Ahmad Taherkhani

Static Program Analysis for Recognizing Sorting Algorithms

May 26th, 2008

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences

ABSTRACT OF THE MASTER'S THESIS

Author:	Ahmad Taherkhani	
Title of the thesis:	Static Program Analysis for Recognizing Sorting Algorithms	
Date:	May 26th, 2008	**Number of pages:** vii + 91
Department:	Department of Computer Science and Engineering	
Professorship:	Software Technology	**Professorship code:** T-106
Supervisor:	Professor Lauri Malmi	
Instructor:	Docent Ari Korhonen	

Automatic computer program analysis and code recognition is an interesting subject in computer science. The reason for this can be found mainly from software industry, and particularly from a certain phase of software life-cycle: *maintenance*.

By automatic computer program comprehension and code recognition we mean a system that could tell us what does the input program appears to be trying to do, what algorithms does it resemble and how closely, or what kind of structure and style does the program have.

If properly and comprehensively developed, such a system could help software developers gain a quick understanding of the software they are maintaining, and thus save them from reading the source code, which is a time-consuming task. A system of this kind could be well used in other phases of software projects like verification and validation tasks as well.

Another use of such a system is automatic assessment of exercises in computer science courses. There are a number of large size computer science-related courses at universities, which require many exercises to be submitted by students. The system could take in an exercise on particular subject, and tell the instructor, whether the exercise does what it is expected to do, and if not, how close is it to the desired functionality.

Previous works on automatic program analysis and code recognition are surveyed and different approaches to the problem are explained and evaluated. As a new technique, a static program analysis and code recognition based on the numerical and descriptive characteristics of algorithms is presented. *Roles of variables* used in algorithms plays an important role in the method. This work is limited to include only different well-known sorting algorithms, and further developing the system to cover other algorithms is left to the future research. Finally, other limitations of the work and some suggestions for future research are described.

Keywords: Program analysis, static program analysis, program recognition, sorting algorithms

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ
Informaatio- ja luonnontieteiden tiedekunta

Author:	Ahmad Taherkhani	
Title of the thesis:	Ohjelman staattinen analyysi lajittelualgoritmien tunnistamiseksi	
Päivämäärä:	26.5.2008	**Sivumäärä:** vii + 91
Laitos:	Tietotekniikan laitos	
Professuuri:	Ohjelmistotekniikka	**Koodi:** T-106
Työn valvoja:	Professori Lauri Malmi	
Työn ohjaaja:	TkT Ari Korhonen	

Automaattinen ohjelman analyysi ja ohjelmakoodin tunnistus on kiinnostava ongelma tietotekniikka-alalla. Syy tähän löytyy ohjelmistoteollisuudesta, ja varsinkin ohjelmiston elinkaaren yhdestä vaiheesta: *ylläpidosta*. Automaattisella ohjelman ymmärtämisellä ja ohjelmakoodin tunnistuksella tarkoitetaan järjestelmää, joka voisi kertoa käyttäjälle, mitä syötteenä saatu ohjelma näyttäisi tekevän, mitä algoritmia se muistuttaa ja kuinka paljon, tai millainen rakenne ja tyyli ohjelmalla on.

Tällainen järjestelmä voi - hyvin ja laajasti kehitettynä - auttaa ohjelmistokehittäjiä ymmärtämään ylläpidettävän järjestelmän toimintaa, ja säästämään heitä lukemasta ohjelmakoodia, joka on aikaa vievää toimintaa. Tällaisella järjestelmällä olisi käyttöä myös muissa ohjelmiston elinkaaren vaiheissa: järjestelmä voisi helpottaa ohjelmistokehittäjien työtä ohjelmiston verifiointi- ja validointitehtävissä huomattavasti.

Kyseisestä järjestelmästä olisi apua myös yliopistoissa ja korkeakouluissa tarjolla olevien tietotekniikan kurssien yhteydessä suoritettavien tehtävien automaattisessa tarkastuksessa. Tietyt tietotekniikan kurssit ovat yleensä laajoja, ja niiden suorittaminen vaatii opiskelijoilta monen tehtävän suorittamista. Järjestelmä voisi ottaa vastaan opiskelijan lähettämän tiettyyn aiheeseen liittyvän tehtävän, ja kertoa sitten opettajalle, onko tehtävä ratkaistu oikein, ja jos ei, niin kuinka lähellä oikeaa ratkaisua se on.

Tässä diplomityössä tutkitaan aikaisempia automaattiseen ohjelman analyysiin ja ohjelmakoodin tunnistukseen liittyviä töitä ja esitetään erilaisia lähestymistapoja ongelmaan. Uutena lähestymistapana esitetään mm. ohjelmakoodissa käytettyjen muuttujien lukumäärään ja rooliin pohjautuva staattinen ohjelman analyysi ja koodin tunnistus. Tämä työ rajataan koskemaan ainoastaan tunnettuja lajittelualgoritmeja, ja järjestelmän jatkokehitys ja sen toiminnan laajentaminen muiden algoritmien käsittelemiseksi jätetään tulevalle tutkimukselle. Lopuksi kerrotaan työn muista rajauksista ja esitetään muutama ehdotus tulevalle tutkimukselle.

Avainsanat: Ohjelman analyysi, staattinen ohjelman analyysi, ohjelman tunnistus, lajittelualgoritmit

Acknowledgements

This work was carried out at the Laboratory of Software Technology, Department of Computer Science and Engineering at Helsinki University of Technology.

I would like to thank my supervisor, Professor Lauri Malmi, for giving me the opportunity to do this work and for his great advices and highly respected guidance. I am sincerely thankful to him.

I also want to thank my instructor, D.Sc. Ari Korhonen, for his fresh and valuable ideas and suggestions, which were always of great help to find the way forward.

I am also grateful to Mikko-Jussi Laakso for his contribution to this work and for the help he provided on the VILLE system along with Teemu Rajala. I appreciate their assistance.

I would like, however, express my greatest gratitude to my family for standing by me during this work. My wife's passionate attitude toward science has always been the greatest support and encouragement. I dedicate this work to them.

Otaniemi, May 26th, 2008

Ahmad Taherkhani

Contents

Chapter 1

Introduction

Automatic computer program comprehension and code recognition has been an interesting subject for researchers and experts for a few decades. Many different approaches with more or less success have been presented as a result of these years-long researches. But what the program understanding problem really means and what aspects does it involve?

We can classify the problem of understanding a program roughly into the following three main categories.

Understanding functionality: Perhaps the most obvious and straightforward meaning of program understanding problem is, as the name suggests, simply understanding the meaning of a program, i.e., what the program does. This is, in fact, at the same time the most challenging and complex definition of the problem. In this sense, understanding a sorting algorithm, for example, simply means that we can say the program executes the sorting functionality, and its output would be a set of ordered digits.

Classifying algorithms: In this sense, understanding a program imply being able to classify algorithms. Therefore, the process of understanding an algorithm involves finding out what family of algorithms the understandable algorithm belongs to, or what kind of algorithm does it resemble.

Analysing structure and style: Program understanding could mean to examine the structure of the code to see, for example, how control structures are used. The coding style could also be investigated in this process. The objective of these analyses could be, for example, to monitor students' progress or get a rough idea about efficiency.

The primary reason for extensive research on program understanding problem is that a good program understanding tool could be beneficial for maintainers and developers working on software projects and teaching staff offering courses at universities, in many different ways, some of which are as follows.

Software verification and validation: Automatic program comprehension could be used in the software verification and validation process along with or ideally replacing other techniques like code review, inspection and walkthrough. Reading source code is a time-consuming and error-prone activity. An automatic code recognition system could be of great

help in a situation where an inspector is expected to review thousands of lines of source code in some short period. Moreover, an automatic code recognition system along with text recognition techniques, such as LSI (Latent Semantic Indexing) [46], could be a helpful and powerful tool to validate software systems and insure that they meet the specifications throughout the whole project life-cycle. Source code could be verified against all kind of available documents from requirements specifications to UML diagrams. Such a system could be very helpful also in software testing activities.

Software maintenance: Software maintenance is, as well-known, the most expensive phase in software life-cycle. This is particularly true if there is no adequate documentation about the system to be maintained, and source code is not commented properly. In many cases, maintainers are not the same people who have developed the system, which makes the situation even worse, because they do not know what it is all about. Legacy codes and different programming languages and programming paradigms increase the difficulty of the maintenance phase. To be able to maintain a system, maintainers have to understand the system first. When there is no proper documentation for a system, the only way to understand it is to read the source code, which is a slow process. As a result, understanding programs is the most time-consuming task in software maintenance [25]. Automatic code recognition can help the maintainers in their work and reduce the cost of maintenance phase significantly. Software maintenance has actually been pointed out in many researches as a primary motivation and as an activity, which is supposed to benefit most from automatic program comprehension tools [17, 25, 27, 39, 42, 64]. In addition to helping the maintainers to understand the code by telling them what the code does, automatic code recognition could provide the maintainers with information about what is wrong with the code in question and suggest some improvements [25, 33].

Automatic assessment: Computer science-related courses at universities are sometimes very large [45], and passing them successfully often requires submiting a noticeable number of homeworks and exercises. Evaluating all submitted exercises manually is a difficult and error-prone task and takes a lot of time. Automatic code recognition could carry out the assessment of exercises automatically, and thus help the teaching staff very much. For example, the course Data Structure and Algorithms lectured at Helsinki University of Technology is a course with hundreds of attendees. Students are required to submit a number of exercises on different subjects, for instance on sorting algorithms. Automatic code recognizer could take a submitted sorting algorithm in and give information to the instructor as an output about whether the input program does what it is meant to do and to what extent. If the automatic code recognition system fully confirms the exercise doing what it is expected to do, a complete score can be given to the student who submitted the exercise. If the system does not fully confirm the exercise as being doing what is expected, it could tell the instructor how the exercise differs from the right or standard answer. The exercise could be scored based on these differences. Automatic code recognition could also be used by novice students in their learning process, for example, by giving them information about whether their program works correctly or not. This kind of automatic code recognition system could use a knowledge base that consists of all algorithms it can recognize. As a result, the system can recognize different implementation of a particular algorithm if that type of algorithm is included in the knowledge base of the system. If the system does not recog-

nize an algorithm, either its knowledge base does not include the information about that algorithm or the algorithm is implemented somehow differently than what the system has knowledge about. In these situations, the instructor can carry out the recognition manually, and extend the knowledge base of the system to cover that algorithm, if necessary.

In addition to the few abovementioned examples, program understanding could be used in many other fields. Therefore, there is no question about the usefulness of such tools and their applicability in solving various problems. The question, instead, is how effective these tools can be and to what extent they can fit real life requirements? What about efficiency and accuracy concerns? Is it theoretically possible to solve the problem of automatically understanding a program?

These questions are discussed later on in this thesis. To distinguish between the process of understanding a program by computer and by human in this thesis, we use the term *understanding* for computer and the term *comprehension* for human.

Next, a brief argue about the complexity of the problem of understanding a program is presented.

1.1 Complexity Concerns

There are many problems in computer science known as undecidable problems, meaning that there is no provenly working algorithm that can solve them in finite time. The problem of deciding the equivalency of syntactical definitions of programming languages, which is also known as the equivalency problem of context-free grammars, is one of these problems. This problem is, as described in [26], proven undecidable by Bar-Hillel et al. [5]. The undecidability of this problem means that there exists no algorithm which is able to tell in a finite amount of time, whether two given syntactic rules describe the same language.

The problem of understanding a program can be considered equal to the problem of syntactical equivalence of languages in the sense that in syntactical equivalency the problem is to decide whether two given input set of syntax rules are equivalent, i.e., whether they define the same language. The problem of understanding a program, on the other hand, can be regarded to be a problem of deciding whether two given programs are equivalent, i.e., whether they do the same thing or solve the same problem. In order to be able to decide whether two given programs solve the same problem, one must first understand what those programs do. In other words, being able to say whether two given programs do the same thing can be considered equal to being able to say what those two programs do. Therefore, program understanding problem and syntactical equivalence problem seem to fall into the same category, resulting in program understanding problem being considered as undecidable problem, since syntactical equivalence problem is, as well known, an undecidable problem.

Program understanding problem can also be regarded undecidable considering the halting problem, which is proven undecidable by Turing [67], as likewise described in [26]: there is no proof whether a program R will ever terminate in a finite time when running on some

input X. Even though some instances of the undecidable problems might be solvable, they can not be generally considered as decidable in nature, since it has not been proven.

As a matter of fact, as Steven Woods and Qiang Yang notice, the previous researches on program understanding problem rarely present a formal complexity analysis of the problem [70]. Steven Woods and Qiang Yang prove the problem to be NP-hard, but their proof, however, concerns knowledge-based program understanding approaches rather than the problem itself. In knowledge-based program understanding approach, which is, as we will describe later, a specific technique suggested as a solution to the problem, the program understanding problem is converted into the problem of mapping the plans from the plan library to the source code. Therefore, Steven Woods and Qiang Yang prove the knowledge-based program understanding approaches to be intractable, and not the problem of program understanding itself.

Having pointed out the complexity issue about the problem of understanding a program, one can conclude that to solve the problem, it needs to be converted into a decidable problem, which can be solved in an efficient manner. As an example, in the same way as is done in knowledge-based program understanding approach or other techniques that we will discuss later in Chapter 3. In this work, we consider the problem of understanding a program from algorithm classification point of view, which can obviously be regarded as an easier problem than, e.g., understanding functionality of a program.

1.2 Thesis Outline

In this thesis, we develop a static program analysis and code recognition system for recognizing different sorting algorithms. Our approach is different from the previous work [10, 25, 30, 43, 51] in that we analyze the code from different point of view: our approach is based on the number of variables used in the algorithm, roles of variables [59], number of loops and blocks and so on. We will explain this in more detail in Chapter 4.

This thesis is structured as follows: Chapter 2 explains the background and the objectives of the work. Chapter 3 introduces the previous work about program analysis and code recognition, and explains their approach. Chapter 4 is an overview of our approach: we present our approach in detail and explain the characteristics of the code we have used for program analysis and code recognition. Chapter 5 outlines the method used in designing the system emphasizing the database structure and briefly descries the application architecture and class diagram. Issues related to the implementation of the system are also described in this chapter. Chapter 6 discusses the results and finally Chapter 7 concludes the whole thesis presenting some suggestions for future work.

Chapter 2

Objectives and Motivations

2.1 Background

This work is part of the TAPAS (Text and Program Analysis) project, which is a joint project between the Laboratory of Software Technology at the Helsinki University of Technology (HUT) and the Department of Computer Science at the University of Joensuu. The objective of the project is to develop novel methods and tools for free text and program text analysis and evaluation. Such methods and tools could be used for, e.g., comparing specifications of a system with design diagrams and program code and developing feedback and information for instructors and students.

In this work, we concentrate on the program code analysis part. Our objective is to examine the possibility of developing a method for automatic algorithm understanding, which can perform an analysis on the subject program and provide the user with the information about what type the subject program seems to be or what algorithms does it look like. Therefore, in this work, the problem of understanding a program is considered to be the problem of classifying algorithms.

2.2 Requirements

As presented in Chapter 1, the problem on understanding a program can be divided into many different categories. As will be presented in Chapter 3, each research on the field is focused mainly on one particular aspect of the problem, leaving the other ones out of consideration. For example, knowledge-based program understanding methods attempt to provide a solution to the problem considering the problem as belonging to the first category, and plagiarism detection systems regard the problem as belonging to the third category. In this work, we consider the problem of understanding a program as a problem of classifying algorithms.

We present a static analyzer that analyzes the program code in order to carry out aforementioned tasks. The concept of "Roles of Variables", devised by Sajaniemi [59], plays

an important role in our static analysis. Other than roles of variables, our analysis is also based on other characteristics of the program code, such as the number of variables, information about loops and information about blocks. We will present a detailed explanation about the characteristics of the program code we have used in our approach in Chapter 4. In addition, we will carry out a thorough literature survey and study the related previous researches carefully in Chapter 3 to find out about the other possible approaches or code characteristics we could use in our work.

The Analyzer will be built on top of the VILLE platform [48]. Developed at the Department of Information Technology at the University of Turku, VILLE is a visual tool designed to be used by teachers and students in programming courses. VILLE helps novice programmers in learning programming by highlighting the code line that is being executed, and showing how each line's execution affects the program state.

The Analyzer will be implemented in Java and will process programs written in Java language. There are several reasons for choosing Java, some of which are as follows:

- VILLE is written in Java.
- Java is a widely used programming language in computer science courses. At least the programming courses taught at HUT which may use the Analyzer, use Java as the programming language. Moreover, the TRAKLA2 system [40], into which the Analyzer is likely to be integrated, is developed in Java. TRAKLA2 is an automatic assessment system for data structures and algorithms, which is currently used at HUT and some other universities.
- Java is also often used in software industry. Presumably, software projects using Java make up a large proportion of whole new software projects, and according to author's experiences, the proportion seems to be changing constantly in Java's favor.
- Other Java-related issues, like portability, complexity and implementation concerns.

The Analyzer should work as a text-based interface that takes the subject program file as an input from the command line and outputs the textual results. The output should be as concise as possible. This conciseness, however, should not result in losing any valuable information which may be useful to the user. An example output would be: "The program appears to be a Quicksort algorithm".

Considering the complexity issue of the problem of understanding a program discussed in Chapter 1, the complexity issue must be taken into account in our approach, and the Analyzer must be efficient enough.

2.3 Scope

Program analysis and automated program understanding is obviously an extensive research field in computer since. As a result, we must carefully define the scope of our work. As the

name of this master's thesis suggests, our work is limited to process only sorting algorithms. In particular, we will consider the following well-known sorting algorithms, leaving all other algorithms out of scope of this work: Quicksort, Mergesort, Insertion sort, Selection sort and Bubble sort.

In this work, we focus on the main subject, which is to analyze the subject program with the purpose of extracting the information we need. Therefore, we will leave out of consideration other features like user interface related issues, etc.

Chapter 3

Related Work

In this chapter, we will discuss various studies, techniques and methods related to code analysis as thoroughly as it is possible in the scope of this work. The techniques and methods investigated here could be used as an input in the future work on the subject, although again, taking all of them into account in designing and implementing of the system in this work is probably not possible.

The structure of this chapter is organized as follows: an overview on different types of program analysis is presented in the next section. Each of these types is then discussed in its own section, and is divided into subsections if necessary. As an exception to this, the categories under Section "Static Program Analysis" are sections themselves, which are in turn further divided into subsections. This is done to improve readability, since the "Static Program Analysis" is a lengthy chapter. The chapter will be closed with conclusion where each presented approach is evaluated highlighting its strengths and weaknesses, and a brief summary where the useful approaches for our purpose are presented.

3.1 Types of Program Analysis

Program analysis could roughly be divided to two main categories based on whether the program is executed or not:

- Static program analysis
- Dynamic program analysis

In static program analysis, a program is analyzed without being executed. Static analysis is a structural analysis, which describes the behavior of the program in a general manner, regardless of what the input of the program possibly is. In this sense, static program analysis could be referred to as a kind of analysis that runs the code with "all possible inputs". Because static program analysis could be considered the way of analyzing the program to comprehend its behavior at run-time without being restricted by using a specific input,

it is regarded as a fast way to analyze the program thoroughly, but its accuracy has been criticized [29]. This claim about the accuracy concern of static analysis is based on the fact that static analysis examines the code with several inputs at the same time. As a result, it can offer only information about upper and lower bound within which the output of an imaginary input could occur. That is, the output cannot be verified, since there is no specific input. Control flow and data flow are two most commonly used techniques in static analysis.

Dynamic program analysis, on the contrary, includes code execution. To understand the program's behavior, we execute it by some input, and then examine its output. In order to understand the program better, we have to execute it using many different sets of input. This makes dynamic program analysis slower than static program analysis. According to the same aforementioned claim, dynamic program analysis is, however, more accurate than static program analysis, since its output is resulted from a precise, concrete and verifiable input, rather than a set of undefined and assumed inputs from which the resulted outputs can not be verified [29].

3.2 Static Program Analysis

In static program analysis the program is analyzed using structural analyzing methods, without running the code. Static analysis on a program code can be carried out in many different ways, focusing on different features of the code. For example, the goal of static analysis could be investigating the program's control flow, investigating the program's data flow, investigating the complexity of the program using different metrics, etc.

Most of the researches on automatic program understanding or related subjects are based on static program analysis. Program understanding and related works can be divided into the following categories based on the techniques and the approaches used.

- Knowledge-based approaches
- Using fuzzy reasoning in program understanding
- Program similarity evaluation approaches
- Reverse engineering approaches
- Other approaches

Knowledge-based techniques are based on a knowledge base that stores predefined plans. To understand a program, program code is matched against the plans. If there is a match, then we can say what the program does, since we know what the matched plans do. The plans can have other plans as their parts in a hierarchal manner. Based on whether the recognition of the program starts with matching the higher-level plans first or vice versa, knowledge-based approaches can be further divided into three subcategories: top-down, bottom-up and hybrid approaches.

The researches on knowledge-based techniques approaches started in the 1980's and continued to the late 1990's. These approaches have not been studied recently as extensively as they used to be in the past two decades.

Program understanding approaches using fuzzy reasoning are essentially the same as knowledge-based approaches, in that they both use plan library as their knowledge base. However, they can be distinguished by the fact that purely knowledge-based approaches do not use fuzzy reasoning in the process of deciding what the program does, whereas other approaches do. These approaches were researched in the late 1990's when researchers sought new methods to improve knowledge-based techniques.

As the name suggests, program similarity evaluation techniques, i.e., plagiarism detection techniques are used to answer to the question about whether two given programs are the same and to what extent. Although these approaches do not directly offer a solution to the program understanding problem, they cover such techniques that might be applied to find a solution to the problem. These approaches can be further divided into two subcategories: attribute-counting approaches and structure-based approaches. In attribute-counting approaches, some distinguishing characteristics of the subject program code are used to find the similarity between the two programs, whereas in structure-based approaches the structure of the code is examined to solve the problem.

From these two approaches, researches on attribute-counting approaches were started earlier. As a matter of fact, the researches on these approaches started shortly after Halstead's parameters and metrics were introduced in 1970's. The researches on structure-based approaches, on the other hand, were started later in 1980's and, although not as extensively as before, still continues, whereas the latest researches on attribute-counting approaches can be found from late 1990's.

