting algorithms a challenging task. Finally and most importantly, as we will discuss later, sorting algorithms include both very similar and yet very different algorithms with respect to their characteristics. As an example, Insertion sort and Bubble sort algorithms are very similar, while Mergesort and Bubble sort algorithms are quite different. This is a very beneficial feature of sorting algorithms from the perspective of our method, since it results in the performance of the method to be tested more thoroughly.

In addition to the characteristics presented in Table 6.1, we computed the descriptive characteristics of Table 6.2 which can be used in the process of recognizing the sorting algorithms. These characteristics can be easily computed based on those shown in Table 6.1. *OIID* and *IITO* are computed using the characteristic *NoNL*, *Loop counter information* and *Dependency information*. Moreover, absence of an *Auxiliary array* presented in Table 6.1 implies that the corresponding sorting algorithm is an *in-place* algorithm. Finally, by examining the roles of the variables in the target algorithm detected by an automatic role detector, the existence of the roles *MWH* and *TEMP* can easily be found out (see Table 4.1 for the definition of the roles).

At the beginning of our study, we manually analyzed a number of different versions of the five aforementioned sorting algorithms. Based on the results of these analyses, we posited a hypothesis that the numerical and descriptive characteristics shown in Tables 6.1 and 6.2 are sufficient to describe the sorting algorithms and could be used to identify them. The problem that we were trying to solve was whether a new unknown algorithm from the testing data could be reliably enough identified by comparing its characteristics with the corresponding information of the algorithms from the learning data.

Next, we developed a prototype Analyzer that is able to automatically compute all the data and convert algorithms into characteristic vectors. The Analyzer is implemented in Java and the current version is able to process source code written in Java. It parses the

code, calculates all its numerical and descriptive characteristics shown in Table 6.1 and 6.2 and analyzes all the related data. The Analyzer stores the data into a database consisting of the following four tables: *Algorithm*, *Block*, *Variable*, and *Dependency*. We used an automatic role detection software to detect roles [6].

Table 6.3 shows the numerical characteristics computed from the collected sorting algorithms. For each characteristic in the table, the first number shows the minimum value of the characteristic for the corresponding algorithm, and the second number depicts the maximum value. For each sorting algorithm, the first row shows the results of applying the MDT-method (which we call MDT-experiment), and the second row shows the results of applying the CDT-method (respectively called CDT-experiment). Note that the difference between the values of numerical characteristics in two rows is due to the different methods we used in each experiments with regard to the data. In the MDT-experiment, we used *Holdout Method* [53], where we first randomly collected the learning data consisting of 70 samples for analysis, then we manually constructed a decision tree based on this analysis, and in the next step we collected separately a total of 217 more samples as testing data to carry out the experiment. Thus, the numerical characteristics shown in the first row of Table 6.3 are resulted from 70 samples of the learning data. In the case of the CDT-experiment, we collected 209 different sorting algorithms as the data set, ran them through the Analyzer and then fed the resulted characteristics to the C4.5 algorithm and finally, used leave-one-out cross-validation technique to evaluate the performance of the decision tree constructed by the C4.5 algorithm. Therefore, the numerical characteristics shown in the second row of Table 6.3 are resulted from 209 samples of the data set. We discuss the process of data collection in the next chapter where we present the experiments and discuss the results.

A quick glance at Table 6.3 reveals that these numerical characteristics divide the five sorting algorithms into two groups, a group with smaller number of numerical characteristics consisting of the Insertion sort, Bubble sort and Selection sort algorithms, and a group with larger number of numerical characteristics comprising the Quicksort and Mergesort algorithms. Therefore, the recursive and non-recursive sorting algorithms can already be differentiated by their numerical characteristics alone. The decision trees presented in Figures 7.1 and 7.4 in the next chapter illustrate the mechanism of recognizing the aforementioned types of the sorting algorithms. For detailed explanation on applying the method to sorting algorithms see the original articles [P2, P3].

In the next chapter, we further describe the application of the methods to the aforementioned sorting algorithms along with the results in the context of the experiments that we conducted. In the following, we describe the role detection software that we used.

Algorithm	**NAS**	**LoC**	**MCC**	N_1	N_2	n_1	n_2	N	n
Insertion	8/11	13/21	4/6	40/57	47/58	3/6	2/4	87/115	5/10
	8/12	13/32	4/6	40/69	47/69	3/9	2/5	87/138	5/14
Selection	10/12	16/25	4/5	47/61	51/64	4/6	2/5	98/125	6/11
	10/14	16/31	4/5	42/68	47/73	4/9	2/9	89/141	6/18
Bubble	8/11	15/21	4/5	46/55	49/57	4/6	2/4	95/112	6/10
	8/13	15/30	4/6	37/71	45/72	3/12	2/5	82/143	5/17
Quicksort	6/15	31/41	4/10	84/112	77/98	9/17	2/7	161/210	11/24
	6/19	26/57	4/14	72/152	69/138	7/24	2/12	141/290	9/36
Mergesort	14/22	33/47	6/8	96/144	94/135	11/16	5/10	190/279	16/26
	11/27	27/56	6/10	94/159	84/146	10/26	3/14	178/305	13/40

Table 6.3: The numerical characteristics of the 70 sorting algorithms of the learning data in the MDT-experiment (first row) and of the 209 sorting algorithms of the data set in the CDT-experiment (second row). The first number indicates the minimum and the second number the maximum of the value of the corresponding characteristic (see Table 6.1 for explanation on the abbreviations)

6.4 The tool for detecting roles of variables

A tool developed by Bishop and Johnson for automatic detection of roles of variables [6] is integrated into the Analyzer. The tool detects roles using program analysis techniques, particularly program slicing and data flow analysis. First, all occurrences of each variable in the program are captured. The outcome of this analysis is the program slice for each variable. This is followed by data flow analysis for each program slice. Based on the initial analysis of the example programs, the tool associates each role with a set of assignments and usage conditions. To detect roles, the tool compares the assignments and usage conditions of each variable of the target program with these predefined sets. If the user has provided a role for a variable, the tool checks whether all corresponding conditions for the provided role are met by the corresponding variable. If so, the tool confirms that the role provided by the user is correct. If the conditions are not met, the tool prints the role it believes to be correct and justifies its decision by giving an appropriate message. If there is no role suggested by the user, the tool simply prints the role it considers the most appropriate for the variable in question. More detailed description of the assignments, usage conditions and how the role detector works is beyond the scope of this thesis. For more information see [6].

Bishop and Johnson have developed their role detector for educational purposes. Therefore, the tool allows users to provide a role for a variable. Although providing a role for a variable is optional, special tags along with the name of variable must be provided for each variable, otherwise the tool will not consider the variable. The tool can be further developed so that the tags and name of variables are provided automatically. When preparing the data

for our experiments (see Chapter 7), we also assigned a role to all the variables to be able to detect the possible differences between the roles generated by the tool and those that we believed to be correct. All the roles, however, were detected automatically.

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

Chapter 7

Experiments and Results

In this chapter we explain the two experiments [P2, P3] we conducted on the five sorting algorithms: Insertion sort, Bubble sort, Selection sort, Quicksort and Mergesort. The first experiment is conducted using the MDT-method, where the decision tree is constructed manually, and we call it MDT-experiment. The second experiment is based on the CDT-method and uses a decision tree automatically constructed by the C4.5 algorithm, and is called CDT-experiment. The results of each experiment are also presented after the description of the experiment. In this chapter, the process of data collection and preparation as well as presentation of the results are described briefly. For more detailed explanations, see the original articles [P2, P3].

7.1 Manual decision tree experiment

In this section, we describe the different phases of the experiment including data collection, data preparation, manual decision tree construction and the results.

7.1.1 Data

A total of 287 algorithms (both for learning and testing data) were collected for the experiment, without any preference for particular sources. The algorithms were collected from various textbooks on data structures and algorithms, as well as from course materials available on the Web. Some of the Insertion sort and Quicksort algorithms were from authentic student work.

The collected algorithms, both the learning and testing data, were investigated on the source code level, they were run and their claimed type and correctness were verified. If the algorithms included non-relevant code (e.g., printing statements, code related to interface, etc.), these extra code was removed since our method, at its present state, cannot process these kinds of application data. However, the implementation of the algorithm itself was

Algorithm	Learning data	Testing data
Insertion sort	17 (24%)	35 (16%)
Bubble sort	11 (16%)	30 (14%)
Selection sort	14 (20%)	29 (13%)
Quicksort	16 (23%)	23 (11%)
Mergesort	12 (17%)	22 (10%)
Other	-	78 (36%)
Total	70	217

Table 7.1: The number and the percentage of each sorting algorithm in the learning and testing data (from P2)

left untouched.