Reverse engineering approaches are used to understand a system in order to recover its high-level design plans, generate documentation for it, rebuild it, extend its functionality, and so forth. Reverse engineering approaches are the youngest methods employed in the field of program understanding. The researches on these approaches were started in 1990's, and still continue extensively.

There have been many other attempts to develop a technique for understanding programs. As will be discussed later in more details, these techniques include clone detection methods, program understanding based on constraints satisfaction, task oriented program understanding and data-centered program understanding. These approaches are introduced mainly in 1990's and 2000's.

In the following, we describe these approaches in more details.

3.3 Knowledge-based Program Understanding

Most of the tools developed for automatic program understanding use knowledge-based program understanding techniques. The basic idea of knowledge-based program understanding is that we simply compare the input source code with the code snippets from the

library, which are frequently used stereotype codes or idioms. These code snippets are often called *plans*, *clichés*, *chunks*, etc. Since we know what the plans do, we can easily say what a piece of source code does, if we find a match between that piece of the source code and a plan.

Knowledge-based program understanding is often referred to as a cognitive process of program understanding, suggesting that the process of understanding a program is about understanding the *goal* of that program, i.e., the intention of the programmer, which is achieved using plans, that is, the techniques used for implementing the intention [50, 57]. In this concept, writing a new program is a process of rewriting the goal of that program into a set of subgoals using plans as the rules for this rewriting process. Program understanding can be regarded as a reverse process: understanding the goal through understanding the subgoals using plans. In this reverse process of understanding a program, single plans that are recognized from the subject program are combined in a hierarchical manner into plans with higher-level abstraction. The final goal of the subject program is ultimately recognized by continuing this process. Heuristics and artificial intelligent techniques are often exploited in the process of concluding the goal of the subject program.

Separate and simple plans can be composed in a complex way to create a program. Algorithms can be considered to be formed from plans. For example, as Letovsky and Soloway describe [57], a mergesort can be thought of to be comprised from several plans: a plan for recursion on a binary tree, a plan for splitting a sequence into two, a plan for sorting pairs of numbers, and a plan for merging sorted lists. Therefore, plans are not to be confused with algorithms or procedures since they are conceptually different.

Knowledge-based program understanding has been a subject of research for nearly two decades, and there are different approaches developed around it aiming to make it more efficient and more accurate. It is mostly a bottom-up approach, meaning that first we try to recognize and understand the small pieces of code, i.e., basic plans. After recognizing the basic plans, we can continue the process of recognizing and understanding higher-level plans by connecting the meaning of these already recognized basic plans and by reasoning what problem the combination of basic plans tries to solve. By continuing this process, we can finally try to conclude what the source code does as a whole. Some top-down approaches of knowledge-based program understanding have also been presented [33]. The idea here is that by knowing the domain of our problem, we can select the right plans from the library that solve that particular problem and then compare the source code to these plans. If there is a match between source code and library plans, we can answer to the question about what the program does. Since we have to know the domain, this approach requires the specification of the problem.

In addition to these, some approaches hybrid top-down and bottom-up techniques. An example of these combined approaches is [51], where programming plans are first recognized in a bottom-up manner, after which 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 an indexing facility, the system is able to quickly associate a piece of the source code with a plan in the knowledge base.

Rudolph Seviora [63] has divided the knowledge-based program debugging systems into three groups: Program analysis approach, I/O-based approach and Internal-trace-based approach. In program analysis debugging, as the name suggests, the source code is analyzed and matched against system's knowledge base. This approach can be further divided into two groups: bottom-up and top-down. I/O-based debuggers, e.g., Falosy, Fault Localization System [62], execute the program and compare the output with the model output. In Intemal-trace-based systems, like Message Trace Analyzer [23], debugging is carried out by tracing transactions internal to the program.

In the following, we explain a few studies on knowledge-based program understanding.

3.3.1 PAT (Program Analysis Tool)

PAT (Program Analysis Tool) [25] is a system designed for assistance software maintainers. It is a bottom-up approach for understanding programs functionality; the PAT builds events from the program and matches these events to its plan library. If it finds a match based on data and control flow, it can say what the subject program does. The PAT can answer to following three questions: What does the program do, how does the program do it and is the program buggy.

The main modules of the PAT are as follows:

Program Parser: Parses the given programs to the events and stores them in the Event Base.

Plan Parser: Parses the plans and stores them in the Plan Base.

Understander: Builds new high-level events based on the low-level events. These new high-level events are derived using deductive-inference-rules based on program's plan library found in the Plan Base.

JTMS (Justification-based Truth-maintenance System): Records the new events found by Understander and thus provides the PAT with ability to maintain the logical connection between events. It also records the information about how the new events are derived. This information can be used by Explanation Generator to answer the question about how the program does what it does.

Explanation Generator: Explains in detail how a high-level event has been derived from the low-level ones. To do so, it uses the information recorded by JTMS.

Paraphraser: Using the value of text field of each plan and the output of Explanation Generator on how a high-level event is derived from the low-level events, Paraphraser generates program documentation in natural-language. It does so by exploring the JTMS from the top.

Debugger: Finds and expresses program bugs by comparing the given program's events to the events of the plan from the system's library that has been recognized by the system as a match.

In PAT, the knowledge is represented using an object-oriented approach. The program has been thought of as comprising from events. Each event is an instance of an event class.

Each event has an attribute denoting the class it belongs to. Most common attributes are defined in a top-level class, from where each event can inherit them. Each event has also an interval, which can be either control interval (expresses the place of the event in the control path) or lexical interval (expresses the place of the event in the program).

Each plan has 4 parts: path (event-path-expression), test (binding-constraints), text (documentation information) and miss (near-miss-expression). An event is recognized as a match to a plan if the event's interval meets the requirements of the path expression of the plan and satisfies its test part. The text part of the plan contains the explanation of what it does, and the miss part stores the information related to debugging.

Recognition of events in the PAT is heuristic-based in a way that the PAT recognizes an event as a mach even though it does not go exactly like sequences in plan line by line, as long as it meets control and lexical requirements of the path. In other words, PAT examines the equivalency of the parts that look important, and ignores the rest.

3.3.2 Flow Graph Parsing

A different approach for program recognition is presented by Wills [17, 43]. In this approach, program is first analyzed. Analysing the program includes macro expansion, control flow and data flow analysis. Macro expansion phase consists of transforming the program into a more primitive form, which is argued to make the process of matching the program and library plans easier. The output of analysis phase is the plan representation of the program. In the second phase, a flow graph, which is a direct acyclic graph (DAG), is generated from the program plans. After this, the flow graph is parsed with the grammar derived from library plans. Derivation of the source code can help us understand the program, as we know how it maps to the library of clichés [71]. The order of the statements and function calls in the source code has no effect on recognition, since the method uses control flow and data flow analysis.

3.3.3 PROUST

Developed by Johnson and Soloway, PROUST [33] is a knowledge-based system for debugging and understanding Pascal programs written by novice programmers. PROUST uses top-down approach, where the idea is based on the methods used by experienced programmers when they try to understand a given code with knowing the goal of the code: from his experience, a programmer has some plans in his mind about how that goal can be achieved. He breaks down the code into smaller pieces, i.e., plans and examines whether those plans conform to the ones he expected. If the plans used in source code are different from those the programmer expected, then there could be either a bug in the code, or a new plan that the programmer was not aware of could be used in the code. In the latter case, the programmer could learn from the new plan and add it into his knowledge base.

Johnson and Soloway argue that reconstructing the program written by novice programmers is the best approach to understand the program and find its bugs, and they call this approach

reconstructive program analysis. In PROUST paradigm, a program consists of *goals*, and to achieve goals, program uses *plans*. To understand the program, PROUST selects a goal from the program specification and then tries to match the plans implementing that particular goal that are stored in its knowledge base against the code. If the goal itself contains other goals, the process is carried out recursively. After processing one goal, the next goal is selected and this successive decomposition of goals is continued until all goals are processed. The fact that PROUST needs the input program specification to select a goal from, is one of its most important limitations [17]. Plans are designed and added to PROUST in a manner that they can cover as much programs intention as possible. For example, plans that novice and expert programmers use might be quite different. It must be taken into consideration, however, that the more plans there are in the system the more difficult it is to determine what plan the program has been aiming at to use. The order in which plans occur is also important, since using the same plans in different order could result in different outcome. In PROUST, a set of heuristics are used to decide between different possibilities about what kind of bug may be in question, to evaluate matches, near-misses, misses, and so on. At the moment this article was written, the knowledge base of PROUST was limited to one novice program and it was to be extended to cover introductory programming tasks.

3.3.4 Concept Recognition System

Kozaczynski et al. [39] have developed a concept recognition system that is able to find the concepts in a program and recognize these concepts from the lower-level concepts. It uses a library of concept recognition rules as its knowledge base. Kozaczynski et al. argue that their system is designed by taking into consideration the practical constraints that are typical in automatic program understanding systems: usability, scalability and generality. Usability means that it should be easy for a user to modify the knowledge base of the system and add new concepts to it. By scalability they mean the ability of the system to handle real-world programs rather than only "toy programs". Generality refers to the ability to understand a wide range of different programs written in different languages.

3.4 Using Fuzzy Reasoning in Knowledge-based Program Understanding

BUG-DOCTOR [10, 11, 12] is a knowledge-based system for program comprehension and fault localization. Instead of performing exhausting and costly task of comparing the code to all plans, that is the approach commonly used in knowledge-based program comprehension, it first recognizes a group of statements as a partition of the code that can be mapped to a higher level concept, and represents them as code chunks. Code chunks, or simply chunks, are pieces of code that can be understood apart from other parts of code, and do something meaningful independently. Chunk is sometime referred to as a cliché, a plan, a schema, a slice or a prime [11]. An example of chunk is "swap values". Chunks, generated by BUG-DOCTOR, are abstraction sets of the program code, that help programmers in

laborious task of understanding the program and supporting them to construct a comprehensive and accurate mental model of it. The idea and the way of automated chunk generating is consistent with the way that, according to empirical models, programmers gain understanding of program code: programmers start the process of understanding the unfamiliar code by reading the individual statements and grouping them into a larger coherent components, called chunks, that help understanding the whole picture. By producing chunks in the same way, automated chunks generator could help programmers to build the mental model of the program code. At the beginning, potential chunks are identified from the source code. In larger programs, there might be a huge number of potential chunks identified. Thus, there is a need to somehow reduce this number by selecting the most promising chunks from potential chunks. In BUG-DOCTOR, this is done by using some heuristics, resulting in a smaller group of chunks, called candidate chunks [11]. Once the candidate chunks are known, the system retrieves from the plan library only those plans that look similar to these chunks. It ranks these plans according to their similarity to the code and selects the highest-ranked plans to be more closely investigated and compared. This more detailed investigation and comparison includes computationally more expensive operations, and by not having to compare all plans to the code using these operations, a considerable amount of time can be saved. Fuzzy reasoning is used for finding the most similar plans and ranking them. In order to apply fuzzy reasoning, first we have to find those kinds of code characteristics that could distinguish the codes to be compared, and thus help us to effectively find the similarities between the plans and chunk.

In the BUG-DOCTOR, code characteristics studied by Berghel and Sallach [8] are used as similarity measurements. These code characteristics are shown in Table 3.1. After selecting suitable code characteristics as similarity attributes, the next step is to quantify them. There are various approaches on quantification of the selected factors. The best and most suitable approach should be selected based on common sense and with consideration of context and the type of the problem. The next step is to convert the quantified factors to the linguistic fuzzy set, using for example LOW, MEDIUM and HIGH indicating low, medium and high percent of differences between the plans and the code chunk respectively. Moreover, each member of this fuzzy set can be represented by a member of closed interval [0.0, 1.0]. When the value of each factor is calculated, a set of rules are generated. These rules are then used in reasoning: a single variable is derived from different values of various factors. This single variable which can be named as, e.g., SIMILARITY, indicates the closeness of particular plan to the code chunk. A rule, for example, can be as follows: if the value of number of variable is LOW and the value of unique operands is MEDIUM, then the SIMILARITY is MEDIUM. The last step in the process of reasoning includes combining the results of the rules to one single digit that represents the final result: the similarity of the plan and code chunk. Again, there are various methods for generating the rules and combining the results to the single final digit, and the most suitable should be selected depending on the problem.

Table 3.1: Code Characteristics Used by Berghel and Sallach

Code characteristics	**Effectiveness[0-1]**
Total operators	0.99
Assignment statements	0.98
Total operands	0.95
Real variables	0.91
Initializations	0.90
Total variables	0.87
Unique operands	0.85
Total lines	0.78
Unique operators	0.75
Keywords	0.52
Others with lower effectiveness	-

3.5 Techniques Used in Program Similarity Evaluation Systems

Detecting program similarity has been a subject of interest for a long time. The main problem in program similarity researches is to detect plagiarism. The goal is to develop methods that can help the user with the problem of figuring out in an effective and accurate manner, whether two programs are similar and how closely. These tools are mainly used at universities to reveal plagiarism between students' submissions and prevent students from copying other's works. Therefore, these approaches do not attempt to solve the problem of understanding a program directly. Despite this, we will discuss program similarity approaches here for the following reasons.

Program similarity approaches include various interesting methods. These methods perform such different interesting static analyses on the subject program that can be applied in program understanding, too. These analyses include structural analysis, control flow analysis and data flow analysis, to name a few. In addition, program similarity researches examine the code from such point of view that can be useful in program understanding, as well. For example, as will be discussed in the following in more details, attribute-counting approaches that can be considered as a subcategory of program similarity researches, investigate the code with regard to metrics like operands, operations, assignment statements, and so on. Especially in our approach, these metrics could perhaps be used in process of recognition an algorithm. Thus, we find it well justified to discuss these approaches in more details in this thesis.

Methods used in plagiarism detection researches vary from those which try to solve the problem by counting the program attributes without considering the structure of the program, e.g., [22], to those looking to the problem from structural analysis point of view only, e.g., [69]. Some methods make use of both these two approaches. An example of these methods is [18]. It is also possible to classify plagiarism detection systems based on whether they are designed to examine natural languages, or program code. Detecting

Table 3.2: The Halstead's parameters

Symbol	Explanation
n_1	Number of unique operators
n_2	Number of unique operands
N_1	Total number of operators
N_2	Total number of operands

plagiarism in natural language is generally considered to be more difficult, for example, because of ambiguity and complexity of natural language [16]. Some tools, though, are capable of carrying out both program code and natural languages plagiarism detection [47].

In the following, we present a survey on some researches on this field. We divide the researches into two main categories, namely, attribute-counting methods and structure-based methods. Those researches that use both methods are described in attribute-counting category. It should be noted that since we are interested only in those aspects of plagiarism detection techniques that could contribute in program understanding, we leave irrelevant issues, like motivation and moral, out of our discussion. Also, those technically important issues in plagiarism detection systems that are not relevant in program understanding, e.g., number of comment lines, number of indented lines and number of blank lines, are not considered here. Moreover, the focus of this survey is, naturally, the systems processing program code rather than natural languages.

3.5.1 Methods Based on Attribute-counting

Earlier studies focus mainly on attribute-counting methods, while later studies emphasize the structural analysis. In attribute-counting methods some characteristics of program are selected as being important factors in distinguishing programs. The most important characteristics that have been used in researches, as we will explain in the following, are four Halstead's numbers. These characteristics are then counted from program code, and the result is compared with the corresponding result obtained from another program code following the same procedure. Finally, a number indicating the similarity between two programs is derived from the results by applying some formula, which varies in different researches.

One of the earliest approaches based on this method is Ottenstein [49] approach that uses the code characteristics mentioned in the Table 3.2 for detecting plagiarism. As Ottenstein claims, these characteristics give us a good indicator to investigate how similar or different two programs are.

In the following, we preset other researches that use these code characteristics to examine program similarity.

Donaldson et al.'s Approach

Donaldson et al. [18] criticize the Ottenstein method for being too broad, that is, the probability of two programs being tagged as similar using his method is high, even though Ottenstein himself argues that there is a slight probability for this to happen. This is especially the case in introductory level classes, where the students' submissions are comparatively simple. Donaldson et al. make use of structural analysis to determine the similarity between two programs. Using this approach, two programs can be considered as being the same, even though a student may have tried to make them look different using some simple methods. Using these simple methods often results in the following simple differences between the programs:

- Different name of variables
- Different order between statements when the order of statements does not change the functionality of the program
- Different position and form of format statements
- Different appearance in single statements such as variable declaration and output statements resulted from breaking it up

The system developed by Donaldson et al. is described in the following. Note that the following analysis relates to FORTRAN language.

The system takes students' submissions as an input file, analyzes each of them and generates a table containing information about the structure and content of each program. After processing all submissions, the system then compares the content of the table generated from each submission to all other submissions. If two submitted program differ from each other only in one or more of the four ways mentioned above, they are tagged as similar and suggested to the instructor. The system collects data from all submitted programs. Each type of statement is given a counter. The value of the counter is incremented by one each time the corresponding statement occurs in the program. The following are collected:

1. Total number of variables
2. Total number of subprograms
3. Total number of input statements
4. Total number of conditional statements
5. Total number of loop statements
6. Total number of assignment statements
7. Total number of calls to subprograms

8. Total number of statements of type 2-7

From the different types of statements listed above, some are regarded to be more significant in describing the structure of the program than others. These statements are processed, and others are simply ignored. As the system continues to scan the input program, the order of the statements is stored using the following coding method: A single letter is chosen to refer to each statement. As a statement is read, the corresponding letter is added to the code in such a way that the left-to-right order of the code corresponds to the top-to-bottom order of the statement in the program. An example of such code is 'VVSD===HI', where the code characters are assigned to statements as follows:

V Declaration statement

S Subroutine or function definition

D DO loop

I IF (conditional expression) THEN DO

H WHILE (conditional expression) DO

= Assignment statement

After an entire program is scanned, the generated code is added to an array, where the corresponding code indicating the order of the statements in other programs will be added too.

After data collection phase, the collected data is analyzed. Data analysis phase consists of two phases itself: in the first phase the degree of difference or similarity between the counters of any two statements are calculated. The result of this analysis is a digit indicating the similarity between each pair of statements. In the second phase, the statement order of each two pair of statements is compared.

The first phase contains the following methods:

- Calculating the sum of differences: The value of the counter of each pair statements is subtracted and the absolute values of these subtraction results are summed. The result indicates how different two programs are.
- Calculating the count of similarity: This method includes the calculation of the similarity factor between two programs. Similarity factor indicating the similarity between two programs is established having zero as its value at the beginning. The value of the similarity factor is incremented by one or remains unchanged depending of whether the corresponding counters between two programs are equal or not, respectively.

- Calculating the weighted count of similarity: This method is similar to the previous one, except instead of incrementing the value of the similarity factor by one when the corresponding counters are equal, the value of the similarity factor is increased by a particular amount set by instructor. This makes the system interactive and allows the instructor to emphasize the importance of some particular statement type or vice versa. For example, if the program in question is very simple, students' submissions are unlikely to contain any subprograms. This would result in incrementing the value of similarity factor by one, since the corresponding counters would be equals in all programs. The instructor could eliminate the effect of this statement type by assigning a value of zero to it.

The second phase, i.e., the comparison between the orders of the statements goes as follows: First, the similar code characters are compressed so that there remains no identical characters in the result. For example, the code characters 'VVSD===HI' become 'VSD=HI'. Since one common way for students to make a copy program look different from the original one seems to be breaking up single statements into multiple statements, this algorithm reduces a succession of the same statement type to one, making this trick inefficient to affect the results. After this, the process of comparing the code character begins. The process of comparison stops, when the first difference between code characters occurs, resulting in a mismatch. If the comparison ends without any difference occurring, a message is printed indicating that a match is found.

Donaldson et al. conclude that especially in elementary programming course, the programs might look alike according to the numbers generated by the algorithms. Therefore, the results should be further evaluated by instructor and be considered as relative to the entire class.

Donaldson et al. method looks simple and still efficient. Some points in their method seem to be suitable to be applied in program analysis as well. However, it should be noted that the second phase of their method, i.e., the reduction algorithms, could not directly be used in program analysis purposes. This is because deletion of successive loops and conditional statements in order to reduce the amount of them to one will lead into inaccurate results in program analysis.

Berghel and Sallach's Approach

Berghel and Sallach [8] have investigated the code characteristics and their effectiveness in their research related to finding plagiarism, and have come up with the factors and their effectiveness mentioned in Table 3.1.

Some of the code characteristics mentioned in Table 3.1 belong to the Halstead parameters [22], which are shown in Table 3.2.

After the Halstead's code characteristics shown in Table 3.2 are counted from the source code, Halstead measures can be derived from them. Some most common Halstead measures, their symbol and the formula for each one describing how it is derived from Halstead's code characteristics are shown in Table 3.3.

Table 3.3: The Halstead's metrics

Measure	**Symbol**	**Formula**
Program Length	N	$N_1 + N_2$
Program Vocabulary Size	n	$n_1 + n_2$
Program Volume	V	$N \log_2 n$
Program Difficulty Level	D	$\frac{n_1}{2} \frac{N_2}{n_2}$
Program Level	L	$\frac{1}{D}$
Effort to Implement	E	D * V
Time to Implement	T	$\frac{E}{18}$
Number of Delivered Bugs	B	$\frac{(E^{\frac{2}{3}})}{3000}$

Berghel and Sallach, on the other hand, have compared Halstead's measurements against a conceptually simpler alternative one in their study on program similarity [8]. They have applied these two measurements to detect plagiarism in the students' submissions. Their goal was to find out how efficiently and accurately these two measurements work in comparison to each other. The alternative measurements include metrics like numbers of Assignment Statements and Total Variables among the others. They state that Assignment Statements is a subset of Total Operators in Halstead measurements and that Total Variables could be treated as a counterpart to Total Operands in Halstead measurements. Berghel and Sallach then compared the number of Assignment Statements and the Total Variables with their counterparts in Halstead measurements in regard to specificity and broadness/narrowness. They explain that the number of Assignment Statements is more specific, or narrow, that its counterpart, and that Total Variables is likewise narrower than its counterpart. The reason why the Total Variable is narrower than its counterpart is that the former does not include constants, whereas the latter does. It is noteworthy, that Berghel and Sallach state other similarities between the metrics from the two measurements too, one of them being the relationship between Keywords in alternative measurements, and Total Operators in Halstead measurement. The difference between these two metrics in respect to specificity and broadness/narrowness can not be, however, clearly expressed, since it depends on the situation where they are used: some times Keywords is narrower than Total Operators, and sometimes vice versa. It thus seems to be reasonable to remove one of these similar pair from the Table 3.1, and use only one that looks best to the purpose.