The role detection tool that we integrated into the Analyzer (developed by C. Bishop and C. G. Johnson [6]) requires that all variables of the program are annotated using special tags (for more information about the tool, see Section 6.4). We analyzed all variables in all algorithms and assigned a suitable role to them along with their name and the required tags. Although providing the roles is not required by the tool, we did it to be able to easily track the cases where the failure in recognizing an algorithm has been due to the failure in detecting roles correctly. We used this information to report the reasons for the incorrectly recognized test cases.

Learning data

This experiment consists of the two following phases. The first phase includes running the algorithms of the learning data, analyzing the resulted characteristics and constructing a manual decision tree based on this analysis. The second phase involves running the algorithms of the testing data by the Analyzer and evaluating the accuracy of the method and the decision tree.

In the first phase, we collected 70 algorithms as the learning data. The learning data included only the five types of aforementioned sorting algorithms. The numerical and descriptive characteristics of the algorithms computed by the Analyzer were stored in the database, along with the actual type of the respective algorithm. The number and the percentage of each type of the analyzed algorithms in the learning data is shown in the second column of Table 7.1. In the table, the first number indicates the number of the corresponding algorithm, and the second number in the parentheses shows the percentage of the algorithm from the total number of the algorithms of the learning data. In the process of collecting the learning data, the criterion was that all algorithms should have an adequate number of representatives in the learning data and the exact number of each algorithm in comparison to other algorithms was not particularly planned.

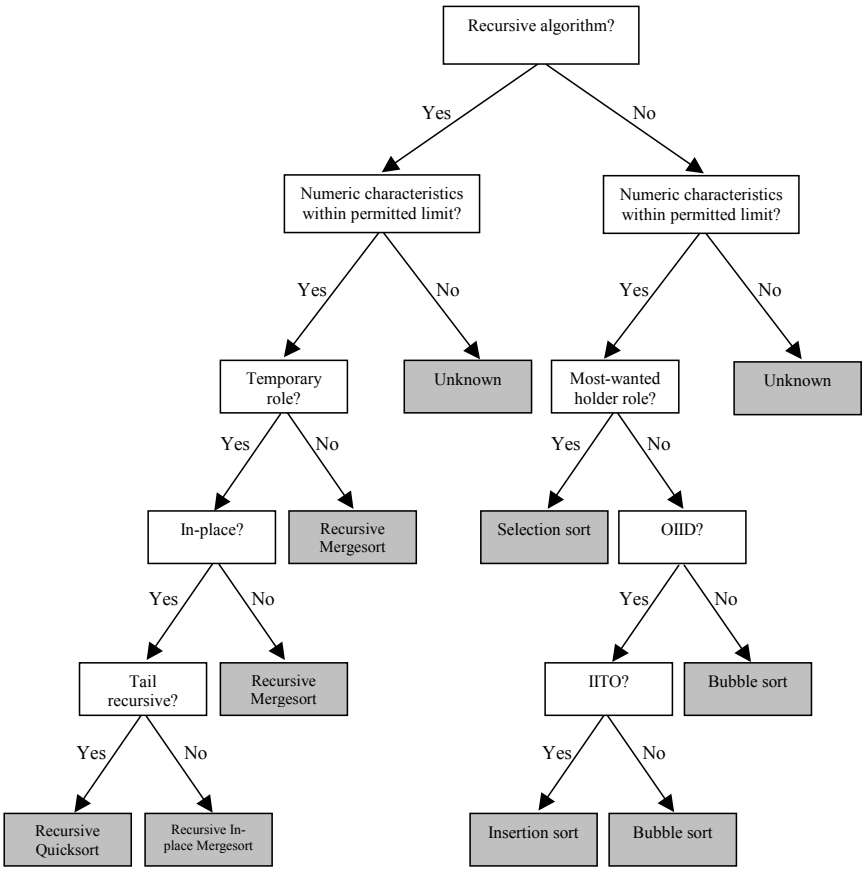

Figure 7.1: The manually constructed decision tree for recognizing the type of the five sorting algorithms in the MDT-experience (from [P2])

Decision tree

Based on the learning data, we constructed the decision tree of Figure 7.1 for classifying the sorting algorithms. The decision tree has nine internal nodes (including the root) and ten leaves. The internal nodes are represented by rectangles with white background and the leaves are depicted by rectangles with gray background. Each internal node includes a test. The outcome of a test in an internal node determines which child of that node will be visited next. There are arcs from each internal node to its children (or to a leaf) annotated by the outcome of the test in the corresponding node. Each leaf is labeled by a type of the sorting algorithms, or by the word "Unknown".

Characteristic "Recursive algorithm?" is the best discriminator and thus is used in the root. It divides the sorting algorithms into two groups, that is, recursive and non-recursive, with the accuracy of 100%, as all the Quicksort and Mergesort algorithms of the learning data are recursive and all the Insertion sort, Bubble sort and Selection sort algorithms are

non-recursive[1]. The tests in both children of the root examine whether the numerical characteristics of the recognizable algorithm are within those values retrieved from the database for the recursive or non-recursive algorithms. If the numerical characteristics are not within the permitted limit, the algorithm is labeled with "Unknown" and the process of recognition is terminated. Otherwise, in the case of recursive algorithms, the next visited node is the one labeled with "Temporary role?", where the performed test is whether the algorithm includes a variable with a temporary role. Quicksort algorithms typically include the temporary role, which appears in connection with swap operations to rearrange the elements of an array. However, Mergesort algorithms do not usually include a temporary role, since due to performing merge operation, there is no need to perform swap operations. Nevertheless, further investigation of the analyzed Mergesort algorithm implementations revealed that some of them do use swap operations and thus include the temporary role. Therefore, if the outcome of the test is positive, the process is continued by visiting the left child of the node, which is labeled with "In-place?". The test performed in this node is whether the recursive algorithm performs the sorting without making use of an auxiliary array. Again, using an auxiliary array is typical for Mergesort algorithms, but does not appear in Quicksort algorithms. However, some of the Mergesort algorithms work In-place and thus there is a need to perform another test to distinguish them from the Quicksort algorithms. Next, the left child is visited, which is the last node of this subtree and examines whether the recursive algorithm is tail recursive. All the analyzed Quicksorts are tail recursive, which is not the case with the analyzed Mergesort algorithms.

On the right subtree of the root, the three non-recursive algorithms are examined. If the numerical characteristics of the recognizable non-recursive algorithm are within the permitted limit, the left child is visited with testing the existence of a variable with a most-wanted holder role. Analyzing the non-recursive sorting algorithms of the learning data showed that all the Selection sort algorithms include the most-wanted holder role, whereas none of the Insertion sort and Bubble sort algorithms do. This is because Insertion sort and Bubble sort algorithms do not include selection of a minimum (or a maximum) element from a list (see Table 4.1 for the definitions of the roles). Thus, presence of the most-wanted holder role in the analyzed type of sorting algorithms is a strong indication (i.e., beacon) of a Selection sort. If the target non-recursive algorithm does not include the most-wanted holder role, the next visited node is the node labeled with "OIID?", which stands for "Outer loop Incrementing Inner Decrementing". All the Insertion sort and Bubble sort algorithms included two nested loops. In the analyzed Insertion sort algorithms, while stepping through the elements of an array, the value of the loop counter of the outer loop increases, whereas the value of the loop counter of the inner loop decreases. Although

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

the Bubble sort algorithms mostly work so that the outer loop is decrementing and the inner loop is incrementing, with regard to OIID characteristic, some of them work like the Insertion sort algorithms. Therefore, further test is needed to distinguish these kind of the Bubble sort algorithm implementations from the Insertion sort algorithms. With positive outcome of the test performed in this node, the next node to visit is the node labeled with "IITO?", which is an abbreviation for "Inner loop counter Initialized To Outer loop counter's value". As shown in Figure 7.1, this is the case in the Insertion sort algorithms. But in Bubble sort algorithms, the inner loop counter is initialized whether to zero or to the length of the array.

Note that the node labeled with "Numeric characteristics within permitted limit?" does not include a single characteristic of an algorithm, but rather the test for investigating all the numerical characteristics of the algorithm. It filters out those algorithms with less or more numerical characteristics, labels them with the Unknown type, stores them in the database, and gives a message about the particular characteristic(s) that does not fit within the permitted limits. If these filtering nodes are not taken into account, the number of internal nodes that include a test using a single characteristic would be seven, and the maximum depth of the tree, which is five, would be reduced to four.

Characteristics like OIID or IITO may appear very specific lacking the value of generalizability. They are indeed specific. They are, for example, not applicable to the Quicksort algorithms nor to the Mergesort algorithms, because these algorithms either do not have two nested loops or they cannot be separated by these characteristics. Moreover, the fact that these characteristics are located near the leaves, reflects their specificity. These characteristics, however, provide a valuable mean to separate the Insertion sort algorithms from the Bubble sort algorithms. Specially, the role of IITO is crucial in this regard. In terms of the other characteristics, these two algorithms are so similar that it seems very difficult to otherwise differentiate between them (see Table 6.3 and Figure 7.1). While these characteristics are specifically computed to be used in recognition of the sorting algorithms, their applicability as good discriminators for other fields of algorithms remains to be seen in future work.

As can be seen from Figure 7.1, the roles of variables play an important and distinctive role in the process. Specially the most-wanted holder role that separates the Selection sort algorithms from the Insertion sort and Bubble sort algorithms is highly important, since the other characteristics of these algorithms are quite similar and thus cannot separate them reliably (see Table 6.3).

After conducting our second experiment, which we will describe in the next section, we noticed from the results that the manually constructed decision tree of Figure 7.1 is not optimal. The automatic decision tree constructed by the C4.5 algorithm in our second experiment (illustrated in Figure 7.4) showed that Quicksort and Mergesort algorithms can be

distinguished solely based on "Tail recursive" characteristic and thus there is no need to examine the characteristics "Temporary" and "In-place". Correspondingly, with regard to the non-recursive sorting algorithms, the automatic decision tree of Figure 7.4 showed that the Bubble sort and Insertion sort algorithms can be distinguished using only the characteristic "IITO", and thus the characteristic "OIID" can be left out of the process. Therefore, the automatic decision tree is more optimal and simple with regard to the size, and as discussed in Chapter 5, it has been claimed that in terms of the quality, the right size of a decision tree is more important that the right split [36]. This comparison between the manual and automatic decision trees with regard to the optimality, simplicity and comprehensibility shows that in algorithm recognition problem, even in a relatively simple task of recognizing sorting algorithms, and with a fairly small data set as we used, the decision tree should be constructed using machine learning methods. Therefore, the manual decision tree is more like an educational example that depicts the steps that recognition of the sorting algorithms may consist of, and although it performs with a reasonable accuracy (86% of correctly identified true positive and true negative cases, as will be discussed in the following), its value resides in the fact that it illustrates the justification of using machine learning methods in constructing the decision tree to classify algorithms, and exemplifies their applicability and suitability in the task. For larger data sets and a more comprehensive set of algorithms, the necessity of using an automatically constructed decision tree seems evident.

Testing data

In the second phase, 217 algorithms were collected as testing data. The testing data mainly consisted of the five sorting algorithms, but included other algorithms as well. We refer to these algorithms as *Other*. The number and percentage of different algorithms of the testing data is shown in the third column of Table 7.1.

The Other algorithms were collected so that they do not differ greatly from the five sorting algorithm in terms of the numerical characteristics. The majority of these algorithms were other sorting algorithms such as Heapsort, Shell sort, Radix sort, Topological sorting, etc. The Other algorithms also included various enhanced types of the five sorting algorithms. These enhanced sorting algorithms include some additional code that changes the essence of the corresponding algorithms. As the result, the enhanced algorithms cannot be regarded as the basic form of the original algorithms. A typical example of these enhanced algorithms is the hybrid algorithm of Quicksort and Insertion sort, where sorting is carried out using Quicksort as long as the number of elements to be sorted is above some prefixed number (e.g., 10), and below this number Insertion sort is used. Another example is Cocktail sort, also known as Bidirectional Bubble sort, which is basically a Bubble sort that uses an additional loop and conditional statements to speed up sorting. Moreover, the Other algorithms contained some completely different algorithms, including binary search

algorithms, or purely application data, such as small parts of interface code or code written to test how some algorithm works.

This method of separating learning and testing data is commonly known as *Holdout Method* [53]. In this method, the proportion of the number of samples used as the training and testing set can be decided by the person who conducts the experiment. In order to avoid overfitting problem and achieve the best generalization performance, machine learning and classification methods often follow a commonly used pattern to split data set into training and testing set. For example, the proportion of 50-50 or 70 percent for training set and 30 percent for testing set is widely used. We use a comprehensive testing data in order to evaluate the performance of the method as accurately as possible.

7.1.2 Testing strategy

For testing the performance of the method and decision tree, we used the following strategy. We measured the number of the *True Positive (TP)* (indicates a case where the Analyzer correctly recognizes an algorithm that belongs to the target set of the five sorting algorithms), *True Negative (TN)* (correspondingly indicates rejecting an algorithm that is not among the target set), *False Positive (FP)* (denotes that an algorithm not belonging to the target set is incorrectly recognized as one belonging to the set) and *False Negative (FN)* (correspondingly an algorithm belonging to the set, which is not recognized as such or is recognized as another member of the set) cases. TP and TN cases indicate successful passing of tests and accurate performance of the Analyzer, whereas FP and FN cases mean failure in tests and poor performance of the Analyzer. In the following, we further illustrate these cases with examples and outputs of the Analyzer in each case.

An example of TP is a Quicksort recognized as such. In these cases, the Analyzer simply outputs *"The algorithm is a Quicksort"*. If a binary search algorithm, for example, is recognized as Unknown, a TN case is in question. In this case, the message would be something like the following: *"The algorithm seems to be a recursive algorithm, that has the following characteristics out of the permitted limit: Number of operators is below the permitted limit, Number of operands is below the permitted limit, Assignment statement is below the permitted limit, Line of code is below the permitted limit, Number of loops is below the permitted limit, Program length is below the permitted limit"*. If, on the other hand, a Quickselect is falsely recognized as a Quicksort, we have the FP case with the Analyzer providing the following message: *"The algorithm is a Quicksort"*. As we will discuss below, the FP cases are very rare. Finally, an example of FN is when a Quicksort is falsely recognized as a Mergesort, for example, due to the failure of the role detector to recognize temporary role (see Figure 7.1). The output of the Analyzer in this case would be *"The algorithm is a Mergesort"*. Another typical example of the FN cases is when a Selection sort algorithm is not recognized as such (i.e., recognized as Unknown) because

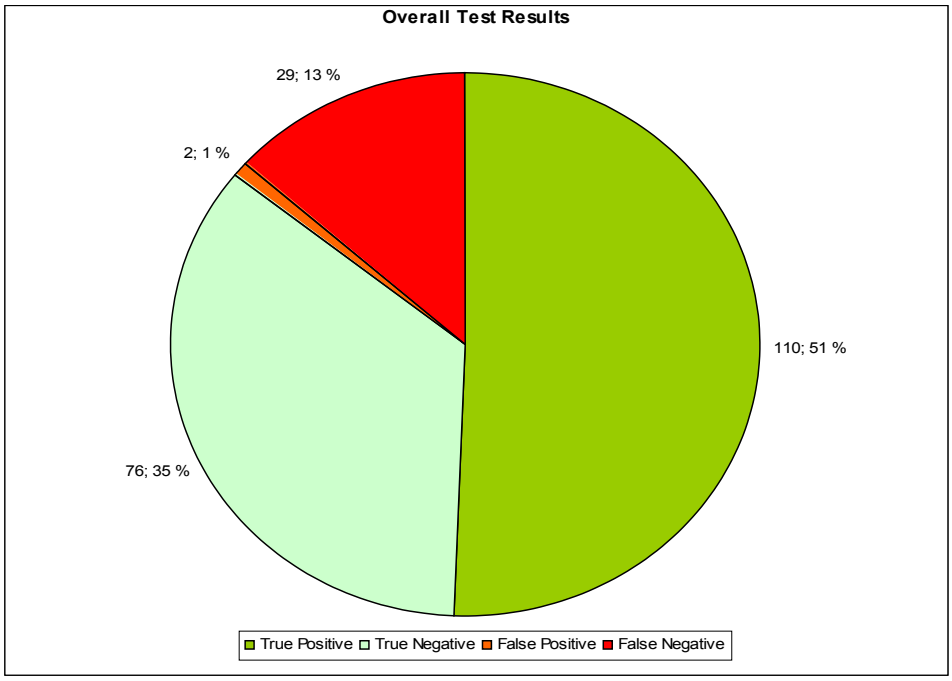

Figure 7.2: The number and percentage of the TP, TN, FP and FN cases from the overall results (from [P2])

some of its numerical characteristics are below or above of those Selection sorts known to the Analyzer (i.e., those which exist in the learning data).