Berghel and Sallach conclude that there is no reason to distinguish the isolated Halstead measurements as having unique features and regard them as absolutely more powerful than others. The most suitable and efficient measurements must be selected according to and depending on how broadly or narrowly the comparison between two program is intended to be done.

Robinson and Soffa's Approach

Robinson and Soffa take a different approach in their work [55]. In addition to plagiarism detection task, the system developed by them, called ITPAD (Instructional Tool for Program Advising), is designed to carry out the following tasks: investigating whether a student has used some particular structure in his program or not, following the student's progress by monitoring whether he makes use of particular programming facilities, evaluating the programming assignment, e.g., by ensuring that to accomplish a particular assignment, a student must use a particular set of procedures that are intended to be taught by doing that assignment, and finally giving suggestions for program improvement. It uses code optimization techniques and software metrics in order to analyze the program. ITPAD consists of three phases: lexical analysis, structural analysis and characteristic analysis. In the following, each phase is discussed.

Lexical analysis includes data collection in order to be able to count code characteristics. The code characteristics used by Robinson and Soffa to perform analysis are the following.

1. Total number of unique variables
2. Total number of variables
3. Total number of unique operator
4. Total number of operator
5. Program length
6. Vocabulary of the Program
7. Volume of the Program
8. Level of the Program
9. Intelligence content
10. Effort of the Program
11. Total number of assignment statements
12. Total number and type of control structure
13. Total number and type of data structure
14. Total number of function and subroutines, i.e., subprograms

The first ten characteristics belong to Halstead metrics. It is noteworthy that Robinson and Soffa have used the terms unique variables and variables instead of the terms unique operands and operands, which are the terms used in Halstead metrics. Whereas, as can be seen in Table 3.1, Berghel and Sallach [8] distinguish between total variables and total operands.

Robinson and Soffa use Ottenstein's method [49] as a control method in their approach. As mentioned earlier, Ottenstein uses the first four characteristics listed above in their method.

In order to be able to carry out the tasks discussed earlier, ITPAD makes two profiles using code characteristics: student profile and assignment profile. By comparing each student's solution with the model solution provided by instructor the system can tell the instructor whether the student has been able to use the required programming concepts in a successful way. The system is also able to report the student's progress by comparing his submission with previous ones during the term. The assignment profile consists of the report generated by ITPAD about how student have solved an assignment, i.e., how they have used control and data structures etc. This helps the instructor to ensure that the assignment is suitable for teaching the intended programming concepts to the students. The assignment profile phase employs program optimization techniques.

Structural analysis includes the following steps. The system first breaks the input program down into basic blocks. These basic blocks are then counted and compared to the basic blocks of the model program. Big difference between these may suggest that there is something missing from the student solution. The number of basic blocks also provides the instructor with an idea of how difficult the assignment is. The number of basic blocks is used in plagiarism detection too: if the number is the same in two submissions, then those submissions are probably structurally similar, and should be investigated more in details by instructor. Other steps in structural analysis are constructing flow graph, constructing Direct Acyclic Graph (DAG), performing data flow analysis and recognizing the loops. The results of these steps are used in the next phase to provide students with some improvement suggestions, if necessary.

Characteristic analysis Results of analysing the submission in the previous phase are investigated in this phase, and based on the result of this investigation and the result of comparing the student profile and the model profile, improvement suggestion are made to the student. As an example, improvement suggestion may be as follows: unnecessary statement found, remove it.

As described above, the main purpose of ITPAD seems to be providing improvement suggestions for students, although it is capable of detect plagiarism as well. Overall, it uses a good combined method by employing the metrics and the graph representation of the program.

Grier's Approach

Grier [22] has developed a system, called Accuse, for detecting plagiarism in Pascal programs. In addition to the Halstead's 4 parameters shown in Table 3.2, Accuse measures the following 16 parameters as Grier lists: total lines, code lines, code comment lines, multiple statement lines, constants and types, variables declared (and used), variables declared (and not used), procedures and functions, var parameters, value parameters, procedure variables (includes 9 and 10), for statements, repeat statements, while statements, goto statements, indenting function.

By testing different combinations of these 20 parameters, Accuse selects seven parameters, which are used in computing a correlation number. Various counting heuristics are also used in computing the correlation number. The correlation number determines the similarity between programs. The process of selecting the seven parameters is not discussed in more detail. The characteristics used in heuristics are total operators (excluding assignment operator) and code line (excluding blank lines, command lines and declarations). Parameters, like parentheses, that could sometimes be added to the program without changing the functionality, are not included in counting the operators. The seven characteristics used to compute the correlation numbers are total operators (5, 6), total operands (5, 6), unique operators (3, 5), unique operands (3, 5), code lines (3, 5), variables declared (and used) (2, 3) and total control statements (1, 2). The first number in the parentheses indicates the importance of the characteristic and the second number shows that how big the differences between that characteristic in two programs can at maximum be in order to that characteristic be taken into account in calculating the correlation number. The correlation number is computed by adding together the subtraction result of the importance value of each characteristic and the difference between that particular characteristic in two programs when this is to be taken into account. The correlating number grater or equal to 29 suggests possible plagiarism, with 32 being its maximum value. Grier admits that the correlating number is computed in an ad hoc manner, and there are probably other ways to compute it in a more efficient manner.

Rees's Approach

STYLE [54] is a system developed by Rees, originally meant for checking coding style in students' submission. A plagiarism detection system called CHEAT was afterward added to it. STYLE uses the following parameters for detecting similarity in programs:

- Total of non-comment characters
- Percent of embedded spaces
- Number of reserved words
- Number of identifiers
- Total number of lines
- Number of procedures/functions

Rees does not explain in detail how well CHEAT has been able to detect similar programs, but he states that in one experiment, CHEAT detected 3 out of 100 students' submissions as being copied, and 2 other pairs suggested by CHEAT as being similar were proven not to be copied after manual examination.

Leach's Approach

Ronald J. Leach [41] uses three complexity metrics in plagiarism detection: Halstead's metrics, McCabe complexity (Cyclomatic Complexity) and Conte, S. D., Dunsmore complexity.

From Halstead's metrics he uses effort metric (see the Table 3.3), as he points out as an example.

As Leach explains, McCabe describes program complexity as a graph. Each statement is a vertex of the graph. There is an edge from vertices A to B if B is executed immediately after A, that is, either B occurs after A, or there is a connection between A and B through a loop, condition branch or subprogram call. McCabe's complexity is computed by formula E-V+2P, where E is the number of edges, V is the number of vertices and P is the number of separated parts, i.e., subprogram called.

The third group of measures used by Leach includes S. D., Dunsmore's measures and is about flow of information between program modules via subprogram's parameters or global variables. As Leach explains, S. D., Dunsmore defines 6 different levels regarding to coupling between modules starting from no coupling and ending to content coupling. Leach computes these couplings as function calling including both user-defined and system-supplied functions.

Ronald J. Leach justifies his approach by the fact that the metrics he uses are not effective enough if used alone, e.g., Halstead's metrics takes into account only the number of operator and operands, and ignores the structure of the program, or McCabe's method considers only the control flow ignoring the number of operator and operands of the program. According to the results of his experiment, combination of these three metrics was proved to make a good plagiarism detector.

Jones's Approach

Edward L. Jones [35] uses the following characteristics of source code and Halstead's metric's in his method: Number of characters (c), Number of words (w) and Number of lines (l), Program length (N), Program vocabulary (n) and Program volume (V) (see Table 3.3).

He puts these two measurements together (l, w, c, N, n, V), what he calls composite profile, code characteristics and Halstead's metrics being called physical and Halstead profiles respectively.

Similarity between two programs is expressed as the normalized Euclidean distance between their profiles, the value of which being zero for two identical programs. The value indicating two programs being copied is not predefined, but varies from assignment to assignment.

Edward L. Jones admits that his method can be misled by student by, e.g., adding extra command lines or splitting a line to many. He also states that the value of similarity computed between two programs depends on programming language in question: programs implemented using verbose and structured programming language like COBOL may ap-

pear more similar than others.

Elenbogen and Seliya's Approach

Elenbogen and Seliya [19] look at the plagiarism from a totally different point of view. They have developed a tool to discover whether submitted solution to an assignment is student's own, or is it done by someone else outside the class, e.g., provided by some services provider from the Internet. Other plagiarism detection tools fail to solve this problem, since they detect similarity between programs submitted by students from the same class and on the same course. The method is based on the assumption, that students' programming style remains quite the same and does not change dramatically in short time. The main idea is as follows: the students' programming style are studied from their several previous submissions using the following metrics: number of code line, number of comment line, average length of variables name, number of variables, number of for loops/number of total loops and amount of bits in compressed program. Natural changing or improvement in programming which is acquired on programming courses is taken into account by comparing the changes in individual's programming style during the semester with changes in all other students' programming style. This process provides information about the programming style of each student. Having this information, the system investigates whether the program submitted by the student is his own or not. This is down by comparing the style of the submitted program with the programming style of the student known to the system. The system uses data mining techniques.

Malmi's Approach

Lauri Malmi's study on detecting the similarity of Pascal programs [44] is unique and noteworthy: to evaluate similarity between programs, it applies both attribute-counting and structure-based analysis, and in addition, it performs a dynamic analysis, too. In Malmi's method, programs are first investigated using attribute-counting technique, which is not as costly as two other techniques. After this, the investigations continues with structural-based analysis, and at the end, with dynamic analysis. In dynamic analysis, programs are executed using the same input, and the number of execution of some particular parts of the programs, e.g., if-statement, are calculated and compared. In each phase, the similarity between programs is indicated by a percent number between 0 and 100. The results obtained by all three analyses contribute to the final decision on how similar two programs are.

3.5.2 Structure-based Methods

Later researches on plagiarism detection focus on Structure analysis, rather than characteristics analysis. This has been justified by claiming that attribute-counting methods fail to detect the copied programs that have been modified even in textual manner, and that structure-based methods are much more tolerant to different modifications imposed by student to make the program look different [47]. Structure-based methods can be divided

into two subcategories: string matching based systems and tree matching based systems. As Maxim Mozgovoy describes [47], string matching based systems use different string matching methods, such as the Running-Karp-Rabin Greedy-String-Tiling used, for example, in [69] and parameterized matching algorithm used in [4].

Joy and Luck's Approach

Mike Joy and Michael Luck [37] introduce a new approach in plagiarism detection problem. They state that the lack of many desirable features in previous plagiarism detection systems have motivated them to develop a new one. As an example of these lacking desirable features they mention the fact that a plagiarism detection system should easily and without a major effort be changeable from one language to another, since a student is likely to use more than one language throughout their course. SHERLOCK, system developed by Mike Joy and Michael Luck, compares two programs five times in the following steps:

- Original programs
- Programs with the maximum amount of whitespace removed
- Programs with all comments removed
- Programs with all comments and maximum amount of whitespace removed
- Programs translated into tokens

One or more matches between two programs during this five-step comparison indicates potential similarity between those programs and gives a reason for further examination. All comparable programs are compared with each other and similar sequences are saved in a record file. Two sequences are considered similar if the difference between them does not exceed a predefined value, which is changeable by the instructor. For example, 10-line-long sequences are marked similar by default, if there is only one extra and one missing line in one of them, but are not marked similar, if the number of extra and one missing lines are 3 and 3, respectively. They argue that building the system merely based on the comparison between the token sequence would result in a false match especially in introductory programming classes, where there are not so many alternative algorithms to implement an assignment. This incremental comparison ensures that those unnecessary matches do not occur.

After all programs have been compared, a neural-network is invoked to create a visual PostScript file, where the similarities between programs saved in the record file are depicted. In the generated PostScript file, the original files are shown as points and the similarity between two points is illustrated by a line joining those points. There is no line between totally different points. Length of the line drawn between two points indicates degree of similarity between those two points: the shorter the line, the stronger the similarity. The visualization of the result of the comparison between programs can help the instructor to get a quick comprehension about similarities between programs and is thus very useful.

Gitchell and Tran's Approach

Sim [21] is a system developed for detecting similarity between C programs. Sim generates a sequence of tokens from program code removing white spaces, comments and all other dispensable parts. After this, Sim uses the string alignment algorithm to compare two programs. The similarity between two programs is indicated by a number between 0.0 and 1.0.

Schleimer et al.'s Approach

MOSS [60] is a plagiarism detection system based on string matching. Three aspects was deemed important by developers when designing the system: inessential characteristics, like whitespaces and capitalization, should not have any effect on result, common words and idioms should not imply plagiarism and changing structure of program code, removing or adding parts to it should not affect result. These are taken into consideration as follows. When tokenizing program code, all letters are converted to lower case and all inessential characteristics including punctuations are removed. The string matching algorithm is then applied to carry out the comparison. The second above-mentioned issue is solved by choosing a sufficient long part of token sequence to compare, so that common words and idioms are shorter than this chosen threshold.

A tree matching approach is presented in [38]. In this method, first the program code is parsed and an abstract syntax-tree is generated. Abstract syntax-tree is then arranged in sequential order of nodes in a linear manner. This is done in a process, which Kim and Choi call the "unparsing". The comparison between two programs is carried out on these node strings. Similarity is indicated by values varying from 0 to 1, 1 meaning two programs being completely similar. The method uses a faster algorithm in the beginning of process to make sure that two programs are not completely different. Further detailed investigation is carried out only if two programs appear to be similar to some extent. This way, the number of comparisons is reduced resulting in improved efficiency.

Ji et al.'s Approach

Ji et al. [32] have recently introduced a method that they claim being more effective than all other previously released plagiarism detection methods. In their method, first program code is parsed and a symbol table is generated. After this, the program is statically executed at syntax-level using what they call static tracing method, and then it is transferred to a set of predefined sequences of tokens generated according to the executing order. These sequences of tokens are then compared with each other to evaluate the similarity between programs.

Wise's Approach

YAP3 [69] is an improved version of YAP [68] that is capable of detecting plagiarism both in program code and natural language. It first tokenizes the source code removing, e.g., comments and performing some other changing on it, after which it compares the programs using Running-Karp-Rabin Greedy-String-Tiling string matching algorithm.

Mozgovoy et al.'s Approach

Maxim Mozgovoy et al. [47] present a fast substring matching algorithm for plagiarism detection. The algorithm uses a suffix array as an index data structure, making some improvement in efficiency of the previously known substring matching algorithms. In brief, the method works as follows: it first produces a token sequence from the program code, then selects neighboring parts of this token sequence in turn and tries to find a substring match for it from the other programs. Every match has an increasing effect on the similarity value. The similarity value is computed by dividing the number of matched tokens by the total number of tokens in the program being compared.

Mozgovoy et al. introduce a new plagiarism detection tool that is based on the combination of this fast algorithm and a reliable plagiarism detection system called Plaggie [1]. Their method consists of two phases. In the first phase, all comparable programs are processed using the fast algorithm, and in the second phase all suspicious programs found in the first phase are investigated in more details using Plaggie. Mozgovoy et al. argue that their combined approach is more efficient than earlier released tools. They also claim that this efficiency is not achieved by the cost of reliability, since Plaggie, which is responsible for more detailed comparison of the potentially similar programs, is one of the most reliable plagiarism detection systems ever released. Plaggie could be considered as an open source version of JPlag regarding to its functionality, and JPlag is a well-known reliable plagiarism detection system that uses enhanced Running-Karp-Rabin Greedy-String-Tiling algorithm [47].

3.6 Reverse Engineering

Reverse engineering tools offer useful methods to maintainers to understand the code they are maintaining. In reverse engineering the structure of the code is investigated by going through the program code and analyzing, e.g., control flow, data flow and data structures in the program. As a result of these analyses, various reports and documents can be generated describing the functionality of the system, helping maintainers to gain a quick understanding about the system and allowing them, e.g., to see directly what parts of the system are going to be effected by the change they are about to make.

Since program understanding is, on one hand, an essential part of maintaining and reengineering legacy systems [66] and reengineering and, on the other hand, forms the core part of reverse engineering, program understanding and reverse engineering can be regarded as

two very closely related research fields. Therefore, it is well justified to have a brief survey on reverse engineering techniques here.

The word "reverse engineering" is usually used in its broader meaning, which can be articulated as "reverse engineering approaches". However, as will be discussed in the following, it has a precise meaning that can be distinguished from the other concepts of reverse engineering approaches, e.g., from "reengineering". Reverse engineering, in its broader meaning, is a process of investigating a system in order to understand it for some particular purpose. The purpose of reverse engineering a software can be, for example, to create high-level documentation for it, to discover its design, to fix its faults, to enhance its functionalities, to add a new functionality to it or to make a new software with the same functionality. The subject system being reverse engineered can be either a hardware system or a software system, although here we mean the latter one. In the following, we present a brief discussion on reverse engineering, since it is about analyzing and understanding software.

Chikofsky and Cross [14] present a definition for six terms related to reverse engineering approaches: forward engineering, reverse engineering, re-documentation, design recovery, restructuring, and reengineering. In order to gain a better understanding about the exact relationship between reverse engineering approaches and program understanding on one hand, and the interrelationships among different concepts of reverse engineering approaches on the other hand, it is worth having an overview of the definition of these terms.

Chikofsky and Cross explain these terms using software life-cycle concept represented as the waterfall model, which they assume to consists of only three phases with different levels of abstraction: requirements phase, design phase and implementation phase (including coding, testing and deployment of the system). The definitions are as follows.

Forward engineering is the traditional way of developing the system: starting from the requirements phase, i.e., the high-level abstraction and ending to the implementation phase.

Reverse engineering is the process of analyzing and examining the system in order to recognize its components and interrelationships between those components. Reverse engineering, moreover, produces a specification with a higher-level abstraction from the system. The output of reverse engineering process is thus an alternative way which the system can be represented with. This alternative way is more understandable and less implementation-dependent than the system itself. It must be noticed that in reverse engineering process the subject system is only investigated, meaning that no action is taken in order to rebuild the system, modify it, etc. Reverse engineering can, therefore, be thought of as a process being associated with actual understanding of the system more that other reverse engineering approaches.

Re-documentation is the process of creating documentation within a phase with the same level of abstraction. The main objective of re-documentation is to represent the system in a way that is easier for a user to understand. This objective is often achieved by visualization of the components of the system and interrelationships between them. Pretty printing and control flow diagram are examples of re-documentation. Re-documentation is a very useful process in maintenance activity, and as the name implies, is often used when there is no documentation available. Re-documentation can be considered as a subarea of reverse

engineering.

Design recovery can also be regarded as a subarea of reverse engineering which produces a documentation that is more informative than what is possible to obtain by merely investigating the system. This informative documentation with the higher-level abstraction is obtained by using domain knowledge, external information, deduction and fuzzy reasoning. As Biggerstaff expresses [9], knowledge base, or domain model, as he puts it, is an important part of every design recovery tools, and this is actually what distinguishes design recovery from reverse engineering. Personal experiences and existing design documentation are also employed in design recovery process. The output of design recovery tools should be able to answer to the user's questions about what program does and how or why it does it. One can conclude that since design recovery process requires more detailed input information, it is able to produce more detailed output information, respectively, in comparison with, e.g., reverse engineering process.

Restructuring means to represent a subject system in a different way at the same level of abstraction, so that the functionality of the subject system remains the same. Data normalization is an example of restructuring in the database design process which aims at improving the logical data model. Restructuring does not necessarily require the understanding of the subject system, as an example, many activities related to code refactoring can be carried out without knowing what the program exactly does. As another example, restructuring the subject system in a bid to make it compatible with the requirements of the new environment does not necessitate the code or the problem domain being understood.

Reengineering usually consists of reverse engineering and forward engineering or restructuring. It includes the investigation of a subject system to produce specification at a higher-level abstraction (reverse engineering part), the possible modification of the acquired specification in order to transform the system into a new form, and the rebuilding of the system in that new form (forward engineering or restructuring part). The purpose of reengineering is usually to add new functionalities to the subject system. In this sense, reengineering is a different process than reverse engineering, and must not be confused with it: in contrast to reverse engineering, reengineering involves implementing the subject system. According to Chikofsky and Cross, some confusion between reengineering and restructuring has been observed too. Chikofsky and Cross argue that although reengineering includes both reverse engineering and forward engineering, it must not be regarded as some supertype of these two. These are different technologies that are evolving quickly and must be distinguished.

After presenting the aforementioned taxonomy of reverse engineering approaches, Chikofsky and Cross lists the following six key objectives for these technologies.

- Overcoming the complexity problem: Reverse engineering techniques provide software developers and maintainers with a way to understand complex systems so that maintaining them becomes easier. This is carried out through extracting the desired information out of those complex systems.

- Representing the system in different ways: Representing the system in different graphical ways can help in comprehension and verification the system. This is a

valuable step in completing the system documentation, which can be achieved using reverse engineering techniques.

- Updating design documentation: Especially in the case of legacy systems, design documentation does not often reflect the correct and exact information about the system. It is due to the fact that the system is modified during its life-cycle at the code level, but not at the design documentation level. Design recovery techniques can be used for recovering this missing information and updating the design documentation to correspond to the code.

- Detecting ripple effect: Ripple effect, i.e., side effect that could occur as a result of bad design or successive modification of the system, could be detected effectively using reverse engineering techniques.

- Generating higher-level abstractions: Reverse engineering techniques can be used to generate a higher-level abstractions as an alternative and more understandable view of the system.

- Reusing software components: Reusable software components can be detected using reverse engineering techniques.

Criticism of Reverse Engineering

Although reverse engineering techniques have become very popular during past two decades and various useful tools have been developed based on these techniques to help maintainers and developers in their work, some aspects of these techniques have also been criticized. As briefly presented in the following, these criticisms mainly pertain to the generality, scope, accuracy and efficiency concerns.

Holtzblatt et al. [30] criticize the reverse engineering tools on not being able to fully contribute in automatic understanding of the program code, which include higher-level abstractions, such as distributed behaviour and interprocess communication and control; features of the system that are not explicitly represented at the implementation level in the code. The inability of reverse engineering tools comes from the fact that these tools are able to analyze and extract information from the systems that function in a sequential manner, and not in the way, where individual tasks are executed concurrently on different processors, or on the same processor by interleaving the functions. Capability of reverse engineering tools in analyzing the program code and extracting information from it is based on explicitly represented data in syntax of the program language. For this reason, reverse engineering tools are not able to extract information from, e.g., concurrent systems since these systems can not be fully understood by only gaining an implementation level understanding from them, but understanding them requires a thorough understanding of their design, the processing model they use, and the way this processing model is implemented in a particular operating system. Holtzblatt et al. then conclude that as long as reverse engineering methods extract information from program code based only on the implementation level, and do not go beyond the analyzing the code syntactically by using the design information, they will fail to

offer proper information about concurrent, or any other kind of systems implemented using a particular software architecture. They have developed a tool making maintainers able to recover those aspects of system that reverse engineering tools are not able to understand from the code. We will not explain this tool in more detail here, since it is out of the scope of this work.

Quilici [52] criticizes the idea of completely automated extraction of abstract formal specification from a legacy system. He refers to the following definition of reverse engineering presented by R. Arnold's [3]: "The process of deriving abstract formal specifications from the source code of a legacy system, where these specifications can be used to forward engineer a new implementation of that system.". Quilici then asks a question about whether reverse engineering a legacy code has any chance to succeed in the way that aforementioned definition introduces it? He answers negatively to the question highlighting the following three issues from the definition as the reasons why reverse engineering is likely to fail: Firstly, the definition argues that the process of extracting abstract formal specifications from source code is totally automatic. Secondly, according to the definition, the extracted specifications are so accurate and at the proper level of abstraction that a new implementation can be carried out from them. And finally, the definition implies that the effort, time and cost of completely automatic extracting of abstract formal specifications are less than if the specifications were generated from the scratch without using the source code of the legacy system.

Quilici argues that the situation in real-word confirms his claim: a tool that automatically and cost-effectively derives abstract formal specifications needed for a new implementation of a legacy system has not been presented yet. Quilici concludes that the goal of reverse engineering should be extracting an informative knowledge base from a legacy system that can help to forward engineer it, rather than extracting a complete abstract formal specifications. Moreover, the process of extracting the knowledge should not be attempted to make totally automated, but rather to be a combination of both automated and human phases. Quilici presents the following definition of reverse engineering which he thinks have a chance to succeed: "The automated or assisted process of deriving a knowledge base describing a legacy system from its source code, where this knowledge base lessens the effort required to forward engineer a new implementation of that system.".

Stroulia and Systä [65] criticize the using of static analysis in reverse engineering techniques as traditional and insufficient technique that if used alone, can not lead to desired results any longer. The reason they present is that many legacy system are nowadays object-oriented and are used in distributed environments. Moreover, legacy systems today are more complex than what they used to be. In the past, software systems were mostly written in procedural languages and were run in a single computer. In those days, in order to understand a system and extract abstract specifications required in reengineering or maintenance activities, it was enough to investigate the components of the system and understand the interrelationships between them. Today, situation has changed: legacy systems are used in a distributed environment in connection with other systems. Therefore, the required result of reverse engineering process is a specification of subsystems that function together. Their solution to the problem is to investigate the system using a new approach: a combination

of static and dynamic analysis. Using reverse engineering tools that show the system's dynamic behavior as well as static diagrams will help maintainer gain more comprehensive and deeper understanding of the system. In addition to reverse engineering and legacy systems, these kind of tools can be useful in forward engineering activities as well. We will not discuss Stroulia and Systä's work in more details here, since it not within the scope of this thesis.

Reverse engineering approaches have also been criticized of not making proper use of *domain knowledge*. Domain-based program understanding is a method suggested to be used with program understanding techniques, like reverse engineering, in a bid to enhance the accuracy and efficiency of these techniques. The idea is that while, for example, reverse engineering technique is based on analysing the source code and tries to understand a program basically by performing the control flow analysis of that program without taking advantage of the knowledge about the purpose of the program, making use of the fact that what the problem area, i.e., domain of the program is, can be of much help in understanding the program [15, 56].

3.7 Other Techniques

In addition to researches presented above, there are other researches on different techniques to understand program or discover similarities between two given programs. In the following, we describe these approaches briefly.

3.7.1 Technique Used in Clone Detection Methods

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 system to be reused directly or with some small modifications, or is created by him without awareness of the existence of a code snippet elsewhere in the same system that solves the same problem and could have been reused. Clones make the maintenance task even more difficult and time consuming than what it is and it is thus important to find them in the system being maintained. The clone detection problem is about searching for the same or almost the same code in program and in this sense, it resembles the plagiarism detection problem. Since techniques used in clone detection methods solves the problem of finding the similarity between two pieces of code, it is well justified to be discussed here. In the following, we will briefly present some researches on this field.

Marcus and Maletic [46] introduce a new approach to detect high-level clones in source code. Their approach differs from other methods introduced previously in that the previous clone detection methods mainly detect similar codes using structural analysis, i.e., structural organization, control flow and data flow of the source code while the method presented by them detects clones by identifying the implementation of similar high-level concepts. Their method uses LSI (Latent Semantic Indexing) as an information retrieval technique to statically analyze the software systems and to identify semantic similarity (similar words)

among the code. Various files and documentations, as well as comments and identifiers within the source code can be investigated using LSI when trying to find the similarity between two programs and detect clones.

Marcus and Maletic have done an experiment on NCSA Mosaic as medium-size software. They conclude that although the presented method leads to satisfying results, it should be used along with other methods, such as methods based on structural information, to give the best results.

Basit and Jarzabek [6] have designed a tool prototyped named Clone Miner for detecting clones in a file, simple clone as they name it, and clones in different files, which they call structural clones or design-level similarities. Simple clones are detected in a process where source code is first tokenized and then similarities in token sequence are evaluated. After this, data mining techniques are used to find structural clones. This is done by investigating the pattern of co-occurring simple clones in different files. Basit and Jarzabek argue that according to their experiments, their method is capable of successfully detecting design-level similarities, which allows developers to understand and reuse software components affectively. The approach is also claimed to be the first one employing data mining techniques to detect design-level similarities, and being capable of scaling up to handle big systems.

3.7.2 Program Understanding Based on Constraints Satisfaction

Steven Woods and Qiang Yang [70, 72] claim that all program understanding is essentially an NP-hard problem. They present a new approach, based on knowledge-based program understanding, to the problem by considering the problem as a Constraint Satisfaction Problem (CSP) and trying to solve it through a heuristic approach. In their view, a program could be seen as a set of different components that are related to each other through various constraints. Knowledge-based program understanding is then the process of explaining and satisfying these constraints using plans. Plans can be viewed as domain values that explain program components. There are two different kinds of constraints: knowledge constraints and structural constraints. Knowledge constraints describe how different plans can be joined to form a larger plan, and structural constraints describe the relationship between different programs components [72]. Their solution to the CSP problem is briefly as follows. First an intermediate representation from the source code is made, including control flow and data flow. After identifying program blocks, each block forms a variable of CSP. Constraints between variables of CSP are derived from the relationship in the intermediate representation of the source code and from the program plan library. Actual program understanding comes from the solution of the CSP, which is an assignment of one plan from the program plan library to each variable in a way that all constraints are satisfied.

Steven Woods and Qiang Yang argue that their approach is usable and capable of producing an understanding of larger programs, while some earlier plan recognition approaches have been applied mostly to *toy domains*, as they put it, meaning that the knowledge base and the search domain used in those approaches have been small and limited. Viewing the problem as CSP provides us with the possibility to apply all well-known CSP algorithms to the problem.

3.7.3 Task Oriented Program Understanding

Erdem et al. [20] present a different approach in program understanding which they call *Task Oriented Program Understanding*. They argue that understanding a program has different aspects, and that previous approaches do not take one important aspect into consideration: the user. Their approach focuses on the user's task, experience and expertise. They believe that investigating users' questions about a particular task and forming a task model makes it possible to provide a user by an explanation more effectively. Erdem et al. present a theoretical study on users' behavior when they are given a certain task to solve. Users' questions about the given task are studied and the similarities or differences between the questions are evaluated. The results of these task studies are used for producing a task taxonomy. This task taxonomy can then be used, for example, to determine the user goal, and thus to produce an explanation tailored to the user. Erdem et al. do not, however, explain in a concrete manner or by using an example, that how a program can really be understood using their approach.

3.7.4 Data-centered Program Understanding

Data-Centered Program Understanding, presented by Joiner et al. [34], starts the process of program understanding by analysing the data flow of the program, in contrast to other approaches, e.g., those used in reverse engineering, that analyze the control structure of the program first. Joiner et al. justify their approach by the fact that by investigating variables and interrelationship between them in a program a programmer can gain a quick understanding about that program. On the other hand, variable names are often very informative as well. For example, variable name MIN often suggests that it holds a smallest element of some set. In addition to this, an important question in maintenance tasks, i.e., what other parts of program will be affected if the value of some variable is changed, can be answered by analyzing the relationship between variables. Based on their method, Joiner et al. have developed a program understanding tool environmental named DPUTE. As Joiner et al. describe, techniques used in DPUTE consist of dependence analysis, variable classification, variable slicing, program slicing and ripple effect analysis. In the following, these techniques are briefly explained.

Dependence analysis refers to the investigation on how the variables in the program affect each other. Data-centered approaches maintain a variable dependence model that can explain the relationships among the variables so that the way each variable affects another one is known.

In *variable classification* process, the variables are classified into different categories. The goal of variable classification is to make the process of program understanding easier, especially in large programs that may have thousands of variables. In DPUTE, there are eight variable classification categories: domain variables (variables related to the domain problem that are useful in understanding the program), program variables (variables used, e.g., to hold the intermediate computation results and error handling), local variables, global variables, input variables, output variables (variables that are related to output events), constant

variables (variables with fixed value) and control variables (variables used in predicates, i.e., variables that control the execution sequence of the program).

Program slicing is a program analysis technique that transforms a given program to a new one that includes only those parts (statements) of the original program that affect the value of a variable in the slicing criteria, ignoring all other parts. Slicing criteria is a particular place in the program where the variable of interest is located. Program slicing technique can be applied in different ways, one of them being forward slicing. Forward slicing technique discovers those parts (statements) of the program that are affected by a given variable. *Variable slicing*, on the other hand, is also a program analysis technique that produces a list of variables that affect the variables in the slicing criteria. The modules where these variables are located are also listed in the output produced by variable slicing technique. Forward variable slicing, respectively, finds all those variables that are affected by a given variable.

Ripple effect analysis means the process of discovering and fixing the undesirable changes that occur in a program as a side effect of particular change performed in that program. The version of ripple effect analysis used in data-centered approach is variable ripple effect analysis that, as the name suggests, is used for discovering and correcting the undesirable changes in variables resulted from a deliberate changing of the value of a variable in the program. Ripple effect analysis uses program slicing and forward slicing techniques, whereas, variable ripple effect analysis uses variable slicing and forward variable slicing techniques, respectively.

3.8 Dynamic Program Analysis

In dynamic program analysis, the program is executed by some test input, and the output is then investigated in order to understand the functionality of the program. For analyzing the program thoroughly and understanding its functionality comprehensively, lots of tests must be done.

Dynamic program analysis is often used in automatic assessment systems, where the accuracy of students' submissions is tested by running their program using some test input, and comparing its output with the expected one. In addition to this, some debugging systems, e.g., aforementioned Falosy [62] use Dynamic system analysis as well.

In the following, we present an overview of some automatic assessment systems. Since we are primarily interested in static program analysis, we keep this presentation very brief.

3.8.1 Automatic Assessment Systems

Ala-Mutka [2] has investigated various automatic assessment systems, that can be used by teaching staff at universities in students' submissions assessment tasks. Student's exercises could be divided into several categories [2, 13] including multiple choice, programming assignments, visual answers, text answers and peer assessment. Different automatic as-

sessment systems are capable of assessing different types of exercises. In the following, we discuss briefly some of the systems from the programming assignments category, since they include analyzing and recognition of source code submitted by students.

ASSYST

Developed at the University of Liverpool, ASSYST [31] is an automatic assessment system that grades the students' submissions based on five measurements: correctness, efficiency, complexity, style, and test data adequacy. It applies a dynamic analyzing method for evaluating the correctness of the code. First, the instructor writes a specification of a model output that the correct program should produce. After this, the student's submission is executed and its output is compared to the model output using the pattern matching approach. ASSYST uses Unix Lex and Yacc tools to examine whether the student-written program output conforms to the model output. ASSYST does static analysis to some extent as well: it analyzes program blocks and counts all statements in each block in order to measure the efficiency and the test data adequacy of the students' submission.

Ceilidh and CourseMarker

Ceilidh [61] is another automatic assessment system that like ASSYST, analyzes the students' submission dynamically. Like ASSYST, it runs the program with some test data, and then examines the output to determine whether the output matches the model output generated by model solution. For evaluation on complexity of the code, Ceilidh uses Unix C lint tool to perform static analysis on the code and to count the number of conditionals, loops, unused variables and so on.

CourseMarker (formerly known as CourseMaster) [28] is a successor of Ceilidh which is improved by new features like network support, window interface, better performance, better feedback system, maintainability and scalability. In addition, unlike Ceilidh which runs only on UNIX platform, CourseMarker is portable and runs on UNIX and Window platforms, since it is developed in Java.

Scheme-robo

Scheme-robo [58], a system for analyzing Scheme programs, is basically similar to previously mentioned other two systems: it does dynamic analysing on the code submitted by students to assess them. Scheme-robo simply executes the code using different test runs and examines the result. Scheme-robo, however, performs assessment based on the value that the student-written program returns, and not by comparing the output of the program to some model output. Scheme-robo performs static analysis, as well. For example, it examines the code to ensure that some particular keywords are not found in the submission. This is because if students are asked to implement a particular functionality, and the language has some built-in utility function to do that functionality, the students are not allowed to use that utility. Scheme-robo uses static analysis to convert the code into an abstract syntax tree,

as well. It then uses the abstract syntax tree to examine the program further, for instance, in plagiarism detection tasks.

Boss

Developed by Joy et al., Boss [36] is an online and platform-independent automatic assignment assessment tool that performs both dynamic and static analysis, e.g., in plagiarism detection. Boss evaluates students' submissions from several different point of views, for example, it uses some metrics to check programming style, e.g., using comments, checks the submissions to find plagiarized programs, checks for authentication, checks the submissions for correctness after which it marks each submission and gives feedback to the submitter. The correctness of students' submissions is tested using two different techniques: running the submission using some test data and then investigating the output file and using JUnit (when Java language is in question).

Web-CAT

Web-CAT [24] is an environment for software testing and automatic assignment assessment environment developed at the Department of Computer Science, Virginia Tech. Web-CAT is designed to encourage the students to use test driven development approach and test their own solution thoroughly. The idea is that instead of assessing the students' submissions based on some predefined test produced by the instructor, the submissions can be assessed by the students' own tests. Practically this means that the student's submission is tested more thoroughly than if it was tested only by the instructor's tests, because the student is motivated to test his submission thoroughly since the student's test completeness contributes to his final grade: the more the student's tests cover the code, the higher his score. In addition to test completeness, the student's final grade is computed from code correctness and test validity. As mentioned earlier, in addition to instructor's tests, code correctness is measured by the student's own test as well: the more tests the student's code can pass, the higher his score. Test validity refers to whether the student's tests are accurate, i.e., do they test the program in a right way. Web-CAT developer claims that the results of using Web-CAT have been very promising: the students' performance has been improved and the amount of bugs in students' submissions is reduced, students test their programs more actively and in more accurate manner and have more positive attitude toward testing, and finally, the students have adopted the test driven development approach.

3.9 Analysis and Conclusion

Here, we will present an analysis of previously discussed related work. We will evaluate each of the categories listed above in turn, and explain which of these approaches we could use and why.

3.9.1 Knowledge-based Approaches

Most of the program understanding approaches fall into this category. Knowledge-base program understanding approaches consider the problem of understanding a program as understanding what it does, i.e., its functionality. There are number of various methods developed based on knowledge-base program understanding approaches in a bid to improve previously released approaches regarding to effectiveness and scalability, among others, but all of them use the same idea. The idea in program knowledge-based program understanding is that the subject program is understood through mapping the program source code to the plans in plan library. If a part of the source code match with a plan, then we know what that part of source code does. This process is continued hierarchically in a top-down or bottom-up manner until the whole code is understood.

Strengths

Knowledge-based program understanding approaches are the first, and perhaps the most extensively researched and used approaches in program understanding field. As such, there have been a lot of improvements suggested to these approaches. In addition, various ways of using different heuristics have been presented and deemed useful in solving the problem.

A clear advantage of these approaches over others, say reverse-engineering approaches, is that in these approaches the process of program understanding is totally automated. This means that once a match between the code and the plans has been detected, the user can be provided with the information about what the subject program does. On the contrary, the process of understanding a program in reverse-engineering approaches for example, is not fully automated: a reverse-engineering based system provides the user with a set of high-level design documentations, diagrams and other information, but the final decision about what the subject program does, is left to be made by the user. In other word, knowledge-base program understanding approaches seem to be the only ones that focus directly on what a program actually does and attempt to solve the problem fully automatically. In this sense, knowledge-base program understanding approaches appear to be promising and are worth to be further researched.

Weaknesses

Perhaps the most serious limitation of knowledge-based program understanding approaches is the issue of scalability. In order to understand a piece of a source code, the corresponding plan must be found from the plan library. This means that for each piece of a code, there must be a plan in the plan library that recognizes it. This implies, in turn, that the more comprehensive program understanding tool is desired to be, the more plans must be added into the library. On the other hand, the more plans they are in the library, the more costly and inefficient the process of searching and matching will get. Since we do not know the domain of the program to be understood, we must add as much plans as we can into the plan library. This results in the fact that even for understanding a relatively small program

(100 lines of code or so), there must be hundreds of plans in the plan library to understand the program. That is the reason why the tools using these approaches are often referred to as being able to handle only "toy programs". Explicitly spoken, it seems somehow very difficult, if not impossible, to provide the plan library with all necessary plans that can guarantee a successful process of understanding a real-life program [52].

Complexity is another problem of the program understanding algorithms. Most of these algorithms are NP-complete in the worse case, and this comes from the fact that these algorithms must scale with the size of the subject program on one hand, and the size of the library plan on the other hand [52].

The other limitation of these approaches is the maintenance issue of the tools based on these approaches. As mentioned before, in order to extend the range of the programs that these tools can understand, new plans needs to be added to the plan library. But in some cases, adding new plans to the library is such a difficult and complex task that could not be carried out by those other than the developers of the tool in question. An example of these kind of tools is [53].

Another disadvantage of knowledge-based program understanding approaches is that there is a possibility that plans can not be successfully matched to the subject program, even though they both do the same thing. This can occur if plans are scattered across files, or if the subject program code is an idiosyncratic code, that is, it is not written with conventional and wildly-accepted programming style in mind. This is the case especially in legacy systems, since different maintainers with different programming style and skill have changed the code, resulting in a kind of spaghetti code. Since plans are designed to be matched against stereotypical codes that follow these conventional and well-recognized programming styles, they can not recognize the code that ignores those styles and is written in a somewhat unusual way [50, 52].

Most of the tools based on these approaches are language-dependent, i.e., they are not able to understand programs written in languages other than the language they are designed for. In the literature, this issue is known as the *generality* problem. Although some efforts have been made to improve these approaches in this regard [39], it is still considered as a serious limitation of the tools.

3.9.2 Using Fuzzy Reasoning in Program Understanding

These approaches use a plan library as their knowledge base, so they can be considered essentially the same as knowledge-based program understanding approaches. However, they use fuzzy reasoning in making the decision about the functionality of the program to be understood.

Strengths

In addition to the strengths explained for knowledge-based program understanding approaches, approaches based on fuzzy reasoning make use of fuzzy reasoning to make the

final decision about what the subject program does. This could result in a more accurate decision.

These approaches also present a method for improving the efficiency of the knowledge-based program understanding approaches. Instead of comparing the whole code with the whole plan library, that can be very inefficient especially if there are huge number of plans in the library, these approaches carry out the process of matching in two steps. In the first step, the coherent pieces of the code called chunks, that appear most likely to be successfully matched are selected, and only those plans that look similar to these chunks are selected to be matched. The second step includes the more detailed matching process of those plans and chunks that successfully matched in the first step. This method can result in saving a lot of time in the process of matching.

As these approaches share basically the same idea with knowledge-based program understanding approaches and, furthermore, attempt to improve the efficiency and accuracy, they can be regarded as more promising approaches in understanding a program from the functionality point of view.

Weaknesses

The weakness described for knowledge-based program understanding approaches can be regarded being valid here as well, since these approaches are essentially the same.

3.9.3 Program Similarity Evaluation Approaches

Program similarity evaluation techniques could be considered as methods to analyze structure and style of a program. Therefore, they do not directly solve the problem of understanding a program from the functionality point of view. As a result, these approaches should be evaluated in a different way using different criteria.

The two types of program similarity evaluation approaches, that is, attribute-counting approaches and structure-based approaches, are very different and thus, have different advantages and disadvantages. In the following, we present an evaluation on strengths and weaknesses of both approaches. The following evaluation is valid also for hybrid approaches that apply both techniques.

Strengths

Since attribute-counting approaches detect the similarity based on number of characteristics of the code, changing the name of those characteristics or other alike changes has no effect on the result. Some of the code characteristics used in these approaches can be used in our approaches in recognizing a program too, since these characteristics offer a way to distinguish a program. These techniques are easy to implement and if desired, the Halstead's measures based complexity of the subject code can also be evaluated while the similarity of two programs is examined. These approaches are efficient, as well.

Structure-based approaches are tolerant against changing the structure of the subject program. Recently, there have been presented fast string matching algorithms, e.g., [47] that make this approaches more efficient than they used to be.

Because methods using combination of these two approaches select the best features of these, they often seem to be more effective than those making use of just one.