7.1.3 Results

From the 217 sorting algorithms tested in the experiment, 186 algorithms (i.e., 86%) were identified successfully (TP and TN cases) and 31 algorithms were identified erroneously (FP and FN cases). These results are depicted in Figure 7.2. As the figure shows, out of the 31 falsely identified cases, 29 cases are FN that can be resulted from, for example, idiosyncratic or different implementation style than those in the learning data. The interesting observation is that out of the 31 falsely identified test cases, only 2 cases are FP (i.e., less than one percent of the testing data): a Heapsort algorithm identified as Mergesort and a Quickselect algorithm recognized as Quicksort (the testing data included several different implementations of Heapsort algorithms but only one Quickselect algorithm). This suggests that the numerical characteristics filter out non-relevant algorithms (algorithms that do not belong to the target set) well. Most of the FN cases were those algorithms belonging to the target set, but were filtered out and falsely recognized as Unknown due to different number of the numerical characteristics than those of the algorithms in the learning data. However, there were four FN cases in the results with the numerical characteristics within

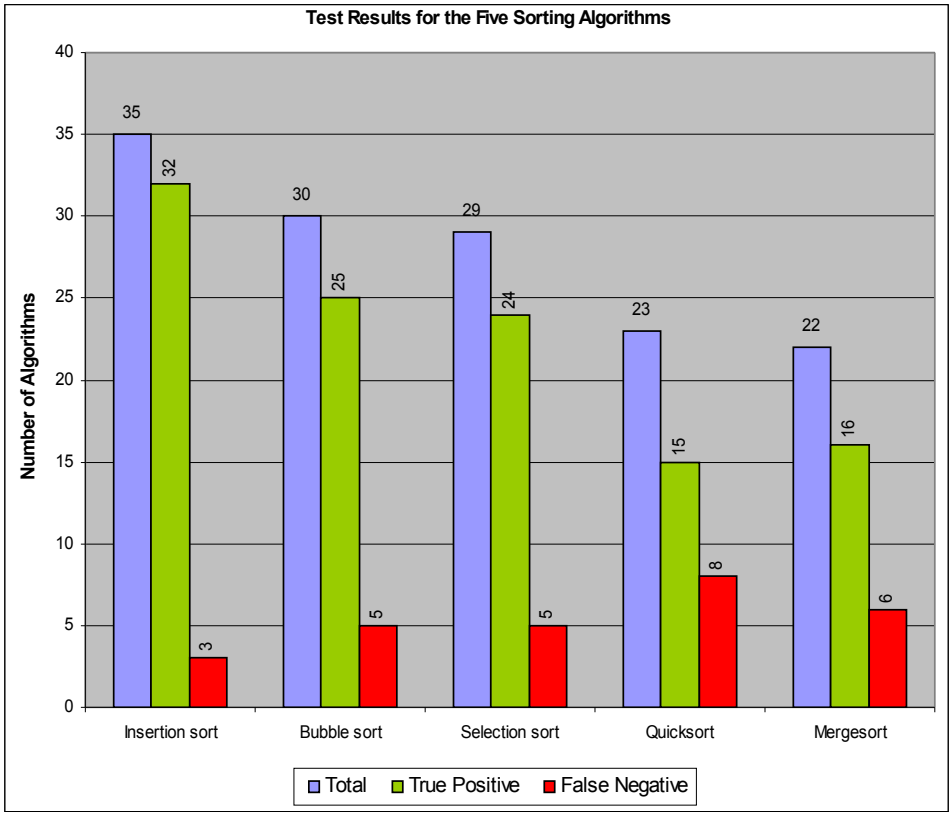

Figure 7.3: The test results (the TP and the FN cases only) for the five sorting algorithms (from [P2])

the permitted limit, but were falsely recognized because of failure of the tool to identify their characteristics (such as the temporary role): one Insertion sort was falsely recognized as a Bubble sort and three Quicksorts were falsely recognized as Mergesorts. Since the algorithms of these cases belong to the target set and are categorized as FN errors, they are not reported as FP cases.

Figure 7.3 shows the results for the five sorting algorithms, separated based on the TP and the FN cases. For each sorting algorithm, the first column in blue color shows the total number of the algorithm, the second column in green shows the number of the TP cases and the third column in red shows the number of the FN cases.

In order to further analyze the results of 7.3, we introduce three more metrics calculated based upon the TP, FP and FN cases. These metrics are presented as percentages and indicate the accuracy of the tests. They are called the *True Positive Rate (TPR)* (equivalently, sensitivity or recall), the *False Negative Rate (FNR)* and the *precision*, and are defined as follows. TPR is the proportion of the positive case algorithms that are correctly recognized, $TPR = TP/(TP + FN)$. FNR is the proportion of the positive case algorithms that are in-

Class	Total	TP	FN	FP	TPR	FNR	Precision
Insertion sort	35	32	3	0	91.4%	8.6%	100%
Bubble sort	30	25	5	0	83.3%	16.7	100%
Selection sort	29	24	5	0	82.8%	17.2%	100%
Quicksort	23	15	8	1	65.2%	34.8%	93.7%
Mergesort	22	16	6	1	72.7%	27.3%	94.1%

Table 7.2: The value of the TP, FN and FP cases, as well as the value of the True Positive Rate (TPR), False Negative Rate (FNR) and precision for each sorting algorithm in the MDT-experiment

correctly recognized as negative $FNR = FN/(TP + FN)$. The precision is the proportion of the actual positive case algorithms to all algorithms recognized as positive, $precision = TP/(TP+ FP)$. The value of these three metrics for the five sorting algorithms, along with the number of the TP, FP and FN cases needed for calculating them are shown in Table 7.2.

Large value of TPR indicates that there are few positive case algorithms that are falsely recognized as negative. As shown in Table 7.2, the value of TPR for the non-recursive sorting algorithms are larger than those for the recursive sorting algorithms. This implies that the Quicksort algorithms and Mergesort algorithms are misclassified more often than the non-recursive sorting algorithms. Moreover, as can be seen from Table 7.2, the larger value of TPR means the smaller value of FNR. Consequently, the value of FNR for the non-recursive sorting algorithms are smaller than those of the recursive sorting algorithms. The value of the precision for the non-recursive sorting algorithms is 100%, indicating that there is no single algorithm not belonging to the target set that is falsely recognized as Insertion, Bubble or Selection sort algorithm. But, as discussed earlier, one algorithm was falsely recognized as a Quicksort (i.e. a Quickselect algorithm) and one as a Mergesort (a Heapsort algorithm).

The fact that Insertion, Bubble and Selection sort algorithms are generally recognized better than Quicksort and Mergesort algorithms can be explained as follows. The more complex algorithms are the more their implementations are likely to vary. Quicksort and Mergesort are more complex than the other three non-recursive sorting algorithms. As the result, they can be implemented in more different ways using different strategies in lower level details. For example, as discussed in Chapter 2, there are several different strategies to carry out pivot selection and partitioning in Quicksort. Existence of different options in implementing algorithms results in a higher possibility for implementations of those algorithms to differ from each other, especially with regard to their numerical characteristics. For example, as will be discussed in the following, all those 27% FN cases for the Mergesort algorithms are due to the difference between the algorithms in their numerical characteristics. This will pose a challenge to us in future work when we further develop our method to

cover other fields of algorithms. We will have to come up with a more sophisticated method that performs better with more complex algorithms.

Table 7.2 also illustrates that in the case of recursive algorithms, the Mergesort algorithms are recognized better in comparison with the Quicksort algorithms (73% vs. 65%). This can be explained as follows. The reason for the failure in recognizing the Mergesort algorithms is that the number of the numerical characteristics of these FN cases (6 Mergesort algorithms) has been out of permitted limit. The same reason applies for the FN cases of the Quicksort algorithms, but in addition, there is another reason beyond the FN cases in this case, namely, failure of the role detector in recognizing the temporary role. Five out of the eight FN cases in the Quicksort algorithms occurred because of the difference between the number of the numerical characteristics of these Quicksort algorithms and those in the learning data, and the rest three of them occurred because of the absence of the temporary role in the algorithms or failure in recognizing it. As illustrated in the decision tree in Figure 7.1, when the role detector fails to recognize the temporary role, a Quicksort will not be recognized correctly. On the other hand, after a bit of tuning, the role detector performed very well in the case of the Selection sorts. The role detector detected all most-wanted holder roles correctly, resulting in a fair percent of correct recognition of the Selection sorts, that is, 83% (see Figure 7.1 and Table 7.2). All the FN cases of the Selection sorts (17%) occurred due to the difference between the numbers of the numerical characteristics.

Note that the FP cases reported for Quicksort and Mergesort in Table 7.2 could also be assigned to those algorithms not belonging to the target set that has been falsely recognized as Quicksort and Mergesort, that is, to the corresponding Quickselect and Heapsort algorithms, respectively. However, since these Quickselect and Heapsort algorithms belong to the Other algorithms, and listing all the algorithms that are members of the Other algorithms in the table is not practical, the FP case are discussed in connection with Quicksort and Mergesort algorithms which are members of the target set algorithms shown in Table 7.2.

By manually verifying those FN cases that have occurred because of the different numbers of their numerical characteristics, and by assigning the right type to them in the database, the number of the true positive cases will increase and the number of the false negative cases will decrease dramatically. This will extend the knowledge base of the Analyzer so that in the future, it will recognize the same FN cases correctly. This will result in a substantial improvement in the performance of the Analyzer. However, as we will discuss in Chapter 8, this could potentially cause also the FP cases to increase.

Class	Recursive	Tail_recursive	In-place	MWH	TEMP	OIID	IITO
Insertion	0.0	0.0	100.0	0.0	100.0	100.0	92.3
Selection	0.0	0.0	100.0	100.0	100.0	2.3	65.1
Bubble	0.0	0.0	100.0	0.0	100.0	16.6	0.0
Quick	100.0	100.0	100.0	0.0	64.1	N/A	N/A
Merge	100.0	0.0	48.5	0.0	12.1	N/A	N/A

Table 7.3: The percentage of the distribution of the descriptive characteristics of the five sorting algorithms in the data set (see Table 6.2 for the explanation of the abbreviations. For Quicksort and Mergesort algorithms, OIID and IITO are not applicable) (from [P3])

7.2 C4.5 decision tree experiment

In this section, we first describe the data set used in the CDT-experiment, following by the description of the decision tree constructed by the C4.5 algorithm and its empirical evaluation using leave-one-out cross-validation technique. The goal of conducting this experiment is to automatize the construction of the decision tree used in recognizing the five types of sorting algorithms and to see how different the steps in the resulted automatic decision tree are compared with the manually constructed decision tree illustrated in Figure 7.1. In other word, what characteristics the automatic decision tree will use as the tests in its nodes and how it distinguishes between the Insertion sort, Bubble sort, Selection sort, Quicksort and Mergesort algorithms.

7.2.1 Data set

For the CDT-experiment, we collected a total of 209 sorting algorithms of the aforementioned five types. The process of data collection and preparation and the sources where the data was collected were similar to those described in Subsection 7.1.1. The number and the percentage of each type of the collected sorting algorithm in the data set are as follows: Insertion sort 52 (25%), Bubble sort 42 (20%), Selection sort 43 (20%), Quicksort 39 (19%) and Mergesort 33 (16%).

Because the main purpose of this experiment is to see how the C4.5 algorithm distinguishes between the sorting algorithms (i.e., what characteristics the C4.5 algorithm uses to construct the decision tree), the Other algorithms was left out from the data set, and only the five aforementioned sorting algorithms were used. After making sure that the algorithms were correct and of the claimed type, they were run by the Analyzer and the resulted characteristic vectors were stored in the database. The numerical characteristics of these algorithms are shown in Chapter 6 in the second row of Table 6.3 and the descriptive characteristics are shown in Table 7.3.

In order to use the descriptive characteristics as inputs to the C4.5 algorithms, we con-

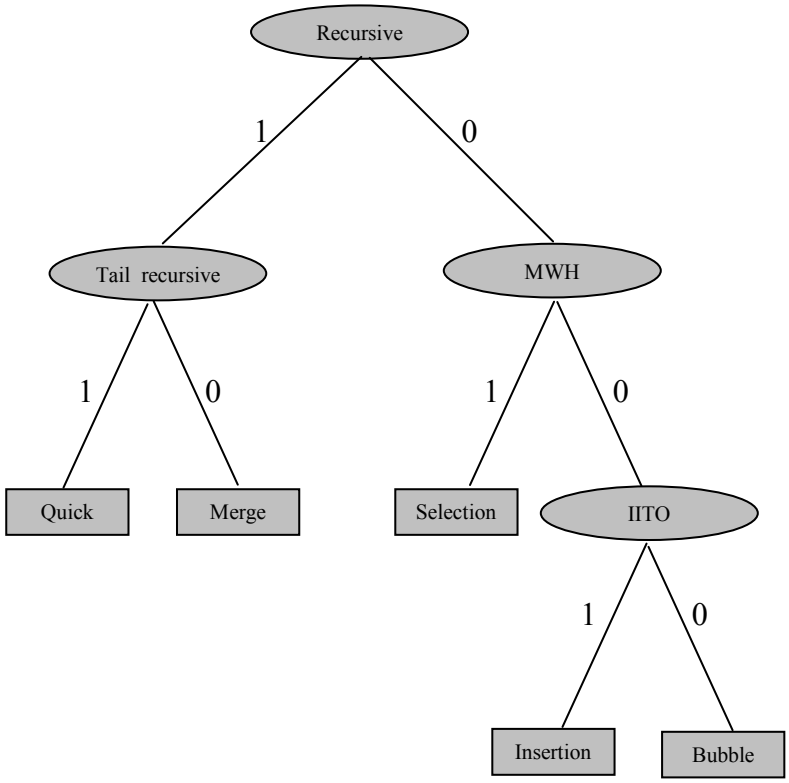

Figure 7.4: Decision tree built by the C4.5 algorithm for classifying sorting algorithms in the CDT-experience (See Table 6.1 for explanation of the abbreviations) (from [P3])

verted them into binary values, 1 indicating the presence of the corresponding characteristic and 0 indicating its absence. Table 7.3 shows how good classifiers these descriptive characteristics are, specially *Recursive*, *Tail_recursive* and *Most-Wanted Holder role (MWH)*. The characteristic Recursive can perfectly discriminate all the instances of Insertion sort, Selection sort and Bubble sort from Quicksort and Mergesort in the data set. Tail recursion, on the other hand completely distinguishes the Quicksorts from the Mergesorts. The existence of MWH in the algorithms is an excellent classifier as well, and can differentiate between Selection sort algorithms and the four other types of the sorting algorithms. As we will see, the decision tree that the C4.5 algorithm builds makes use of these exact characteristics. As illustrated in Table 7.3, the characteristics OIID and IITO (see Table 6.2 for the description of these characteristics) are not counted for the Quicksort and Mergesort algorithms. This is because either the implementations of these algorithms do not have two nested loops, or these characteristics are not good classifiers in these cases. However, as we will explain later, IITO is an important classifier for separating the Insertion sorts from the Bubble sorts.

7.2.2 Decision tree constructed by the C4.5 algorithm

The decision tree constructed by the C4.5 algorithm[2] is illustrated in Figure 7.4. The internal nodes are depicted as ellipses and include the tests based on which the splits are performed. There are four internal nodes (including the root), and thus, the value of four characteristics are used for the tests. The tree includes five leaves, which correspond to the number of the classes, that is, the types of the sorting algorithms. Each leaf is labeled with the associated type. The arcs in Figure 7.4 are labeled with either 0 or 1 from each internal node to its children. These values indicate the outcome of the test performed in each corresponding internal node. We do not go through the internal nodes and the test performed therein, as the corresponding nodes and tests were explained in connection with the manual decision tree presented in Figure 7.1.

The C4.5 algorithm tries to minimize the depth of the tree and classify the instances using the minimum number of characteristics which give the best result. Hence, from all the characteristics, only four are used in the decision tree. As the task includes the classification of the five sorting algorithms, the decision tree is not so complicated and thus, many characteristics remain unused. In a more complex decision tree, more characteristics are expected to be used in the process. Moreover, although four characteristics are enough to build the decision tree, different values of the characteristics result in different characteristics to be used and different tests to be performed. In order to test this and experience how the C4.5 algorithm reacts to the different values of the used characteristics, and how different values of the used characteristics affect the resulted decision tree, we made some very minor changes to the values of some characteristics and ran the C4.5 algorithm with these changed values. It was interesting to see that these minor changes in the value of a single characteristic resulted in a very different tree. As a concrete example, we performed a test where we changed the value of the characteristic Recursive to be 0 for one single Mergesort. The resulted tree used MWH as the test attribute in the root and *number of operands* as a test in an internal node, among the other differences.