These methods are very helpful in investigating the structure and style of a program. Researches on these methods continue (e.g., [47]), and these methods are very likely to become much more effective and efficient in the future.

Weaknesses

Attribute-counting approaches are criticized to fail to detect the similarity between two programs, if the structures of those programs are somehow changed. In addition, since different attribute-counting approaches use different characteristics, it is often the case that some of these approaches are less accurate or efficient than others. In these approaches, moreover, there are often some type of ad hoc methods involved in making the final decision about how similar two given program are. These ad hoc methods vary a lot and some of them are not the best possible.

There exist many different structure-based approaches too, and some of these approaches may suffer from efficiency issues. In addition, although string matching algorithms used in these approaches have been improved constantly, their accuracy and efficiency need to be further enhanced.

3.9.4 Reverse Engineering

Reverse engineering approaches try to understand a program in order to produce high-level design documents, improve or extend the functionality of the program, etc. These methods can be applied to solve various different types of problems. For example, they can be used to understand both the functionality and the structure of a program. Like other approaches, however, reverse engineering approaches have many pros and cons, as we describe in the following.

Strengths

Perhaps the most important benefit of reverse engineering approaches is that they include very powerful techniques that can be used in various fields and in many different ways. Reverse engineering tools allow developers to produce useful and understandable documentation at high-level abstraction from a system that otherwise could be very difficult to understand because of lacking of documentations. These techniques are very useful in maintenance activity and can be used to understand the subject system, saving a lot of time in the time-consuming task of reading the program code.

Reverse engineering approaches are one of the most extensively researched subjects in the field, and they are very likely to be developed further and fast.

Weaknesses

Reverse engineering techniques provide half-automated program understanding, leaving the other half to be decided by the user. These techniques do not tell, directly and clearly, what the subject program does, but rather provide the user with various high-level abstraction diagrams and specifications of the subject system, helping the user to make the final decision on the system's functionality. Therefore, instead of articulating the subject program's functionality, reverse engineering tools rely on user's cognitive abilities to do the final decision about what the subject program does. High-level documents and diagrams generated by reverse engineering tools are certainly very precious to maintainers in the process of understanding the subject program, but ultimately the user will have to decide what the subject program does using his or her experience and the problem domain knowledge. In other words, program understanding is a collaborative effort between the user and the tool, and the responsibility of understanding the program is left to the user.

3.9.5 Other Approaches

The other approaches on program understanding use basically one of the aforementioned approaches, so the strengths and weaknesses of those approaches are valid here as well. Program understanding based on constraints satisfaction, for example, uses plan library as its knowledge base, and suffers from the same issues already stated for knowledge-based program understanding approaches, although it makes an effort to improve the efficiency of those approaches.

Task Oriented Program Understanding approach does not seem to be extensively researched, and the authors do not actually present detailed information about how the process of program understanding can be carried out using this approach.

Data-Centered Program Understanding approach seems to be suitable mainly in maintenance work, but does not appear to offer a way for understanding a program automatically.

3.9.6 Dynamic Program Analysis

Since our method will be based on static program analysis, we will not discuss the strengths and weaknesses of dynamic program analysis here.

3.10 Summary

Here, we present a brief discussion about techniques that can be used in our problem from the techniques discussed above.

As describe above, many interesting methods have been developed in order to understand a program from different points of view. From all aforementioned techniques, however, techniques used in attribute-counting program similarity evaluation approaches seem to be most useful for our problem. The reason is that our approach is based on the code characteristics: we want to examine whether it is possible to tell what an algorithm looks like or appears to be, by investigating its characteristics, e.g., variables. Attribute-counting techniques investigate precisely the same thing: distinguishing characteristics of a code. We can use, as an example, Halstead's measures in our problem.

If it seems useful, we can also make use of knowledge-based program understanding approaches: if necessary, we can use small plans to make sure about the functionality of a piece of the code to make sure that our evaluation about type of algorithm is right.

The other techniques do not appear to be useful in our problem at this phase, although we will keep them in mind during the design and implementation of our method.

Chapter 4

Analysis

As discussed in Chapter 2, by understanding an algorithm we mean recognizing the category to which the algorithm belongs to. More precisely, our objective is to automatically conclude the category that a sorting algorithm falls into.

In this chapter we present an analysis of our approach for recognizing algorithms. We will describe all distinguishing characteristics that we have used to recognize different sorting algorithms. We will, however, first describe briefly the concept of roles of variables as it plays an important role in our method. Other concepts and characteristics are presented after this. We close the chapter by describing the method we will use to achieve our objective.

4.1 Roles of Variables

As mentioned before, the concept of role of variables [59] is first introduced by Sajaniemi. The idea behind roles of variables is that each variable used in a program plays a particular role that is related to the way it is used. 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, roles of variables are part of programming knowledge that have remained tacit. Experts and experienced programmers have always been aware of existing of variable roles and have used it, 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, explaining to them the different ways in which variables can be used in a program. Moreover, the concept can offer an effective and unique tool to analyze a program with different purposes.

Currently, there are 11 different roles recognized for variables used in novice-level programs. In the following we will present these roles and describe each of them briefly. The roles are listed loosely in order of frequency of appearance in sorting algorithms that were analyzed in our study, with the most frequent roles first.

Stepper

A stepper role is assigned to a variable that systematically goes through a succession of values, e.g., values stored in an array. A loop counter, i.e., a variable of integer type used to control the iterations of a loop is a typical example of a stepper.

Temporary

Variable that holds a value for a short period of time appears in temporary role. A typical example of the temporary role is a variable used in a swap operation.

Organizer

A data structure holding values that can be rearranged is a typical example of organizer role. For example, an array to be sorted in sorting algorithms has an organizer role.

Fixed value

As the name suggests, a variable has a role of fixed value if it 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.

Most-wanted holder

A variable is said to have a most-wanted holder role if it holds a most desirable value that is found so far. For example, if we find a minimum or maximum from a set of values and assign it to a variable, that variable would have a most-wanted holder role.

Most-recent holder

Most-recent holder role is given to a variable that holds the latest value from a set of values that is being gone through. Moreover, a variable that holds the latest input value is a most-recent holder.

One-way flag

One-way flag is a role assigned to a variable that can have only two values and once its value has been changed, it cannot get its previous value back again. This role usually appears in a program to indicate, for example, whether an error has been occurred.

Follower

Follower is a role indicating a variable that always gets its value from another variable. In other words, its new values are determined by the old values of another variable.

Gatherer

A variable is said to play a gatherer role if it collects the values of other variables into itself. A typical example is a variable that holds the sum of other variables in a loop and as a result, its value is changed after each execution of the loop. When the loop is terminated, the variable holds a total of other variables' values.

Container

A data structure into which elements can be added or from which elements can be removed if necessary, has a role of container. For example, all Java data structures that implement Collection interface can be considered having container role.

Walker

As the name suggests, the walker role can be assigned to a variable that is used for going through or traversing a data structure.

There are several other aspects associated with the concept of roles of variables that are not presented here. More information can be found from [59].

4.2 Characteristics of Sorting Algorithms

At the beginning of our study, we analyzed 11 common sorting algorithms. These algorithms and their characteristics are shown in Table 4.1. These versions were selected from the course material on Data Structure and Algorithms lectured at the Department of Computer Science and Engineering at the Helsinki University of Technology. From each algorithm, eight characteristics are investigated. The name of each sorting algorithm is shown in the first column of the table. The last column of the table depicts the structure of the algorithms. This helps us to get a quick information about loops and blocks of each algorithm. The dashed line in this column indicates the end of one method and the beginning of another one. The characteristics of the algorithms are shown in the columns between the first and the last columns. It must be noted that these algorithms are only one randomly selected example of each corresponding type of sorting algorithm.

As can be seen from the table, the characteristics of these algorithms seem to be distinguishing enough to make us able to differentiate among them. Most of the characteristics are different among algorithms, and if some characteristics happen to be the same or close,

	Number of variables	Roles of variables	Number of loops	Nested / sequential loop	Number of blocks	Nested / sequential blocks	Recursive	In-place	Depiction of nested / sequential loops and blocks
Insertion sort	3	2 Stepper 1 Temporary	2	Nested	2	Nested	No	Yes	for while
Selection sort	4	2 Stepper 1 Temporary 1 Most-wanted holder	2	Nested	3	Nested	No	Yes	for for if
Quicksort	6	2 Stepper 2 Walker 1 Fixed value 1 Temporary	3	2 Nested 2 Sequential	5	2 Nested 3 Sequential	Yes	Yes	do while while if
Bubble sort	3	2 Stepper 1 Temporary	2	Nested	3	Nested	No	Yes	for for if
Mergesort	6	3 Stepper 2 Walker 1 Fixed value	3	Sequential	7	3 Nested 4 Sequential	Yes	No	if for for for if else
Shell sort	6	2 Walker 2 Stepper 1 Temporary 1 Most-recent holder	3	Nested	3	Nested	No	Yes	for for while
Heap sort	9	3 Stepper 1 Walker 3 Fixed value 2 Temporary	3	Sequential	7	4 Nested 3 Sequential	No	Yes	for while ---------- while if if
Straight Radix sort	7	3 Walker 2 Stepper 2 Fixed value	5	Sequential	8	Sequential	No	No	if if 5 for
Radix Exchange	6	2 Stepper 3 Walker 1 Temporary	3	2 Nested 2 Sequential	6	3 Nested 3 Sequential	Yes	Yes	if do while while if
BinQuickSort	5	3 Walker 2 Stepper	3	2 Nested 2 Sequential	6	3 Nested 3 Sequential	Yes	Yes	if while while while if
Quick Select	8	2 Walker 3 Fixed value 2 Stepper 1 Temporary	3	2 Nested 2 Sequential	5	2 Nested 3 Sequential	Yes	Yes	do while while if

Table 4.1: The common characteristics of sorting algorithms

the collection of all characteristics shown in the table is unique for each algorithm making that algorithm a distinguishable one.

Encouraged by this fact, we analyzed different versions of five sorting algorithm to see how much different versions of the same algorithm type differ from each other. These five sorting algorithms were Selection sort, Insertion sort, Bubble sort, Quicksort and Mergesort. These versions were collected from randomly selected textbooks and course material on Data Structure and Algorithms. These five sorting algorithms were analyzed based on the same characteristics shown in Table 4.1, but in order to keep this thesis short, the characteristics

of different versions of these sorting algorithms are not shown here. Instead, we will present them in charts in Chapter 6. As somehow expected, different versions of the same sorting algorithm were much more similar than different sorting algorithms. This means that if we can recognize a particular type of a sorting algorithm from the other types, then we can be quite sure that different versions of that particular algorithm are similar enough to be recognized from the other types in the same way.

As expected, the more complex an algorithm is, the more different ways there are to implement it. This means that different versions of a more complex algorithm like Quicksort are more likely to differ from each other than different versions of a simpler sorting algorithm like Insertion sort. For example, the Quicksort partitioning might be implemented in a separate method in one version, while in another version it is located in the same method with other statements. This causes differences in the number of variables and blocks in different implementations. This kind of differences should be eliminated in order to get a right statistics about different versions of the algorithms. To get rid of this, we performed inline expansion before analyzing the algorithms.

It should be noted that since our goal is to perform static program analysis, we are only interested in the characteristics of the sorting algorithms that can help us to distinguish them without the need of running them, i.e., the static characteristics of algorithms. Sorting algorithms have also dynamic characteristics, such as stability of algorithm, that can be useful when trying to distinguish them. But because those kinds of characteristics are related to the execution of the algorithm, we do not take them into consideration.

In the following, we explain each of these characteristics shown in Table 4.1. In addition to these characteristics, we also describe two more characteristics that we found distinguishing and useful in recognizing sorting algorithms, namely variable dependencies and the information about whether a loop is incrementing or decrementing.

Number of variables

As the name suggests, this refers to the total number of variables that are used in the algorithm.

Roles of variables

The role of variables is a key factor used as a distinguishing characteristic in our approach. The concept of roles of variables and the classification of different roles were described in the previous section.

Number of loops

This column indicates the amount of loops used in the algorithm is question. A loop can be a *for* loop, a *while* loop or a *do while* loop. Other types of loops are not analyzed in this work.

Nested / sequential loop

This column indicates whether the loops used in an algorithm are nested or non-nested. It appears difficult express the nested / sequential relation among loops in an unambiguous manner using numbers, since the information about each loop's position with regard to others cannot be included in numbers. For example, let us suppose that there are total of three loops, two of which are sequential with regard to each other, and both are located inside the third loop. How this can be expressed in numbers? We can say that there are 2 nested and 2 sequential loops. But this makes the total number of loops four. As we will present later in Chapter 5, the system is designed so that each loop has an unique ID and an unique upper loop ID. This makes it easier to recognize nested and sequential loops and distinguish between them.

Number of blocks

This column shows the number of blocks in the algorithm. A block refers to a sequence of statements wrapped in curly braces. A block can be a method or a control structure (loops and conditionals).

Nested / sequential blocks

This column shows the number of nested and non-nested blocks in the algorithm. The same problem regarding ambiguity appears in definition of nested / sequential blocks, as it was the case with loops. In implementation, this is dealt with in the same way as described for loops.

Recursive

This column contains the information about whether the algorithm is recursive or not. For example, as Table 4.1 shows, Quicksort algorithm analyzed in this study is a recursive algorithm, while Insertion sort algorithm is not.

In-place

This column indicates whether the algorithm needs an auxiliary array to carry out the sorting. In other words, it indicates the need of extra memory. For example, as can be seen from the Table 4.1, Mergesort analyzed in this study is not an In-place algorithm, while Quicksort is.

Variable dependencies

For each variable, direct and indirect dependencies on other variables are examined. If variable X gets its value directly from variable Y, then X is said to be directly dependent

on Y. Moreover, if there is a third variable Z on which Y is dependent (either directly or indirectly), then X is indirectly dependent on Z. A variable may have both a direct and an indirect dependency on another one.

Incrementing / decrementing loop

We also investigated loop counters in the algorithms to see whether a loop is an incrementing or decrementing loop. If a loop counter's value increases after each iteration, that loop is said to be an incrementing loop and, in the same way, if a loop counter's value decreases after each iteration, the loop is said to be a decrementing loop. As we will describe in Chapter 6, this information is a valuable factor in recognizing some sorting algorithm.

4.3 Other Characteristics

In addition to the characteristics described in the previous section, we also use the following characteristics: number of operators, number of operands, number of unique operators, number of unique operands, program length, program vocabulary size, number of assignment statements, line of code and McCabe cyclomatic complexity. These are Halstead's parameters and other characteristics used for evaluating program similarity as described in Chapter 3.

4.4 The Method

Our approach in recognizing algorithms is based on investigating the characteristics of them. By computing the distinguishing characteristics of an algorithm, we can compare these characteristics with those collected from already recognized algorithms and conclude if the algorithm falls into the same category.

The aforementioned characteristics can be divided into the following two types: numerical characteristics and descriptive characteristics. Numerical characteristics are those that can be expressed as positive integers, e.g., the number of loops, or the number of operators. Descriptive characteristics, on the other hand, cannot be expressed in numbers. The descriptive characteristics are the following: *Recursive*, *In-place* and *Roles of variables*. All characteristics other than these three belong to the numerical characteristics.

The method we will use is as follows. In order to carry out the recognition, the characteristics of the recognizable algorithm, both numerical and descriptive, are computed automatically by the Analyzer. The only exception is the roles of variables which, as will be described in the next chapter, is manually evaluated since the current role analyzer integrated to VILLE do not generate the roles accurately enough. These characteristics are then compared to the characteristics of different sorting algorithms that exist in the knowledge base of the Analyzer. By performing this comparison, the Analyzer is able to make the decision on whether the recognizable algorithm belongs to one of the categories existing

in its knowledge base. In other words, the recognition is based on the calculation of the frequency of occurrence of the numerical characteristics in the algorithm on one hand, and the investigation on the descriptive characteristics on the other hand. We will describe this in more detail in Chapter 6.

Chapter 5

Design and Implementation

In this chapter, an overview of the main features of the system is presented. In the first section, we present the architecture of the system. Implementation related issues, description of the classes and the detail structure of the database are presented in the second section. We close the chapter by giving an example of the dataflow of the Analyzer and discussing its limitations.

5.1 The Architecture

The Analyzer is built on top of VILLE, a visual learning tool. This decision was made based on the fact that the Analyzer can use the output of the VILLE. This results in saving time and effort, since there is no need to do all from the beginning. As we will discuss in the next subsection, the Analyzer uses the set of the indexes generated by VILLE during interpretation of a program. Using these indexes, the Analyzer knows what each line of the program is about. The second reason for building the Analyzer on top of VILLE is that the Analyzer can extend the functionality of VILLE in a logical way: VILLE can be used by an instructor in teaching programming skills; the Analyzer can also be use by him, for example, for verifying the student submissions.

Figure 5.1 shows the system's architecture including the architecture of VILLE and the Analyzer. The part below the dash line labeled as "Interface" is the architecture of VILLE, and the part above it is the class diagram of the Analyzer. In the following, we first explain VILLE, its architecture and functionality. The architecture of the Analyzer is described after this.

5.1.1 VILLE

VILLE is a language-independent program visualization tool that is intended to be used by a teacher in teaching programming skills, and to help students to understand the fundamental concepts of programs like variables, arrays, loops, conditionals and recursion, and

to learn the basic programming skills. The VILLE architecture, partly based on technical documentation written by its developers, is described in the following.

Broadly speaking, VILLE consists of three parts: the program Interpreter and the Visualizer. The program Interpreter can be further divided into three modules which are, as shown in Figure 5.1, as follows: the program interpreter, the methods determiner and the program executer. In addition, VILLE has a Role Analyzer integrated to it as a separate module. We will not explain the Role Analyzer in more details, as it is not a direct part of VILLE and has not been investigated in our work. In the following, these parts are explained in turn. A class diagram for VILLE is not presented, as it is not a part of this work.

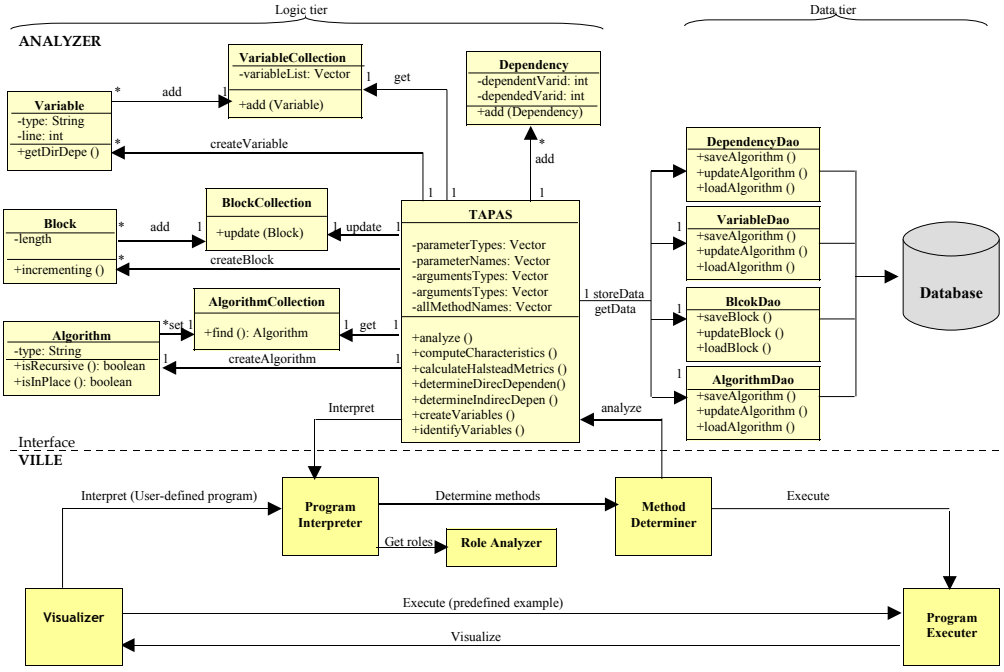

Figure 5.1: The architecture of the system. The part below the line labelled by "Interface" describes the architecture of VILLE, and the part above it describes the architecture of the Analyzer

The Program Interpreter

The program Interpreter is the core of VILLE, which performs the following tasks.

Interpreting the program The first task the Interpreter performs is to read a Java program in and interpret it to the other supported languages.

VILLE interprets Java to other language in the following way. Each supported language to which a Java program can be interpreted has its own *syntax file* that includes the most essential commands and control structures of that language. The syntax file can be regarded as the knowledge base of VILLE. Each line of syntax file has a unique number called *index*. The lines of each syntax file are arranged in the same order resulting in the fact that if two lines of two syntax files have the same index, they also have the same meaning. When a Java program is read in, each line in the input program is examined to find a match with a line from the syntax file. This matching process is carried out using keywords. When a match is found the index of the matched line from the syntax file is stored. Now when VILLE knows what the line of the input program means, it can interpret it to any other language simply by taking the corresponding line from the syntax file of the target language according to the stored index. Once the corresponding line in the target language is found, the parameters stored from the source language are assigned to that line in the target language.

In VILLE, a user can define a new programming language and make the tool show the execution of a program in that particular language. It can be done using the syntax editor, where the user writes the syntax of his own language following the same order and meaning of the indexes that appear on the left side of the syntax editor where Java syntaxes are located. Each line of the new language must be written on the right side of the same line where the corresponding Java command with the same semantic is located. Since VILLE knows the meaning of that particular line from the index of the line, it can understand the new user-defined languages. This is the meaning of language-independency of VILLE. Using this feature, any kind of language can be defined and be used to demonstrate the execution of a program, within the restriction of VILLE in recognizing structural and syntactical features.

Determining methods After the interpretation is performed, the Interpreter determines the methods of the input program. The following information about all methods in a program is stored: name, type, starting line and ending line. This information is needed in the next task, when the Interpreter executes the program.

Executing the program In this phase the program is executed. This is the most complex part of the Interpreter, since it involves processing of the control structures, program blocks, method invocations and changing of variables' values. A program is executed using the aforementioned indexes that tell the Interpreter what command is in question. The execution starts from the main method and ends at the last line of it. All blocks and method invocations within the main method are executed recursively.

As the output of this phase, a series of *execution events* are created. The Visualizer uses these execution events to show the execution of the code line by line.

The Visualizer

The Visualizer is an applet that, as its name implies, visualizes the execution of the program. The state, value and roles of the variables are visualized during the execution of the program. The Visualizer also shows the source code highlighting the line being executed and the previously executed line in different colors. The outputs of the program are also displayed. It is possible for a user to view the code execution in two different languages at the same time. The Visualizer is also able to visualize the execution of different methods using a call stack. This feature especially helps novice programmers to understand how recursion works. In addition, an explanation in natural language is provided for each line that is under execution. These explanations are created using indexes at the same time when an input program's lines are matched against syntax file, as was described before.

More information about VILLE can be found from the system paper of VILLE [48].

5.1.2 Limitations of VILLE

VILLE is intended to be used by novices, and as such, it does not support all structures and features of Java language. Despite this, building the Analyzer on the top of VILLE is well justified by the fact that the output of VILLE is a valuable input for the Analyzer. Using VILLE, the Analyzer does not need to interpret Java itself. The Analyzer uses the set of indexes generated by VILLE indicating what command is in question. This saved us from the laborious task of building a Java interpreter for the Analyzer.

The version of VILLE, on top of which the Analyzer is built, has many limitations. Some of these limitations and their impact on the Analyzer and its Implementation are discussed in the following. It should be noted that these are only some examples of VILLE's limitations, discussing all limitations is beyond the scope of this work.

Perhaps the main limitation of VILLE is that it can only interpret the structures that are located in its syntax files. The syntax files are by no means complete. There are many features and structures of Java language missing from the syntax files. For example, the switch statement is not supported. As a result, the Analyzer has no mechanism for processing the switch statement either. VILLE crashes while processing the switch statement and other structures it does not recognize. Moreover, among those structures recognized by VILLE, only some particular kinds are supported. For example, only the basic form of the for statement is supported. In other words, all three parts of a for statement, namely, the initialization part, the expression part and the variable value updating part must be presented as follows: *for(ForInit; Expression; ForUpdate)*. VILLE does not recognize the enhanced for statement, or the for statement where the initialization part or the updating part are not presented. In addition, there is no mechanism in VILLE to process objects.

In addition to the aforementioned limitations, there are some legal language syntactical features of Java that are not supported in VILLE and cause it to crash. For example, comments written at the same line in front of the for statement causes VILLE to crash. As another example, one or more white spaces between the definition of a class and left curly bracket cause problems, as well. During the implementation and testing phase of the Analyzer at

least 15 of this kinds of limitations were encountered.

These limitations and shortages made the testing phase very difficult and laborious, since extra effort was needed to provide a "clean" version of source code to VILLE so that the execution could continue to the Analyzer, without crashing on the way before it. Although the changes needed extra effort, the most time-consuming issue was to understand what VILLE required to be changed in the first place. The reason is that there are no descriptive error messages associated with these crashes to help the user to find the reason for crashes. It was not easy to find the part that needed to be changed in order to make VILLE happy. Thus, the "error" in the source code had to be found in a trial and error manner: change the most suspicious part, run the application again and hope for the best! It should be noted, however, that these changes were made carefully so that they do not significantly change the numerical characteristics of the code, such as its length.

From the Analyzer point of view, however, the main problem with VILLE is that the role analyzer integrated into VILLE does not work properly. It does not produce accurate information about roles. For instance, the loop counter of the inner loop in Bubble sort has clearly a stepper role. VILLE believes, however, that it should have a role of temporary holder. Likewise, the variable used in the swap operation in Bubble sort has clearly a temporary role, while in VILLE's opinion, it appears in most-wanted holder role. As we will discuss in Chapter 6, the roles of variables plays an essential role in our recognition process. As a concrete example, the appearance of a variable in the most-wanted holder role is a distinguishing factor that helps us to differentiate Selection sort form Insertion sort and Bubble sort (see the decision tree presented in Chapter 6). VILLE, however, returns a role of *Converter* for that variable which is not even presented as a role in [59]. Therefore, we could not afford to trust the roles produced by VILLE, and had to analyze all roles manually.

It is also noteworthy that not all indexes generated by VILLE are precise and unambiguous. Some indexes should be examined by the Analyzer carefully to determine what their exact meaning is. This is because some indexes can have more than one meaning from the Analyzer point of view. For instance, the assignment statement is indicated by an index of 85, regardless of its right side. The right side of the assignment statement should be carefully investigated since it makes a big different to the Analyzer whether the right side is, for example, a method call or a simple variable. Namely, if it is a method call, the Analyzer should perform in-lining (if it is not already done to that invoked method), whereas if it is a simple variable, it would be enough to process the dependency, as an example.

In this work, no attempt was made to improve VILLE or to extend its functionalities. It was not within the scope of this work, nor did the author have the authorization to change the VILLE. Instead, a list of these limitations and shortages was collected to be reported to VILLE's developers. However, the small changes were necessary to make to the interface between VILLE and the Analyzer to make them work together. In addition to these changes, a method was added to VILLE to make the system support method overloading. Before making this change, VILLE returned only the information of the method with the same name that appeared first in the source code, although information of all methods with the same name was correctly collected. With the change made in this regard, VILLE can now return the information of all methods with the same name according to the location of the

method in the data structure were all methods are stored.

These limitations and shortages can partly be related to the fact that VILLE is designed and intended to be used by novices, at the very beginning stages of learning programming skills. Positive feedback from the students and considerable improvement in their programming skills that have resulted from using VILLE keep the developers motivated to improve it. Thus, VILLE is under constant development.

5.1.3 The Analyzer

A brief overview on application architecture and classes is presented in the following. More explanation on classes is presented in the next section.

As was described previously, the Interpreter of VILLE performs mainly the following three tasks: interpreting the program, determining methods and executing the program. The Analyzer is, as can be seen from Figure 5.1, located after the second task of the Interpreter, i.e., the determining methods task. Therefore, the output of VILLE that goes to the Analyzer as an input, is a set of indexes that determine what command is in question. In addition, the aforementioned information about the methods of a program is available to the Analyzer. This is the best the Analyzer way can make us of VILLE. The execution part of the Interpreter involves computing the value of the variables and the other activities that are not useful for the Analyzer.

The Analyzer was designed following the three-tier application model. This is the most widespread use of multi-tier architecture that makes the maintenance of the system easier. The three-tier application architecture consists of the following tiers: *the presentation* tier, *the logic or business tier* and *the data tier*. The Analyzer works as a text-based interface that takes the subject program file as an input from the command line and outputs the textual results. Therefore, the presentation tier and GUI have no significant part in the system. As a result, this tier is omitted from Figure 5.1. Moreover, the logic tier and the data tier are annotated in the figure as two clear parts.

The logic tier contains the core of the system where actual analyzing of a program takes place. The logic tier consists of the following classes: *Algorithm*, *Block*, *Variable*, *Dependency*, *AlgorithmCollection*, *BlockCollection*, *VariableCollection* and *TAPAS*. The classes AlgorithmCollection, BlockCollection and VariableCollection do not include other functionality than the processing of a set of the objects that they hold. These classes contain a data structure to store the corresponding objects, e.g., blocks, and the methods needed for manipulating them, i.e., adding, finding and updating. The system could have been designed and implemented without these classes as well: Algorithm, Block, Variable and Dependency could have included a data structure and necessary methods to manipulate the object inside them. The way the system is implemented, however, makes the architecture more understandable and the maintenance easier, since these four classes contain only the essential methods for carrying out the analyzing with routine methods for manipulating the collection of objects located in their own separate classes.

Moreover, using the Algorithm, Block, Variable and Dependency classes can be justified

by the fact that these are clearly different and independent concepts and therefore, could be defined in different classes. One alternative solution in regard with the blocks could have been to define a superclass and inherit the different types of blocks, e.g. loops and conditional structures from it. This was not, however, considered necessary, because the Block is a relatively small and well-maintainable class in its present form. In addition, VILLE has also Variable class, and it was also considered to inherit the Variable class used in the Analyzer from it. This was not deemed necessary either, since VILLE's Variable class is so different that there is not much to be reused in the Analyzer's Variable class. The classes belonging to the logic tier are located in a package named *tapas_domain*.

The data tier consists of the database and the data access classes used to store and retrieve the information. The data access classes are the following: *AlgorithmDao*, *BlockDao*, *VariableDao* and *DependencyDao*. These classes are located in a separate package named *tapas_persistence*. By locating these classes in a separate package and defining data access objects that are used to create connection with the database, obtain and store data, it is easier to delete or add new columns to the database when necessary. A separate data tier makes the structure of the system clearer and improves scalability.

The data access classes use the objects of the classes from the logic tier in their *load()* and *loadAll()* methods (the latter method is not shown in Figure 5.1) to store the information retrieved from the database into them. This relationship is not shown in the figure.

It should be noted that, in addition to the description of the relationship shown in Figure 5.1 among the TAPAS and the other classes, the TAPAS class uses the other classes in many other ways as well. This is not, however, shown in the figure. For example, the TAPAS uses the Algorithm class to create an Algorithm object, to determining whether the algorithm is a recursive and in-place, among others. While in the figure, the relationship is shown only as "createAlgorithm".

We decided to use a database for the following reason. In order to perform the algorithm recognition task, the Analyzer should have access to such data, which it can use to compare the characteristics of the recognizable algorithm to those of the algorithms known to it. Based on the result of this comparison, the Analyzer can then decide whether the algorithm can be recognized as one of those it knows. Algorithms that are manually or automatically recognized and verified as some particular type in the beginning, can be stored in the database. Each time the Analyzer performs the recognition, it can retrieve the characteristics of algorithms from the database and use it in the process of recognition. The database is necessary in order to store the characteristics of different types of algorithms that the Analyzer recognizes and, thus, to extend the knowledge base of the Analyzer.

Using a database in the system can also be justified from the instructor point of view. If the system does not confirm an algorithm being a type what it is expected to be, an instructor can look at the source code and its characteristics from the database and verify the Analyzer's conclusion. Alternatively, if the instructor believes that the algorithm is of the expected type, he can examine the characteristics of the algorithm from the database to see whether the reason for rejecting the algorithm is related to the numerical characteristics of the algorithm or to the descriptive characteristics of the algorithm. If the reason is that one or more numerical characteristics of the algorithm do not fit within the minimum and

maximum values located in the database, the instructor can correct the type of the algorithm in the database. The knowledge of the system about that particular algorithm can be extended this way resulting in a more accurate decision by the Analyzer in the future: the same or similar algorithms would not be rejected for the same reason next time, since the minimum and maximum values existing in the database are adjusted. If, on the other hand, the reason is related to the descriptive characteristics, it should be investigated in more detail with regard to the decision tree presented in Chapter 6. Changes of this kind require the modification of the logic of the Analyzer. This is described in the next chapter in more detail.

It should be noted that not all the attributes and methods of the classes are depicted in Figure 5.1. We will discuss some important attributes and methods of the classes in the next sections.

5.2 Implementation

In this section, a more detailed overview on classes is presented, following by a discussion on the structure of the database and descriptions of it columns. At the end of the chapter, the system and its functionality is described as a whole using an example of the dataflow.

5.2.1 The Classes

The classes used in the system are presented in the following. Some of the attributes of the classes are described in the next subsection where the database and its columns are discussed (see Table 5.3). We do not explain those attributes here again, but instead mention those attributes that are not stored in the database and discuss some functionality of the classes.

TAPAS This class contains the main-method, as well as many other attributes and methods that carry out the analyzing. One of these is the recursive method that traverses through the code determining new variables, statements and blocks and calculating other characteristics. The efficiency can be improved by carrying out all these tasks at the same time, rather than going the code through separately in order to compute each of these characteristics in turn. Every time a method invocation is encountered, the method performs in-lining by calling itself with appropriate arguments in order to process the invoked method's body.

Most of the functionalities of the Analyzer are located in the TAPAS class. In addition to the recursive method, it contains many other methods to perform different tasks and many attributes to store the information. Some of these class members are presented in Figure 5.1, and to keep the scope of this thesis reasonable, we do not discuss all these class members here.

Algorithm In addition to the columns described in Table 5.3, this class includes some other attributes used in the application that are not stored in the database. For instance,

the program length and the program vocabulary size which are used as algorithm characteristics, are not stored in the database, but are calculated in the following way: the program Length is calculated by summing the total number of operators and the total number of operands, and the program vocabulary size is calculated by summing the number of unique operators and the number of unique operands. Moreover, the Algorithm class includes methods for computing other properties of an algorithm, e.g., whether an algorithm is a recursive one, or whether it is an in-place algorithm.

The method that is used to determine whether an algorithm is recursive works as follows. Each method of an algorithm is investigated in the order the methods appear in the program. If a method invocation with the same amount of arguments and the same types is found from the body of a method, the algorithm is labelled as a recursive algorithm and its execution is terminated.

The mechanism of determining whether an algorithm is an in-place algorithm is as follows. All methods of the algorithm are investigated in the order of their appearance in the algorithm, in the same way as was described previously. When an array definition and initialization is encountered within a method, it can indicate that an auxiliary array is used. But the Analyzer continues its investigations to see whether the auxiliary array is written and read as well. In other words, if the array that is defined within the method is both written and read, the Analyzer can conclude that the algorithm is not an in-place algorithm.

Block The attributes of this class were also introduced in the next subsection. In addition to those attributes, this class includes some other attributes, that are not stored in the database. For instance, the starting and ending lines of a block are also parts of the attributes of this class. These two attributes are used for determining the block length.

The Block class also includes many methods for processing block-related issues, among which the method for determining whether a loop is incrementing or decrementing. This is carried out in the following way. First, the loop counter is recognized. Then the body of the loop is investigated to see whether it contains an assignment statement where the value of the counter is incremented. If no such statement is found, the false value is returned indicating that the loop is decrementing. Notice that it is only a brief description of the functionality of this method. The method considers many detail issues. As an example, *for*, *while* and *do while* loops are processed differently. As another example, the possibility of changing the value of the loop counter as an index of array is also considered.

Variable As is the case with other classes, the Variable class includes attributes that are described in Table 5.3. However, here as well, there are some other attributes that are not stored in the database. Each variable has an attribute named *line*, which refers to the line number the variable was first declared. Each variable has also an attribute named *level* that is used to indicate variable scope. Moreover, each variable has two data structures: one to store its direct dependencies and one to store its indirect dependencies.

As a variable declaration or definition is encountered in the TAPAS class, a new vari-

able object is created storing its information such as the name, the type, the level, the line and the block id within which it appears. If the line included only the variable declaration, then after the variable is created, the next line is processed. If, on the other hand, the variable is also assigned a value in the same line, then the right side of the assignment statements is investigated and the appropriate action is taken accordingly. For example, if the right side of the assignment statement is a method invocation, the next step is to process the invoked method. If, alternatively the right side of the assignment statement is another variable, then the variable calls its corresponding method to add the right side variable to the data structure that stores its direct dependencies.

Dependency The attributes of the Dependency class are described in the next subsection.

In addition to the other class members, this class has a data structure to store all dependencies among variables of an algorithms and a method for adding a new dependency into it.

As can be seen from Figure 5.1, in addition to the aforementioned classes, the Analyzer includes classes AlgorithmCollection, BlockCollection and VariableCollection, that are used, as their names suggest, to hold a set of algorithms, block and variables, respectively. These classes have a data structure to store these objects and the necessary methods to manipulate them. Using these methods, it is possible, for instance, add an object to these collections, find an object by it name, type and scope, update an object, get all objects located in collections and examine whether these collections are empty.

For the database operation purposes the following classes are used: AlgorithmDao, BlockDao, VariableDao and DependencyDao. These classes include methods for saving, loading and updating information from and to the database. Using these data access classes the objects can be added, loaded or updated one at a time, or as a collection at the same time.

5.2.2 The Database Structure

The database consists of four tables, as Figure 5.2 shows: Algorithm, Block, Variable and Dependency. PK and FK in the tables indicate primary key and foreign key, respectively. We describe these tables briefly in the following.

Algorithm Table

Algorithm table contains general information about an algorithm. As can be seen from Figure 5.2, most of the columns of the Algorithm table are characteristics of the algorithm that are discussed in Chapter 4. Other columns of the Algorithm table are as shown in Table 5.3.

Block Table

The information about blocks of an algorithms are stored in the Block table. In addition to the Algorithm id that is a foreign key in this table, the table includes, for example, the information about a block type and length. These columns are shown in Table 5.3.

Variable Table

Variables of a program are stored in the Variable table. The Variable table includes algorithm id and block id as its foreign keys. Role of variable is also stored in this table. In addition to these, the Variable table includes other columns as shown in Table 5.3.

Dependency Table

The Dependency table stores the dependencies among variables in a program and contains the id of two variables that have a direct or indirect dependency relationship. In addition to these, the Dependency table includes other columns which are described in Table 5.3.

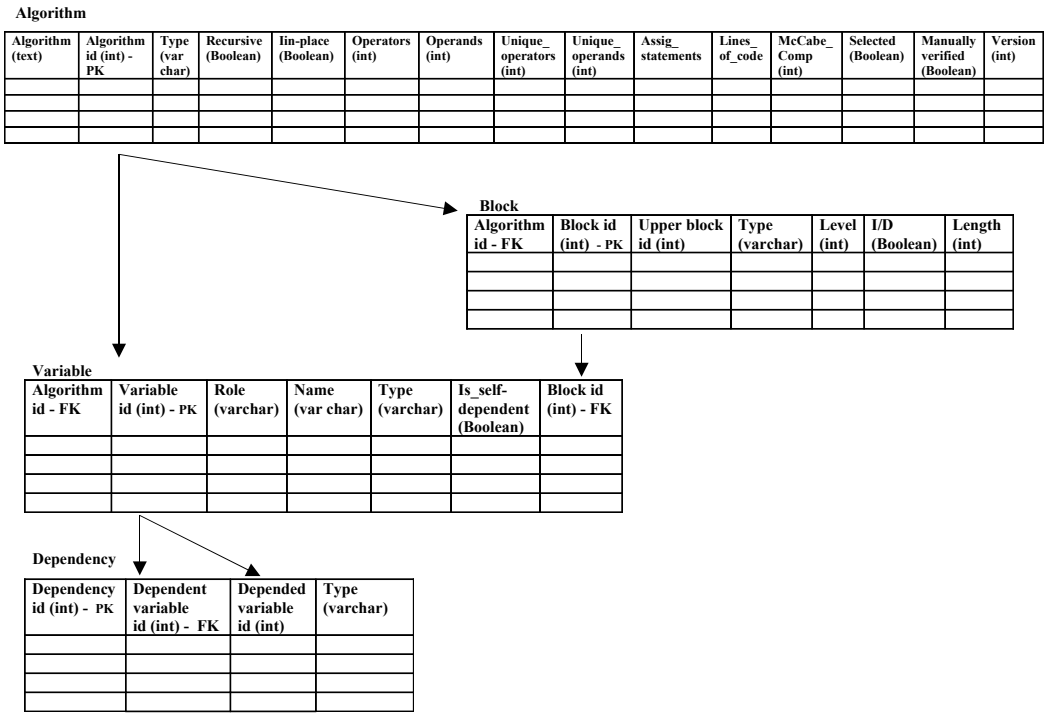

Figure 5.2: The database tables

The system uses MySQL 5.0 as its database. MySQL was selected because it is a widely used, and as the vendor advertises, the world's most popular open source database. The other reason for choosing MySQL as the database is the author's long experience of using

Table	Column	Description
Algorithm	Algorithm ID	This is an auto incremented field that generates a unique id for each algorithm.
	Algorithm	The whole source code is also stored in the database.
	Type	Type of an algorithm refers to the sorting algorithm that has been analyzed. At the moment, type can have one of the following values: Selection sort, Insertion sort, Bubble sort, Quicksort and Mergesort. If the system can not verify the algorithm, the value of this column would be "Unknown".
	Operators	The number of total operators used in an algorithm. A Halstead's parameter.
	Operands	The number of total operands used in an algorithm. A Halstead's parameter.
	Unique_operators	The number of total unique operators used in an algorithm. A Halstead's parameter.
	Unique_operands	The number of total unique operands used in an algorithm. A Halstead's parameter.
	Assig_statements	The number of total assignment statements used in an algorithm.
	Lines_of_code	The number of total code lines used in an algorithm.
	McCabe_comp	McCabe's cyclomatic complexity.
	Selected	This Boolean type column indicates whether the algorithm is used in the process of recognition an algorithm.
	Manually verified	As the name suggests, this column indicates whether the type of this algorithm is manually verified by an instructor or it is recognized only by the system.
	Version	This column refers to the version of the system that has evaluated and recognized the algorithm. Alternatively it could refer to the instructor who has processed the algorithm either manually, using the system or both.
Block	Block ID	This is an auto incremented field that generates a unique id for each block.
	Upper block ID	This column contains the id of the upper block that encompasses this block. This column helps us to determine the nested-sequential relationship between blocks and loops described in Chapter 4. The upper lock id for the global block, i.e., the class in Java, is zero.
	Type	This column refers to the type of the block, that is, whether the block is a global block, a loop (and if yes, what kind of loop), a conditional or a method (and if yes, what kind of method).
	Level	This column contains information about the level of the block. The global block has a level of zero. We can also use this column to determine the nested-sequential relationship between blocks and loops.
	I/D	The information about whether a loop is an incrementing loop or decrementing one is stored in this column. The value is false for other kind of blocks since this information is not applicable for those blocks.
	Length	The length of a block refers to the number of statements inside that block.
Variable	Variable ID	This is an auto incremented field that generates a unique id for each variable.
	Name	This column stores the name of a variable.
	Type	The type of a variable, i.e., the information about whether the variable is an integer variable, an array or of some other type, is stored in this column.
	Is self-dependent	This column contains the information about whether a variable depends on itself or not. A variable is considered to be self-dependent if it has either a direct or an indirect (or both) dependency on itself.
Dependency	Dependency ID	This is an auto incremented field that generates a unique id for each dependency.
	Type	Information about whether the dependency is a direct dependency or an indirect one.

Figure 5.3: The database tables columns

it.

5.2.3 The Dataflow

We present an example of the dataflow to describe the functionality of the Analyzer, and as a verbal explanation of the system architecture discussed in the previous section.