By analyzing and comparing the values of the numerical and descriptive characteristics presented in Tables 6.3 (second row) and 7.3 respectively, it becomes clear why the four aforementioned characteristics are used as the classifiers in the tree. The numerical characteristics of the algorithms, although clearly different, are not the most useful characteristics in discriminating between the algorithms, when compared with the four used descriptive characteristics. Similarly, from the descriptive characteristics, In-place, TEMP and OIID have less discriminating values than those used in the decision tree.

It should be noted that the decision tree of Figure 7.4 only demonstrates how the five sorting algorithms can be recognized and distinguished from each other. The original pur-

[2]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/`

pose of constructing the decision tree and conducting the CDT-experiment was to automate the construction of the decision tree of Figure 7.1 and thus enhance the method. The automated tree is not intended to cover and be capable of classifying other fields of algorithms. When we extend our method to cover other algorithms in future work, the tree will have a mechanism to distinguish, as an example, between tail-recursive algorithms (as opposed to the decision tree of Figure 7.4 which classifies any given tail-recursive program as a Quicksort). In a more complex decision tree built to classify a more extensive field of algorithms, the numerical characteristics, for example, can be used to distinguish between different types of algorithms that share other common characteristics (e.g., characteristic such as being tail-recursive).

7.2.3 Evaluation of the decision tree

In this subsection we discuss the empirical evaluation of the performance of the decision tree and present the results.

There exist various techniques to evaluate the performance of decision trees. *Cross-validation* is a widely used technique, where the data set is divided into N subsets, which include both the training set and the test set. N different model of decision tree is constructed. Every time the decision tree is constructed, $N-1$ subsets are used as training set and one subset is used as test set to estimate the accuracy of the constructed tree. Thus, all of the subsets are used as the test set, and each of them exactly once. Cross-validation has two important advantages. First, it makes the best use of the available data (compare with *holdout* technique presented in the MDT-experiment, where the data set is divided into the mutually independent training and test set). Second, since all subsets take part in both training and test set, instances of data set are distributed uniformly to training and test sets. This eliminates the risk of getting a poor accuracy value for the tree just because of the unseen instances of the test set happen to vary largely from the instances of the training set [44].

Most commonly used value for N in cross-validation evaluation is 10. *Leave-one-out cross-validation*, that is the evaluation method which we used, is a special case of cross-validation, where N is equal to the number of instances in the data set. This makes even better use of data set when the available data is not large. Clearly, the disadvantage of leave-one-out cross-validation is that it can be computationally expensive for large data sets. As described, our data set is relatively small and therefore, we decided to use this technique to estimate the performance of the decision tree. With 209 algorithms of our data set, we get the training-test subsets created 209 times.

The accuracy of the decision tree evaluated by leave-one-out cross-validation technique is 97.1%. In other words, from the 209 algorithms of the data set, a total of 203 are classified correctly, and six of them (2.9%) are misclassified. Table 7.4 shows the overall results. The first and the second columns in the table show the type and the total number of the

Class	Total	TP	TN	FP	FN	TPR	TNR	FPR	FNR
Insertion sort	52	48	0	0	4	92.3%	0%	0%	7.7%
Bubble sort	42	40	0	0	2	95.2%	0%	0%	4.8%
Selection sort	43	43	0	0	0	100%	0%	0%	0%
Quicksort	39	39	0	0	0	100%	0%	0%	0%
Mergesort	33	33	0	0	0	100%	0%	0%	0%

Table 7.4: The value of the metrics indicating the accuracy of the decision tree

algorithms, respectively. The next four columns show the values of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) cases. TP cases are the number of correctly classified algorithms of each class, and FN cases show the number of misclassified algorithms of each class. Note that, as can be seen in the table, since all the algorithms of the data set used in the experiment are a member of the target set (i.e., there are no Other algorithms in the data set as were in the MDT-experiment, and all the algorithms belong to a class of the five aforementioned sorting algorithms), TN cases do not occur in the experience. In other words, there are no non-sorting algorithms within the data set that could be correctly recognized as such. For the same reason, FP cases do not occur in the experiment neither, that is, there are no non-sorting algorithms within the data set that could be falsely recognized as a sorting algorithm of the target set.

The four last columns show the value of True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR) and False Negative Rate (FNR) [53]. TPR and FNR were defined before in Subsection 7.1.3: TPR is the proportion of the positive case algorithms that are correctly captured by the decision tree (*TPR = TP/(TP + FN)*) and FNR is the proportion of the positive case algorithms that are incorrectly labeled as a negative case (*FNR = FN/(TP + FN)*). Correspondingly, TNR is the proportion of the negative case algorithms that are correctly predicted (*TNR = TN/(TN + FP)*), and FPR is the proportion of the negative case algorithms that are incorrectly labeled as a positive case (*FPR = FP/(TN + FP)*).

As can be seen from Table 7.4, from the six misclassified algorithms, four pertains to Insertion sort algorithms and two to Bubble sort algorithms. We use the *confusion matrix* to discuss the misclassified algorithms in more detail. The confusion matrix is a $N \times N$ matrix, where each instance I_{ij} indicates the instance that belongs to class I_i, but is classified as class I_j [53]. The instances located on the diagonal are classified correctly. We have five classes of algorithms, thus 5×5 confusion matrix, as shown in Table 7.5 (were the first column is the class of the algorithms). As the confusion matrix shows, all the four misclassified Insertion sorts are labeled as Bubble sort and both misclassified Bubble sorts are predicted as Insertion sort. All the instances of the three other classes, Quicksort, Mergesort and Selection sort are classified correctly.

Class	Insertion sort	Bubble sort	Selection sort	Quicksort	Mergesort
Insertion sort	48	4	0	0	0
Bubble sort	2	40	0	0	0
Selection sort	0	0	43	0	0
Quicksort	0	0	0	39	0
Mergesort	0	0	0	0	33

Table 7.5: The confusion matrix of evaluating the decision tree by leave-one-out cross-validation technique

The decision tree constructed by the C4.5 algorithm is much more optimal, simple and comprehensible than the manually constructed decision tree illustrated by Figure 7.1. Moreover, although building a manual decision tree for classifying the five sorting algorithms with a reasonable performance is a feasible task (as shown in the MDT-experiment), automating the process of building the decision tree for classifying a more comprehensive field of algorithms is inevitable. We will discuss the manual and automatic decision trees, their differences and the issues related to their accuracy in more detail in the next chapter.

Chapter 8

Discussion, Conclusion and Future Work

In this chapter, we first discuss some issues related to the algorithm recognition process. Then we present some conclusions about the work, and finally, we explain some direction for future research.

8.1 Discussion

We have presented a method for recognizing basic algorithms. The method analyzes algorithms with regard to language constructs, software metrics and roles of variables, computes a set of numerical and descriptive characteristics and converts the algorithms into vectors of characteristics. These vectors are used to distinguish between different algorithms.

To limit the scope of the work, we tested the method on sorting algorithms. We conducted two separate experiments to investigate how the method performs in recognizing Insertion sort, Bubble sort, Selection sort, Quicksort and Mergesort algorithms. In the first experiment, based on the analysis of the algorithms in the learning data, a manual decision tree was constructed to guide the process of recognition. In the second experiment, the decision tree was constructed by the C4.5 algorithm.

8.1.1 The manual and automatic decision trees

The numerical characteristics work as filters in the manual decision tree and have a significant role in the process of recognition. They prevent the algorithms that are not part of the target algorithms from being further processed and label them as Unknown (see Figure 7.1). If the values of the numerical characteristics are too tight (i.e., if the minimum and maximum limits of the values of the numerical characteristics are too close to each other), the number of the false negative cases, that is, the number of the positive case algorithms

that belong to the target set but are falsely recognized as negative cases, will increase. On the other hand, if the minimum and maximum limits of the values of the numerical characteristics are too far from each others, the number of the false positive cases, that is, the number of the negative case algorithms that do not belong to the target set but are falsely recognized as positive cases, will increase (note that if a negative case algorithm passes the filters of the numerical characteristics, it ends up to be labeled as a false positive case). Therefore, like in all machine learning techniques, the choice of the training data is crucial and affects on the results in terms of the false negative and false positive cases.