The main method is located in the TAPAS class. Therefore, the Analyzer can be started running the TAPAS class and giving a program file to it as an input. Right after the Analyzer is started, the Interpreter module of VILLE is called and the input is passed to it as an argument. The Interpreter reads the file in and interprets it by examining each line and determining the corresponding index. The roles of the variables used in the source code are also determined by the Role Analyzer module and the information of the methods used in the program are stored. All this information is then returned to the TAPAS in an object of the Interpreter class. In the next step, the class Algorithm is called to create an algorithm object and examine the characteristics of algorithm, e.g., whether the algorithm is a recursive and in-place algorithm or not. The information of algorithm is stored in the database using an object of the AlgorithmDao class.

The next step is to determine the blocks and variables. These are investigated in a recursive method located in the TAPAS class. The objects of blocks and variables are created and the related characteristics of the blocks and variables are examined and stored in the databases using the objects of BlockDao and VariableDao classes, respectively. These objects are also added to their respective collections for the future use. Notice that each block and variable object has to be stored in the database immediately after it is created. They cannot be added at once after all objects are created. The reason for this is that, as was described in the previous section, blocks and variables have a unique auto incremented id. Since each block needs to know its upper block id, each block id must be retrieved from the database before its inner block is created so that this id can be assigned to this just created inner block as its upper block id. A variable id, on the other hand, is needed for determining and storing the variable dependencies. Therefore, the variable id must be retrieved from the database just after its creation so that it can be used to identify its dependencies. On the contrary, all dependencies are stored at the end of the analysis, since there is no need to know dependency id immediately after its creation. There are usually a large number of dependencies (direct and indirect) among variables even in a small program. Moreover, as well known, the database operations are expensive. Therefore, the way dependencies are stored into the database could improve the efficiency.

Dependencies among variables are investigated at the same time as the analysis continues. In addition, at the end of the analysis all dependencies among variables are processed in order to update the indirect dependencies among old and recently created variables. The other numerical characteristics, e.g., the Halstead's metrics are computed in the same method as they are encounterd.

After the characteristics of the algorithm are computed, the comparison of these characteristics and those of the verified algorithms retrieved from the database starts. The Analyzer

can decide based on this comparison, whether it recognizes the analyzed algorithm. The process of comparison and recognizing an algorithm is described in detail in Chapter 6.

Limitations of the Analyzer

The Analyzer has also its own limitations. Part of these limitations is directly related to the limitations of VILLE. For example, because VILLE cannot process objects, there was no attempt made to add such ability to the Analyzer, since it is based on the indexes delivered from the VILLE. The other part of the limitations comes from the Analyzer itself. These limitations are due to the fact that the Analyzer was implemented primarily with common sorting algorithms in mind. The most severe limitation is that it cannot process some of those structures that VILLE can. As an example, the Analyzer cannot process multidimensional arrays. It was implemented to support only single arrays, in addition to the Java primitives commonly used in sorting algorithms. The Analyzer's architecture, however, allow it to extend its functionality easily if needed.

As the Analyzer was tested thoroughly in order to find its bugs, improve its robustness and add supports for different syntaxes, it was not, however, possible to test it with those test cases that caused VILLE to crash. This is because, when VILLE's crashes, the program's execution is terminated before it reaches the Analyzer. Notice, however, that this only concerns the syntactical issues. The functionality of the Analyzer was also test extensively, showing high quality: it is able to analyze the discussed structures of a program and compute all the aforementioned characteristics of it (except the roles of the variables, as was discussed before).

In the next chapter, we discuss the functionality of the Analyzer more, focusing on the process it uses to decide on the type of an algorithm it has analyzed.

Chapter 6

Results

In this chapter, we discuss the materials we used in or work and present an analysis of the outputs of the Analyzer We divide the outputs into two groups: the *numerical characteristics* and the *descriptive characteristics*. This division is based on, as the names suggest, whether a characteristic is numerical or not. The numerical characteristics can be presented by a positive integer, while descriptive characteristics cannot. Using the numerical and descriptive characteristics, we developed a decision tree to recognize sorting algorithms. The decision tree is also discussed later in this chapter.

6.1 The Process

As stated in Chapter 2, the scope of this work is limited to the following five sorting algorithms: Bubble sort, Insertion sort, Selection sort, Quicksort and Mergesort. We collected randomly a total of 51 sorting algorithms of aforementioned types for the analysis. All algorithms were randomly selected from textbooks, course material on Data Structure and Algorithms lectured at the Department of Computer Science and Engineering at the Helsinki University of Technology (HUT) and other universities and from the students' submissions on the same course lectured in Spring 2008 at the same Department at HUT (only Insertion sort and Quicksort).

Some of the selected algorithms, mainly those from the textbooks, were in pseudo language, while the others were written in Java language. Each selected pseudo algorithm was implemented in a Java class with the main method. Each implemented algorithm was located within its own method, which was invoked from the main method. There were no printing or other extra commands in the analyzed code. Notice that some implementations of Quicksort and Mergesort locate the whole algorithm into one single method, while the others split it into two or more methods. For example, the partition part of a Quicksort might be located in its own method, as well as the swap operation. In the process of analyzing, there was no attempt made to gather these parts of an algorithm into one single method. These implementations was run by the Analyzer as such, with possible printing commands eliminated. It should be noted, however, that because the Analyzer performs in-lining, splitting an al-

gorithm into more methods has no effect on the amount of the characteristics other than increasing the number of operators and unique operators by one, as a method invocation is counted as an operator and an unique operator.

All algorithms were run by the Analyzer and their characteristics were stored in Microsoft Excel. The collected characteristics were the same as shown in Table 4.1 and Figure 5.2. The characteristics were divided into two different categories: the numerical characteristics and the descriptive characteristics. The algorithms of the same type were listed in the same sheet and the minimum, maximum and average of the numerical characteristics of each algorithm type were calculated. A set of charts were created from the average of these numbers and were compared with each others. For the sake of the readability and easiness of comparison, the numerical characteristics of algorithms of each type were transferred into two different charts, one containing the large numbers and another containing the smaller numbers. Figure 6.1 shows the results for the numerical characteristics of the analyzed Quicksort, Mergesort and Insertion sort. In this figure, however, all numerical characteristics of Quicksort, Mergesort and Insertion sort are shown together in order to save the space. It should be noted that those large numbers, which are the total number of operators, the total number of operands, the program length and the lines of code, are divided by 10 in Figures 6.1 to improve the readability. Notice also that the numerical characteristics shown in the figure are average values.

All characteristics shown in Figures 6.1 are defined in Table 5.3. However, the program length and the program vocabulary are not included in those definitions, because Table 5.3 contains only the characteristics that are stored in the database, whereas, as mentioned before, the program length and the program vocabulary are not stored in the database, but are calculated from other parameters. Therefore, we define these characteristics in the following again to make it easier for the reader to follow the discussion. The program length is the sum of the total number of operators and the total number of operands, and the program vocabulary is the sum of the number of unique operators and the number of unique operands.

In addition to the numerical characteristics shown in Figure 6.1, the descriptive characteristics were also evaluated for each algorithm. The evaluated descriptive characteristics are as follows: *recursive*, *in-place* and *role of the variables*. In the following, we will present a comparison of these five different algorithms in two different regards: the numerical characteristics and the descriptive characteristics.

It should be noticed that all numerical and descriptive characteristics are computed automatically by the Analyzer. However, because the role analyzer integrated to VILLE did not generate the roles accurately enough, as described in Chapter 5, we had to evaluate the roles of the variables manually.

6.1.1 The Numerical Characteristics

After generating the numerical characteristics for the five different algorithms, we analyzed the results. The analysis showed that these five algorithms clearly fall into two groups, as

expected: Bubble sort, Insertion sort and Selection sort, on one hand and Quicksort and Mergesort, on the other hand. As the algorithms belonging to the former group are far smaller in size than the algorithms belonging to the latter group, they can be distinguished easily. This can be seen from Figures 6.1. In this figure, the numerical characteristics of Quicksort, Mergesort and Insertion sort are depicted together to make the comparison easier. Insertion sort is selected as a representative of the former group. Notice however, that instead of Insertion sort we could have shown any other algorithm of this group in the figure, i.e., Selection sort or Bubble sort, since as is described in the following, the numerical characteristics of these three algorithms are very close.

The algorithms of the first group, i.e., Bubble sort, Insertion sort and Selection sort are so close in the amount of their numerical characteristics (with the maximum difference of 6 units in program length), that it is not practical to try to distinguish them using these characteristics. The numerical characteristics of these algorithms are shown in Figure 6.2. It should be noted that in Figure 6.2, like it was the case in Figure 6.1, the total number of operators, the total number of operands, the program length and the lines of code are divided by 10 in order to improve readability. The numerical characteristics shown in the figure are average values, as is the case in Figures 6.1.

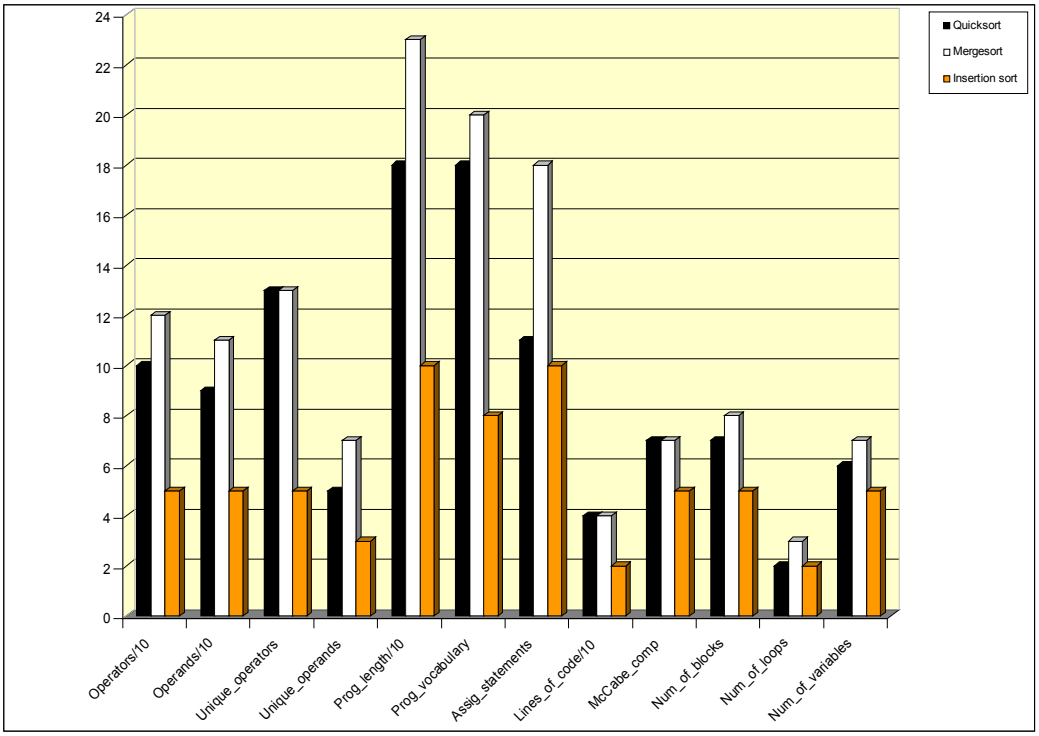

Figure 6.1: The numerical characteristics of Quicksort, Mergesort and Insertion sort. These characteristics are described in Table 5.3. The number of operators, the number of operands, the program length and the lines of code are divided by 10. The values are average values

Quicksort and Mergesort differ from each other with regard to the numerical characteristics clearly much more than the algorithms of the other group do. As can be seen from Figure 6.1, there a relatively big difference between Quicksort and Mergesort especially with regard to the number of operators, the number of operands and program length. However, it is still not quite safe to consider these differences as distinguishing factors. Relying only on these differences in the process of analyzing Quicksort and Mergesort can result in a wrong conclusion if the recognizable algorithm is implemented in a idiosyncratic way.

Instead of using the numerical characteristics to compare the algorithms within these two groups, we can, however, use them to compare the algorithms between these two groups. This is because, as can be seen from Figure 6.1, the difference among the algorithms of the two groups is much bigger than the difference among the algorithms within these two groups, with regard to the numerical characteristic. For example, the numerical characteristic for Mergesort in the order of appearance in Figure 6.1 are 120, 115, 13, 7, 235, 20, 18, 40, 7, 8, 3 and 7, whereas these numerical characteristic for Insertion sort are 49, 53, 5, 3, 101, 8, 10, 17, 5, 5, 2 and 5, respectively. In other words, it is very unlikely that a Quicksort or a Mergesort algorithm could be implemented using only these amount of, e.g., operators, operands or blocks.

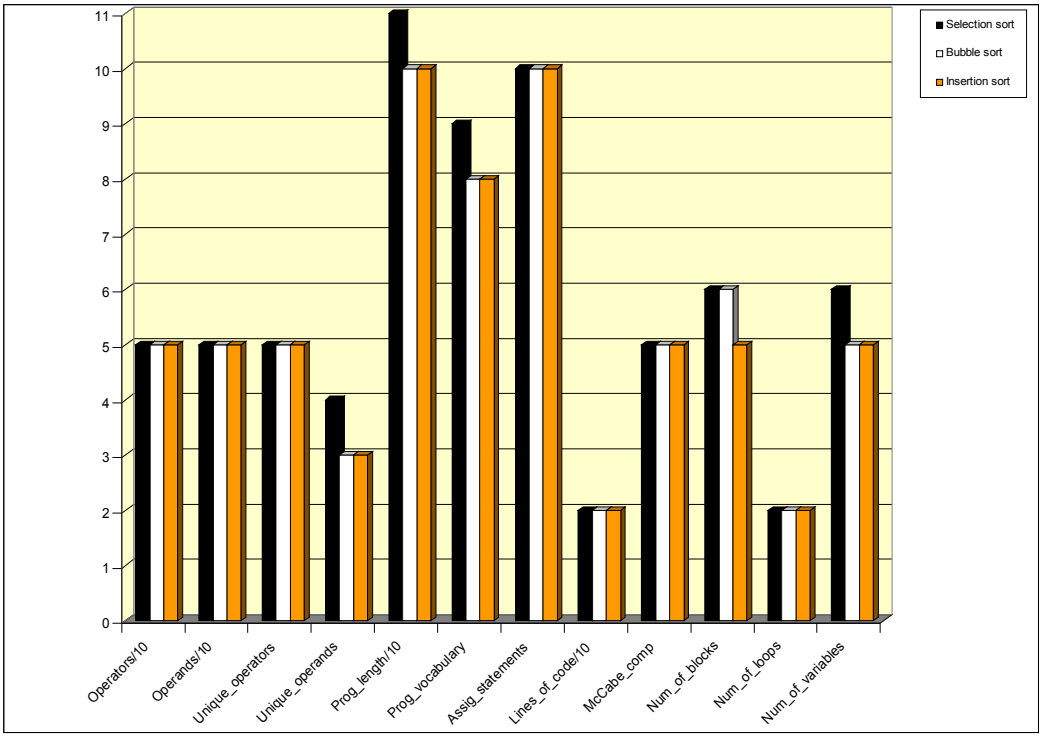

Figure 6.2: The numerical characteristics of Selection, Bubble and Insertion sort. These characteristics are described in Table 5.3. The number of operators, the number of operands, the program length and the lines of code are divided by 10. The values are average values

In the following, we describe how the numerical characteristics can be used to reject or accept algorithms as known sorting algorithms by the Analyzer. In addition, a discussion on how the knowledge base of the analyzer can be extended during its usage is presented.

Extending the knowledge base of the Analyzer

The numerical characteristics can be used to by the analyzer to decide whether an algorithm could possibly be one of the algorithms that it knows.

Figures 6.3 show the minimum and the maximum values of the numerical characteristics of all five analyzed algorithms. The upper figure shows the minimum values and the lower figure shows the maximum values. The Analyzer has all these values in its database and can use them in the process of analyzing an algorithm. After the Analyzer has analyzed the algorithm, it compares each of its numerical characteristics to those minimum and maximum values of the corresponding numerical characteristics retrieved from its database, in turn. If one or more numerical characteristic of the analyzed algorithm is below the minimum value obtained from the database for that particular numerical characteristic or above the maximum value obtained from the database for the characteristic, the algorithm is labeled as not belonging to any of these five types.

As were previously discussed, at the present moment, there are 51 different versions of these five algorithms in the database of the Analyzer. Each of these algorithms has its name in the *Type* column described in Table 5.3. When the Analyzer recognizes an algorithm as a particular type, it assigns the appropriate value to the Type column of the algorithm and stores it into the database. If, on the other hand, the algorithm is not recognized by the Analyzer, its Type column gets a value of "Unknown". This way, it is possible to manually examine the algorithm from the database and verify the decision made by the Analyzer just by searching the algorithm from the database with Unknown type. In these situations, one of the two following cases can be in question: true negative or false negative. The former scenario indicates that the Analyzer has made an accurate decision, i.e., the algorithm is correctly rejected and it does not belong to any category known to the Analyzer. The latter scenario, which is more interesting, indicates that the algorithm does indeed belong to one of the categories and thus, is mistakenly rejected. In these cases the type of the algorithm can be changed from "Unknown" to the type it actually belongs to. The knowledge base of the Analyzer can be extended this way. Next time, when performing the comparisons, the Analyzer also uses the numerical characteristics of the algorithm that was manually labeled to the correct type, and therefore, it can recognize the similar algorithm as that particular type.

6.1.2 The Descriptive Characteristics

The descriptive characteristics of the algorithms analyzed in this work were the following: whether the algorithm is a recursive one, whether the algorithm is an in-place one and the roles of variables used in the algorithm. The descriptive characteristics appear to be very useful and distinguishing characteristics. In the following, we will describe the way these

characteristics can help in recognition of the algorithms. First, we will discuss the different implementations of the same type, and after this, we will explain the case of distinguishing among different types.

The descriptive characteristics appear to be very similar in different implementation of the same algorithm. We will explain this separately for each of the five algorithms is the following. Note that since there is an array to be sorted in all examined algorithms, the role *organizer* appears in all of them and, therefore, is not much informative for our purpose. Thus, it is omitted from the following discussion.

Quicksort

All studied Quicksorts are recursive and in-place. The following roles appear together almost in all of these algorithms: stepper, fixed value and temporary. There are only two algorithms that do not have these roles all together. These roles are actually the only roles that appear in examined Quicksort algorithms. Moreover, the role most-wanted holder appears in two algorithms. Note that variables appearing in walker role shown in Table 4.1 are eliminated by Analyzer as the result of in-lining for the examined Quicksorts.

Mergesort

All examined Mergesorts are recursive and, except one, none of them are in-place, i.e., all of them need an auxiliary array to performing the sorting, except one. The variables in all algorithms appear only in stepper and fixed value roles. As mentioned in the case of Quicksorts, the variables playing walker role shown in Table 4.1 are eliminated by Analyzer as the result of in-lining for the examined Mergesorts.

Selection sort

As expected, none of the Selection sorts is recursive, and all of them are in-place. The following three roles appear in all of them: stepper, temporary and most-wanted holder. The most-wanted holder role is a unique feature of Selection sort among the three algorithm types of this group, and as we will see in the next section, it plays an important role in distinguishing Selection sort from the other two.

Insertion sort

Again, as expected, all studied Insertion sorts are non-recursive and in-place. The roles stepper and temporary are the only roles that variables play in all examined algorithm. There is an extra variable appearing in fixed value role only in one of the algorithms.

Bubble sort

As is the case for Selection and Insertion sorts, all examined Bubble sorts are non-recursive and in-place algorithms. The variables in all algorithms appear only in stepper and temporary roles.

Now that different implementations of the same type are discussed, it is easy to notice that descriptive characteristics really are quite reliable distinguishing factors for the five sorting algorithm. It turns out that perhaps the most helpful and important distinguishing factor is the role of variables used in the algorithms. Since the role of variables is the factor that our work is based on, this looks like a good result with regard to our work. As we will see in the next section, however, the roles of variables are not enough by themselves to recognize algorithms. A combination of roles of variables and the other characteristics is needed.

In the next section, we present an overview of our method to recognize the five algorithms examined in this work and discuss a decision tree that illustrates how the method can be applied.

6.2 The Decision Tree

In this section we make use of a combination of the numerical and descriptive characteristics to find the most reliable and distinguishing method that can be used in recognizing algorithms using these characteristics.

At the moment that we were analyzing the results, there was not available an accurately functioning role analyzer. Therefore, the roles of all variables in all 51 algorithms were evaluated manually. All roles discussed in the following are, thus determined manually.

By analyzing the information we gathered from all five types of algorithms, we developed a decision tree shown in Figure 6.4. The idea is that the closer we are to the root of the tree, the more reliable the characteristics used for distinguishing among the algorithms are. For example, in the root we have the characteristic of whether the recognizable algorithm is a recursive one. This is a very reliable characteristic since Quicksort and Mergesort are always implemented in a recursive way, and the other three algorithms, Selection sort, Insertion sort and Bubble sort are not. Of course one can question the using of the word *always* in this context arguing that since every recursive algorithm can be converted to a non-recursive one and vice versa, there is no certainty here and our assumption can not be backed. Although this is a valid criticism, we can answer to it by arguing that in this sense, certainty is a relative concept, like many other concepts. Reasoning is always based on some assumption. The question is how reliable that assumption is. We consider it very rare that someone implements a, say Insertion sort in a recursive way or a Quicksort in a non-recursive way.