It should be noted that as the purpose of constructing the automatic decision tree has been to demonstrate the feasibility of machine learning methods in constructing the decision tree and to show how the five analyzed sorting algorithms can be distinguished from each others using the automatic decision tree constructed by the C4.5 algorithm, the automatic decision tree illustrated by Figure 7.4 has only a mechanism to recognize the five analyzed sorting algorithms, and shows a more optimal, logical and understandable way to do this (in comparison with the manual decision tree of Figure 7.1). The automatic decision tree minimizes the depth of the tree. It distinguishes between the Quicksort and Mergesort algorithms only based on the "Tail_recursive" characteristic, and removes the internal nodes that include "Temporary role" and "In-place" tests, as redundant. Likewise, it removes the node containing the "OIID" test as unnecessary and discriminates between the Insertion sort and Bubble sort algorithms only based on the characteristic "IITO" (see Figures 7.1 and 7.4). Thus, the resulted tree is more logical and understandable.

The accuracy of the manual and automatic decision trees cannot be directly compared since the data sets used to evaluate their accuracy are different. The testing data of the manual decision tree experiment includes the Other algorithms in addition to the five sorting algorithms, whereas the data set of the automatic decision tree experiment includes only the five sorting algorithms (and the tree has no filtering mechanism to deal with other algorithms). Moreover, the methods used for evaluating the accuracy of the trees are different (Holdout method for the manual decision tree with 70 sorting algorithms as the learning data and 217 sorting and other algorithms as the testing data, as opposed to leave-one-out cross-validation for the automatic decision tree with a data set consisting of 209 sorting algorithms). To provide a more similar conditions that allow us to do this comparison, we removed the Other algorithms from the testing data of the manual decision tree and evaluated the accuracy of the tree using only the five sorting algorithms, as is the case in the automatic decision tree experiment. Note that as there are 78 Other algorithms in the testing data, removing them will result in a testing data consisting of 139 sorting algorithms. Thus, the number of algorithms that are used to evaluate the accuracy of the two decision trees are still not similar, but removing the Other algorithms provide more reasonable foundation to do the comparison. Using 139 sorting algorithms of the aforementioned five types as the

testing data, the accuracy of the manual decision tree is 79% (i.e., 110 true positive cases). Removing the Other algorithms eliminates all the 76 true negative cases, which constituted 35% of the overall results, and the two false positive cases, which was 1% of the overall results (see Figure 7.2). The proportion of false negative cases (the 29 false negative cases) rises from 13% to 21% (see Figure 7.2). These data provide a more reasonable basis to compare the accuracy of the manual decision tree (79%) with the accuracy of the automatic decision tree (i.e., 97.1%) and conclude that using machine learning methods results in a decision tree that performs better.

8.1.2 Application of the method/Analyzer

The main application of the method in its present state is in computer science education. In algorithms and programming courses, students are required to write several programs in order to pass the course. These programs usually implement some basic algorithms. For example, when teaching sorting algorithms, a teacher may ask students to implement Insertion sort, Quicksort, Mergesort, etc. Since the existing automatic assessment tools are not capable of easily and reliably recognizing the algorithms that perform the same task (i.e., sorting in our example), these works must be checked manually. However, this is a time-consuming task especially in large courses. The Analyzer can be used to automatically assess these kind of works. Although due to the statistical nature of the method, its accuracy is not 100%, the results reported in Chapter 7 show that it can help marking students' programs.

Note that as all the algorithms are stored in the database, a teacher can always check the failed algorithms (labeled as Unknown in the database) to ensure that they are indeed not the type which was required from students. For example, if the submitted program is expected to be a Quicksort, the teacher would inspect the negative cases to either confirm that the program is indeed not a Quicksort algorithm (a true negative case), or to correct the type of the algorithm in the database if it is (a false negative case), so that the Analyzer would recognize the similar algorithm correctly in the future. In this context, false positive cases – although they occur less than false negative cases – are more difficult to track and thus more serious problems. If a wrong algorithm is classified as Quicksort (a false positive case), this will not be discovered since the teacher accepts the positive cases and does not inspect them (see [P2] for more detailed information). It should also be noted that if a legitimate algorithm is falsely labeled as Unknown because of its numerical characteristics not being withing the permitted limit (i.e., a false negative case), correcting the type of the algorithm in the database should be done with care. The reason is that, as discussed above, doing so would mean that a wider range of algorithms would pass the filter of the numerical characteristics in the future, and this would potentially increase the number of false positive cases.

8.1.3 Roles of variables (RoV)

The importance of beacon in Program Comprehension (PC) models was discussed in Chapter 4. An initial discussion on how RoV can be seen as beacons was also presented in that chapter. This beacon-like notion of RoV is apparent in decision trees illustrated in Figures 7.1 and 7.4. As an example, when a programmer reads code that implements a non-recursive sorting algorithm, appearance of the most-wanted holder role in the code can help him/her to accept his/her hypothesis that the algorithm is a Selection sort (see Figures 7.1 and 7.4).

A multidisciplinary study linking PC research field to computing education research field can help to understand the beacon-like notion of RoV. As an example of PC models, Soloway and Ehrlich's model suggest that to facilitate comprehension of programs, variables should be used in a stereotypical manner and with consideration of the good rules of programming discourse. For example, "variable names should reflect function" [50]. RoV, on the other hand, "are a classification of stereotypical behaviours of variables that occur repeatedly in programs" [46]. Assuming that *function* and *behavior* from the two quotes above imply the same concept, beacon-like notion of RoV become evident (indeed, as was discussed in Chapter 4, Soloway and Ehrlich use the term *critical line* instead of beacon, but we argue that these terms has the same meaning in the context of our discussion and thus can be used interchangeably. Compare the notion of critical lines with beacons in, for example, [7]). While PC models are focused on perspectives such as how experts understand program code, what are the differences between novices and experts in PC process, what strategies are most effective in PC, etc., and aim at finding models to help programmers and maintainers in software engineering activities, RoV are introduced in the context of computer science education and are considered as a concept to help novices to understand the notion of variable and to learn programming. In [49], an attempt has been made to bring PC research field closer to computer science education field. In that work, based on analyzing several PC models, a number of suggestions is made to computer science educators as to what kind of learning tasks should be given to student, what are learning obstacles and how they should be dealt with, what teaching methods should be used, etc., in order to help students to learn programming more effectively and comprehend programs better. Furthermore, building domain knowledge and extensive knowledge base is also suggested and it is argued that learning tasks and sequences should be so that they support building this knowledge. In this regard, viewing RoV as beacons and introducing them to students from this perspective supports the process of building knowledge base and enhance the ability of students to understand program code. 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., [10, 47], as well as [51] for information from a teachers' point of view). An attempt in the reverse direc-

tion can be made by using RoV as beacons in PC field: bringing a concept from computer science education field into PC research field. Programmers and maintainers, both novices and experts, can utilize RoV in PC process in software engineering activities. Since RoV are tacit knowledge of experts [45], it should be easy for experts to adopt the concept of RoV even if roles are not explicitly introduced to them yet. For novices, understanding RoV and identifying them so that roles can improve PC process may require some effort in the beginning. However, learning the concepts and elements of PC models, such as plans, beacons, code navigating strategy, etc., is also a part of the process of becoming an expert, just like learning RoV.

Based on the experiments presented in Chapter 7, RoV are also very useful in algorithm recognition (AR). However, the value of RoV in AR, as part of automatic program comprehension, remains to be further investigated when other fields of algorithms are taken into the process. There are two concerns for applicability of RoV in automatic program comprehension field. The first is, as the different types and fields of algorithms are taken into the process, how distinguishing RoV would be if the same role appears in many different algorithms? The second concern is the fact that RoV are cognitive concepts [5, 17]. How accurately a tool can detect roles if the target algorithm set includes several different roles, and how these detected roles agree with those assigned by a human? See [P2] for a more detailed discussion on these issues and some suggestions on how to deal with them.

8.1.4 Limitations of the method

The performance of the method has been evaluated by sorting algorithms. To obtain a more comprehensive results, we need to further develop the method to cover other fields of algorithms and evaluate its performance using a wider range of the basic algorithms. We discuss this issue in the following when we give some direction for future work.

In its current state, the method is not able to identify the code that is not related to the implementation of an algorithm. All program code is dealt with as algorithmic code, and thus the method should be further developed to extract the algorithmic code from the application-specific code. One way to address this problem is to define the supported algorithms as schemas, specify the relationship between these schemas and store them in the database. Correspondingly, the target algorithms can be converted into schemas and relationships, and be compared to those from the database in order to identify the algorithmic code from the source code. This approach is used in knowledge-based PC techniques discussed in Chapter 3. When the algorithmic code is located in the source code, it can be highlighted using software visualization techniques to help the user observe it.

The method assumes that target algorithms work correctly. Assessing the correctness of an algorithm requires it to be executed and is out of the scope of our work. There exist number of different tools that are capable of assessing the correctness of programs. They

can be integrated into the Analyzer so that after the target algorithm is recognized by the Analyzer, its correctness is assessed by an assessment tool. Alternatively, if the output of the automatic assessment tool can facilitate the recognizing process, the assessment of the correctness of an algorithm can be carried out before its recognition.

8.2 Conclusion

In this section, we present a few conclusions which are drawn based upon the experiments presented in Chapter 7 and are connected with the research questions stated in Chapter 1. See the original articles [P1, P2, P3] for more conclusions.

8.2.1 Feasibility of the method

The results of the experiments presented in Chapter 7 show the feasibility of the method and good performance of the decision trees. In the first experiment, 86% of the algorithms of the testing data were recognized correctly (the true positive and true negative cases), and as was discussed above, using a testing data that consist of only the five aforementioned types of sorting algorithms, the percent of the correctly recognized algorithms was 79%. Evaluated by leave-one-out cross-validation technique, the decision tree of the second experiment was able to correctly recognize 97.1% of the algorithms of the data set correctly. These results show both the applicability of RoV (the first research question), and the algorithm characteristics (the second research question). RoV distinctively describe some sorting algorithms used in the experiments. Other characteristics also help to differentiate between those algorithms that cannot be distinguished based on RoV. Indeed, these conclusion are drawn based upon the experiments conducted on sorting algorithms. In order to show the performance of the method (including both applicability of RoV and the other characteristics) on other types of algorithms, corresponding experiments must be conducted.

8.2.2 Applicability of machine learning techniques

Converting algorithms to characteristics vectors allows us to utilize machine learning techniques in our method. Using the C4.5 decision tree classifier in our second experiment presented in Chapter 7 and the performance of the constructed decision tree shown in the results therein (see also [P3]) illustrate the applicability of the C4.5 algorithm as a machine learning method. This provides the answer to our third research question presented in Chapter 1. We believe that other machine learning techniques can also be applied in our method as we extend the method to cover other fields of algorithms. The applicability of other machine learning techniques, however, needs to be proved by the appropriate empirical experiments.

8.3 Future work

The method we have presented needs to be further developed to cover other fields of algorithms and to be able to deal with the other limitations. We elaborate on these research directions in the following.

8.3.1 Covering other algorithms

The next step to further develop the method is to extend it to cover also other algorithms than sorting algorithms. We need to analyze other fields of basic algorithms (searching algorithms, tree and graph algorithms, etc.) and evaluate empirically, whether the presented numerical and descriptive characteristics are sufficient for them to be distinguished and identified. We may need to look for other distinguishing factors and/or drop some of the current characteristics as redundant. Furthermore, we need to determine what do we consider to be an algorithmic code that belongs to the core algorithm. As an example, Prim's algorithm for computing minimum-cost spanning tree needs an instance of a graph as input. Should the graph be taken into account when calculating the characteristics for Prim's algorithm? Moreover, we need to come up with a mechanism to deal with the fact that the more complex algorithms get, the more they may vary in implementation. Using the same example, in Prim's algorithm, the next closest vertex can be stored in an array, or alternatively a priority queue can be used for that.

Particularly, we need to analyze the variables used in other target algorithms and investigate what role they play. As the roles are cognitive concepts, more than one person should take part in this activity to make sure that the assigned roles are correct. We probably need to further tune the automatic role detector that we used in order to detect other roles with a reasonable accuracy. Moreover, other data mining and machine learning techniques should be used and their applicability and accuracy should be empirically evaluated and compared. Specifically, we will use K-mean clustering to divide the algorithms based on their characteristics at a high level, and will further examine the clusters in more detail to identify the algorithms. Self-organizing map also seems to be a good alternative method to classify and recognize algorithms.

As discussed in Subsection 8.1.4, another direction for future work is how to deal with application data and identify algorithmic code from source code. See the discussion there for more information about our plans to tackle this problem.

Bibliography

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

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

[3] 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.

[4] 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*, pages 368–377. IEEE Computer Society Washington, DC, USA, 1998.

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

[6] 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.

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

[8] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156, 2002.

[9] 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, Pittsburgh, Pennsylvania, USA, 5–7 May, pages 66–73. IEEE Computer Society Washington, DC, USA, 1999.

[10] Pauli Byckling and Jorma Sajaniemi. Roles of variables and programming skills improvement. *SIGCSE Bulletin*, 38(1):413–417, 2006.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[12] James H. Cross, Dean Hendrix, Karl S. Mathias, and Larry A. Barowski. Software visualization and measurement in software engineering education: An experience report. In *Frontiers in Education Conference, 1999*, 1999.

[13] 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.

[14] Ali Erdem, W. Johnson, and Stacy Marsella. Task oriented software understanding. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering, Honolulu, Hawaii, 13–16 October*, pages 230–239. IEEE Computer Society Washington, DC, USA, 1998.

[15] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. Mental representations of programs by novices and experts. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems (Amsterdam, The Netherlands, 1993)*, pages 74–79, 1993.

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

[17] 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.

[18] 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.

[19] Sam Grier. A tool that detects plagiarism in pascal pograms. In *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, pages 15–20. ACM, 1981.

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

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

[22] David Harel and Yishai Feldman. *Algorithmics The Spirit of Computing*. Addison-Wesley, 2004.

[23] 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.

[24] 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.

[25] J.K. Joiner, W.T. Tsai, X.P. Chen, S. Subramanian, J. Sun, and H. Gandamaneni. Data-centered program understanding. In *Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia, Canada, 19–23 September*, pages 272–281. IEEE Computer Society Washington, DC, USA, 1994.

[26] Edward L. Jones. Metrics based plagiarism monitoring. In *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 253–261. ACM, 2001.

[27] 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.

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

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

[30] 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.

[31] Stanley Letovsky. Cognitive processes in program comprehension. In *Papers presented at the first workshop on Empirical studies of programmers*, pages 58–79. Ablex Publishing Company, 1986.

[32] 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.

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

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

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

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

[37] Thomas Naps, Stephen Cooper, Boris Koldehofe, Charles Leska, Guido Rößling, Wanda Dann, Ari Korhonen, Lauri Malmi, Jarmo Rantakokko, Rockford J. Ross, Jay Anderson, Rudolf Fleischer, Marja Kuittinen, and Myles McNally. Evaluating the educational impact of visualization. *SIGCSE Bulletin*, 35(4):124–136, December 2003.

[38] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

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

[40] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341, 1987.

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

[42] 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.

[43] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

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

[45] 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.

[46] 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.

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

[48] Jorma Sajaniemi, Marja Kuittinen, and Taina Tikansalo. A study of the development of students' visualizations of program state during an elementary object-oriented programming course. In *Proceedings of the third international workshop on Computing education research (Atlanta, Georgia, USA, 2007)*, pages 1–16. ACM New York, NY, USA, 2007.

[49] Carsten Schulte, Teresa Busjahn, Tony Clear, James H. Paterson, and Ahmad Taherkhani. An introduction to program comprehension for computer science educators. *Working group reports on ITiCSE on Innovation and technology in computer science education - ITiCSE-WGR '10 (June 26–30, 2010, Bilkent, Ankara, Turkey, 2010)*, To appear.

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

[51] Juha Sorva, Ville Karavirta, and Ari Korhonen. Roles of variables in teaching. *Journal of Information Technology Education*, 6:407–423, 2007.

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

[53] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, USA, 2006.

[54] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[55] Anneliese von Mayrhauser and A. Marie Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, 1996.

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

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

[58] 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*, pages 130–134. ACM, 1996.

[59] Steven Woods and Qiang Yang. The program understanding problem: analysis and a heuristic approach. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 25–29 March*, pages 6–15. IEEE Computer Society, Washington, DC, USA, 1996.

[60] Steven Woods and Qiang Yang. Program understanding as constraint satisfaction: Representation and reasoning techniques. *Automated Software Engineering*, 5:147–181, 1998.