As Figure 6.4 shows, we start the process of recognizing an algorithm by investigating whether it is a recursive algorithm or not. If it is not, we can assume that the sorting algorithm is one of the three non-recursive algorithms, namely, Selection sort, Insertion sort or Bubble sort. In the next step, we examine whether the numerical characteristics of the

algorithm are within the permitted limits. Permitted limits are, as was discussed in the previous section, values that are bigger than minimum values retrieved from the database and values that are smaller than maximum values retrieved from the database. These minimum and maximum values for the five studied algorithms are shown in Figure 6.3. If it turns out that the amount of one or more numerical characteristics of the algorithm are bigger than the biggest amount that exists in the Analyzer knowledge base, then we can conclude that the algorithm in question cannot be of the type known to the analyzer. As a result, the algorithm is rejected and labeled as unknown without any further examination. An error message is given to the user and the analyzing process is terminated. If the numerical characteristics of the algorithm are within the permitted level, the process continues. The next step is to investigate whether the algorithm include a variable appearing in the most-wanted holder role. As we explained in the previous section, among these three algorithms, only Selection sort include this role, since it works so that the smallest (or the biggest) element is selected from the array to be sorted, and added to an appropriate place (see the definition of most-wanted holder role in Chapter 4). Therefore, if we find the most-wanted holder role in the algorithm, we can conclude that it is a Selection sort. If the algorithm does not include most-wanted holder role, we continue the investigation to see whether the outer loop is incrementing and inner one decrementing. As well known, all of these three algorithms contain two nested loops. In our analysis, the outer loops in all Insertion sorts were incrementing and the inner loops were decrementing. On the other hand, in the case of Bubble sorts there was only one implementation (namely Clifford A. Shaffer's textbook) with this kind of loops. So if the outer loop is incrementing and the inner one decrementing, we continue our investigation. The last step in decision tree is to examine whether the counter of inner loop gets its first value from the counter of the outer loop. This is the case for all examined Insertion sort, but never occurs in the case of the Bubble sorts. In all examined Bubble sorts, the counter of inner loop is always initialized to 0 or 1, or gets its first value from the length of the array to be sorted.

The other branch from the root is the case of recursive sorting algorithms that, in our study, are Quicksort and Mergesort. At the first step, we examine the numerical characteristics in the same way that was discussed above for the algorithms of the other group, and if the algorithm appears not to fit within the permitted limit, an error message is given printing out the numerical characteristics that do not fit and the information about whether the analyzed algorithm is recursive or not. If the numerical characteristics fit within the permitted level, the process goes ahead to the next step. The next investigation to distinguish between Quicksort and Mergesort is to see whether the algorithm includes a variable appearing in temporary role. As was described in the previous section, all examined Quicksorts include this role, but none of the examined Mergesorts does. It is somehow expected: Quicksort includes a swap operation, but in Mergesort, since the merging is carried out, there is no need for swapping. Temporary role appears often in swap operations (see the definition of temporary role in Chapter 4). Therefore, if an algorithm does not include the temporary role, we can conclude that it must be a Mergesort. But if it does, we can continue the investigation to check the other descriptive characteristics instead of just concluding that the algorithm is a Quicksort. This makes the system more precise, tolerant and robust in the cases where an idiosyncratically implemented Mergesort happens to have a temporary

role. The next step is to examine whether the algorithm is in-place. As commonly known, Mergesort typically needs an auxiliary array to carry out the sorting, while Quicksort does not. Because we encountered with one case where a Mergesort was implemented without using an auxiliary array, we consider it possible that although the algorithm is an in-place one, it still might be a Mergesort. The next step will ensure us. In the next step, we examine whether the algorithm is a tail recursive. All examined Quicksorts are tail recursive, while none of examined Mergesort is.

In the decision tree, the numerical characteristics are used to stop the recognition process if the algorithm does not appear to fit within the permitted limit. The reason way the numerical characteristics are examined after the examination of whether the algorithm is recursive or not, is that this allows us to retrieve only the numerical characteristics of the recursive, or non-recursive algorithms from the database, since we know this in this step. It results in considerable efficiency improvement, since we do not need to retrieve the information of all algorithms from the database. It is enough to retrieve the information of half of the algorithms from the database, if we assume that fifty percent of the algorithms in the database are recursive, and other fifty percent are not. This could mean a noticeable improvement in efficiency if the database contains a large number of algorithms. The other advantage of investigating the numerical characteristics in this step is that this makes the Analyzer able to generate more descriptive error messages. The Analyzer can tell to the user whether the rejected algorithm is a recursive or non-recursive one.

The error message that the Analyzer gives to the use is informative. If one or more of these numerical characteristics are above or below the permitted limits, the error message includes the precise information about it. In addition to the information that the error message gives about the algorithm (recursive or not), it also include the particular characteristic/characteristics that is/are not within the permitted range. The information about whether that/those different characteristic/characteristics is/are above the permitted limit or below it is also included in the error message. For example, when the system was tested by a hybrid algorithm consisting of Quicksort and Insertion sort, it gave the following error message: *"The algorithm seems to be a recursive algorithm that has the following characteristics out of the permitted limit: Program vocabulary, Lines of code, McCabe complexity and Number of blocks are above the permitted limit".*

The decision tree appeared to work well with the algorithm we analyzed in this study. It is clear, however, that the decision tree should be extended to include other algorithms as well. Other reliable factors must be devised and used to make the results more convincing and the method more extensive. We will discuss this in the next chapter.

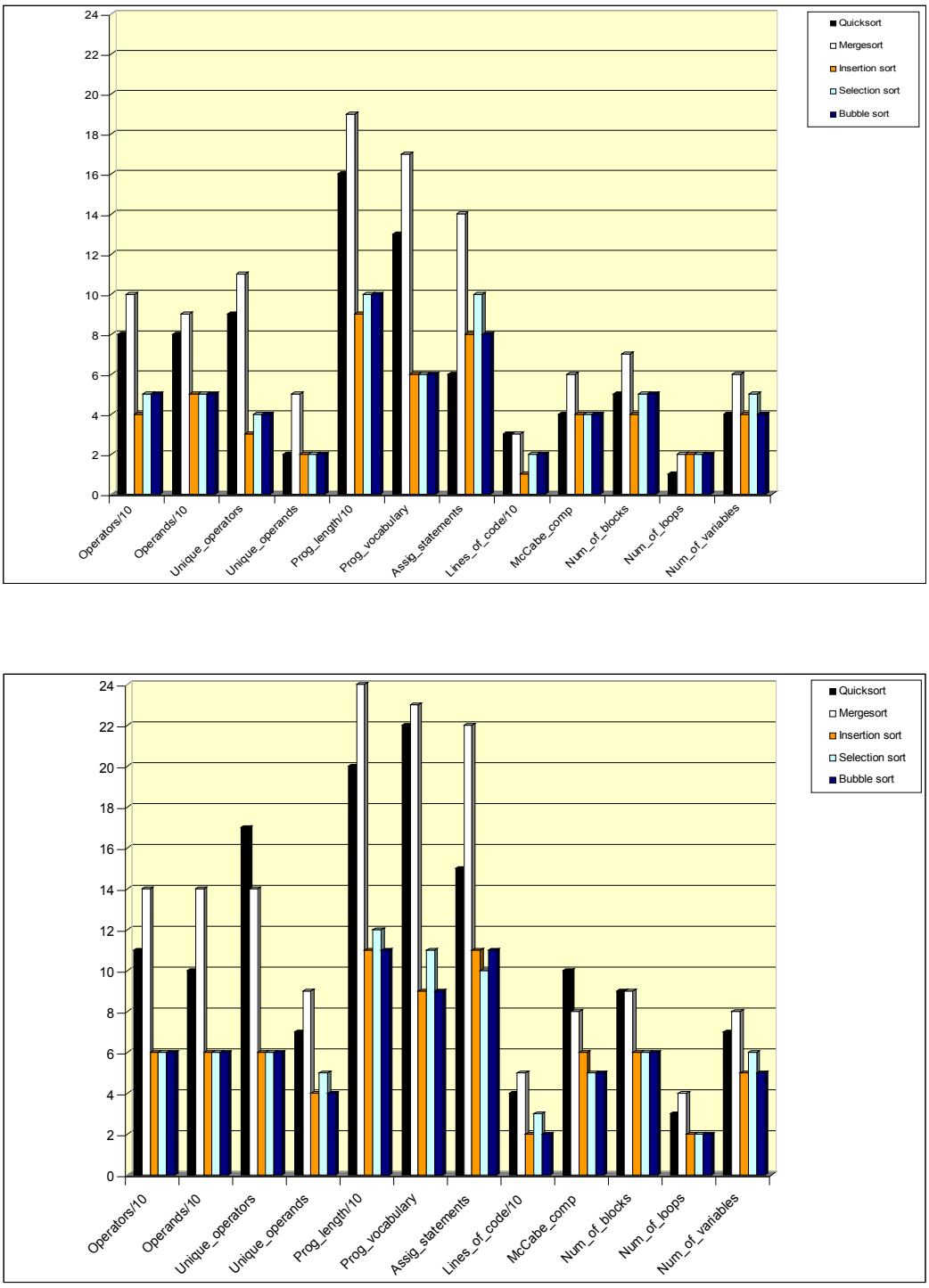

Figure 6.3: The numerical characteristics of Quicksort, Mergesort, Selection, Bubble and Insertion sort. These characteristics are described in Table 5.3. The upper figure shows the minimum values and the lower figure shows the maximum values

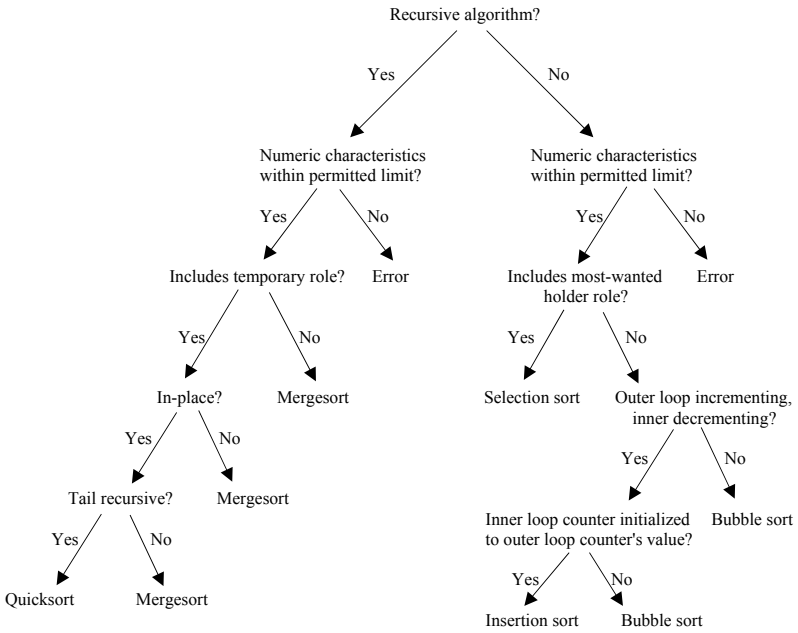

Figure 6.4: Decision tree for determining the type of an sorting algorithm

Chapter 7

Conclusion and Future Work

In this chapter, we present a general discussion on our study from different points of view. Issues related to the reliability of the system are also discussed explaining a set of tests that were used to determine the Analyzer's ability to recognize the sorting algorithms. We also present some ideas and views about how the study could be continued.

7.1 Discussion

Although the objectives and the requirements set in Chapter 1 are met, the Analyzer could not be tested thoroughly with other kinds of algorithms due to the lack of a properly working role analyzer. This was a great disappointment and resulted in the fact that the reliability of the result is yet to be more investigated. Despite this, the Analyzer is able to distinguish and classify the sorting algorithms analyzed in this work and meet all those requirements discussed in Chapter 1.

In this section, we discuss the functionality of the Analyzer by explaining how the reliability of the decisions made by the Analyzer was tested using different test cases. A discussion on the possible applications of the system is also presented.

7.1.1 Reliability

A set of different algorithms were run by the Analyzer to test its functionality and to examine how accurately and precisely it makes its decisions. The algorithms were randomly selected mainly from the students' submissions on Data Structure and Algorithms lectured in Spring 2008 at the Department of Computer Science and Engineering at the Helsinki University of Technology. These algorithms included hybrid algorithm consisting of Quicksort and Insertion sort, graph algorithms and some algorithms submitted by the students on arbitrary precision calculator exercises. In addition, some other types of sorting algorithms were used as test materials, as will be explained in the following.

The tests were designed to cover different cases. We divide the tests into four categories:

true positive, *true negative*, *false positive* and *false negative*. This division allows us to cover all aspects of the behaviour of the Analyzer. The true positive category was not tested, since it has been tested already by the five sorting algorithms analyzed in this work as described in the previous chapter. In order to cover all these categories, some manipulated versions of the five sorting algorithms were used as well. These are discussed separately in the following. It should be noted that the roles of variables was not part of the tests, because as described previously, the role analyzer integrated into VILLE did not generate the roles precisely. Notice also, that a part of these collected test files could not be tested mostly because they contained such structures that was not supported, such as switch statement (see the discussion on limitation of VIILLE from Chapter 5).

True Negative Cases

When Shell sort algorithm was given as input to the system, the Analyzer gave the following error message: "The algorithm seems to be a non-recursive algorithm that has the following characteristics out of the permitted limit: program length and the number of the loops are above the permitted". It is a correct error message, since the Shell sort code was longer and had three for loops.

Many algorithms that were tested fall into this category. For example, graph algorithms and arbitrary precision calculator exercises are not recognized by the Analyzer as one of the five sorting algorithms, since they are much longer algorithms. In these cases, the error messages include almost all numerical characteristics as being above the permitted limits.

Many different versions of Heapsort were also tested resulted in correctly rejecting the algorithm. The error message was as follows: "The algorithm seems to be a non-recursive algorithm that has the following characteristics out of the permitted limit: Number of operators, Number of operands, Number of unique operators, Number of unique operands, Program length, Program vocabulary, Line of code, Assignment statement, McCabe complexity, Number of blocks and Number of loops are above the permitted". In other words, the error message includes almost all numerical characteristics except the number of variables. The numerical characteristics for the tested Heapsort that resulted in this error message were as follows: 98, 94, 12, 6, 192, 18, 15, 36, 7, 8, 3 and 6. Note that the order is the same as the order in Figure 6.2 presented in Chapter 6. As can be seen from Figure 6.2, all numerical characteristics were above the maximum amount of those non-recursive sorting algorithm shown in the figure except the number of variable, which is as same as the maximum amount in the figure.

False Positive Cases

Due to the fact that the roles generated to the Analyzer are not precise, the false positive cases did not occur. That is, although the numerical characteristics could have been adjusted to fit into the limited range, because the roles did not happen to be those the Analyzer uses in its conclusion (see Figure 6.4 presented in Chapter 6), none of tested algorithms that do not actually belong to the analyzed sorting algorithms was falsely labelled as such. For

example, the Heapsort described previously in connection with the true negative cases, was changed so that no error message generated with regard to its numerical characteristics. However, the algorithm was still labelled as "Unknown", because it did not include the appropriate roles to be recognized as a Selection sort, nor it did meet the requirements to be recognized as an Insertion or a Bubble sort.

One of the other tests was carried out by adding extra commands to the program that recursively compute Fibonacci numbers, so that the numerical characteristics fit in the permitted range of recursive algorithms, and make the Analyzer believe that a the algorithm is a recursive sorting algorithms. The type of the algorithm was labelled, however, still as "Unknown", since other requirements were not met.

False Negative Cases

It was easy to deceive the Analyzer to make it produce false negative cases. Simply by adding some extra and irrelevant code, for example a swap operation, the Analyzer can be made believe that the algorithm is not the type it actually is. This was tested by changing many algorithms that the Analyzer knows, resulted in rejecting the algorithm. As an example for non-recursive algorithms, an extra for loop with empty body was added to an Insertion sort, resulting in an error message that the algorithm is a non-recursive algorithm that has the number of loop above the permitted level. The error message comes from the fact that all three analyzed non-recursive algorithm that exist in the Analyzer knowledge base are implemented using only two loops. Three lines of swap code were added to a Selection sort, as another test of this category. The tested Selection sort was the algorithm that has the maximum of the numerical characteristics depicted in Figure 6.2 in Chapter 6. The results was an error message telling that all numerical characteristics were above the permitted limit except the number of loops, which was not changed.

As another example for recursive algorithms, an extra for loop with empty body was added to an Mergesort, that has the maximum values shown in Figure 6.1, resulting in an error message that the algorithm is a recursive algorithm that has the number of operators and the number of operands above the permitted level, among others.

These kinds of false negative occur as a result of a bad coding style or idiosyncratically written code. We will discuss this later in more detail. The analyzer is made to recognize sorting algorithms that are implemented with good coding style. It is not tolerant to the changes that result from using an algorithm in an application. As was discussed in the previous chapter, those algorithms that are mistakenly labelled as unknown can be corrected manually in the database. This results in the fact that the Analyzer will learn from its mistakes and will extend its knowledge base.

7.1.2 Evaluation

The most important result we have achieved is perhaps the fact that the role of variables can indeed be applied in automatic program recognition tasks. Although we can not generalize

this to other algorithms, we can argue that at least in the study of five types of sorting algorithms the role of variables played a crucial role.

It is obvious, however, that the role of variables is not alone enough to distinguish algorithms for the two following reasons. Firstly, two different algorithms may have exactly the same amount of variables with the same roles, e.g., Insertion sort and Bubble sort, as were discussed in Chapter 6. Secondly, the amount and roles of variables might be different when an algorithm is used in an application. This can cause the system to make wrong decisions. In other word, other distinguishing factors are needed. For example, as the results of our work showed, investigation of loops can offer a unique and distinguishing way to perform the task. Overall, we can cautiously claim that perhaps there exists a new method to automatically recognize and maybe comprehend programs based on static analysis that has not been studied or has not been presented in the literature yet.

Argument can be made about whether the numerical characteristics are more reliable as distinguishing factors than descriptive characteristics. Although it is not easy to give an exact answer to this question, the following can be argued in this regard. As was discussed in the previous chapter, it is difficult to distinguish the analyzed sorting algorithms using only the numerical characteristics, as these characteristics are, especially among the non-recursive algorithms, very similar (see Figure 6.2). Therefore, the numerical characteristics can be best used in eliminating algorithms that do not fall within the range of those algorithms that exist in the Analyzer knowledge base. However, it is well possible that false negative and false positive errors occur if an algorithm is implemented in an idiosyncratic or different way, as we explained previously. The descriptive characteristics, on the other hand, are perhaps more reliable in this sense. For example, it is possible to implement a Quicksort in a non-recursive way, but how often can this be the case in real life?

One can criticize our method arguing that, as Ben-Ari and Sajaniemi argue [7], the role of variables is a cognitive concept. Algorithms, on the other hand, are not in the same sense a cognitive concept, since they have a well-known type and functionality. The argument can be made about whether a variable has a role of temporary or fixed value, but the same argument can not be made about whether a sorting algorithm is, say Quicksort or not. Having said this, can we base the recognition or understanding of an algorithm on a cognitive concept in a reliable way? Let us assume the following scenario: on one hand, it is acceptable that two different roles, X and Y, can be assigned to a variable by two different people, and on the other hand, an algorithm is recognized assuming that the role of that particular variable is X. The question is, how reliable the conclusion is? As a concrete example, we can discuss the process of recognition a Selection sort. As can be seen from the decision tree presented in Figure 6.4, the first step in the process of recognizing a non-recursive algorithm is based on whether the algorithm includes a variable appearing in the most-wanted holder role. If it does, then we conclude that it is a Selection sort. In order to be able to conclude this, we must trust that a variable that appears in this role cannot be regarded having another role in the same location of the code. If we use a role analyzer developed by a person who believes that the most-wanted holder cannot be assigned to that particular variable in Selection sort, and it has actually, for instance, a temporary role, then our decision tree does not work properly anymore. The assumption of our approach is that, a characteristic

used in recognizing an algorithm is both distinguishing and unambiguous. According to the definition of the most-wanted holder role presented in Chapter 4, we believe that, however, that the particular variable in Selection sort can only be assigned a most-wanted holder role, and not any other role. These kinds of questions should also be further researched and considered in the future work.

7.1.3 Applications

Perhaps the most useful application of the system is verifying students' submissions. As stated before, there are many large size courses lectured at universities, where students are required to submit a number of exercises in order to complete the course. The system can be used to help instructors to verify the correctness of the type of the submission. As one concrete example, in order to complete a course on Data Structure and Algorithms, the students must submit many sorting algorithms. The system can be used to verify the type of the algorithm that a student is supposed to submit. It is also possible to develop the system to provide the students with detailed feedback about their submissions. As an example, if a submission resembles an algorithm from a textbook, or if it differs from it in a particular way, the student can be noticed about it. The system, at the present time, does not, however, support these kinds of functionality, but these can be integrated into it.

It must be noticed that the system cannot make decisions about whether an algorithm works accurately or whether it contains bugs. This was not a part of requirements in the first place. The system can only inform the user, based on its knowledge base, what type of the sorting algorithms the recognizable algorithms looks like. It is very difficult, if not impossible, to verify the accuracy of an algorithm using static analysis. Although some tools try to examine the correctness of programs by static analysis, the results are by no means thorough and reliable. As an example, PAT, described in Chapter 3, is able to detect a wrong order in a swap operation using the swap plan defined in its knowledge base. This is almost the best what a static program analysis tool can do in order to verify the correctness of a program. It is obvious that this is far away from verifying the correctness of, for example, a Quicksort. In order to examine the accuracy of an algorithm in a reliable way, a dynamic analysis is needed. Therefore, it can be claimed that only by using a hybrid analysis that includes both static and dynamic analysis, a tool could perform both tasks: understanding or classifying an algorithm and verifying its accuracy.

Moreover, the assumption in our study is that the input algorithm is coded in a sensible way and according to common and well-established coding conventions. If an algorithm is implemented in an odd way using, for example, a lot of unnecessary variables or statements, it may well be that, as we discussed previously in connection to the reliability of the system, the system does not recognize it as being a particular algorithm, even if it might be, and vice versa. The system is not equipped by mechanism to detect the cheatings, like some plagiarism detection systems do. As a matter of fact, as we mentioned in Chapter 3, knowledge-based program understanding approaches have been criticized for the same limitation.

7.2 What Is Next?

Although the results we obtained have not been applied to other algorithms and their reliability, certainty and generalizability remains yet to be examined and tested further, they were, however, promising enough to keep us encouraged to continue the research on the subject.

In the future, a reliable role analyzer is needed. It is essential to our research. The VILLE system has to be improved also. As was mentioned in Chapter 5, VILLE is indeed being developed constantly.

In the following, we present some methods that can be used to confirm the results and discuss some ideas that can be applied in the future work. These are left to be done later, as they are beyond the scope of this work.

7.2.1 Further Research

Some ideas and suggestions for the future work are present in the following. Notice that these suggestions are presented in a general manner, and their usefulness should be carefully studied before applying.

In the future, more test materials are needed make us able to perform precise calibration. Applying calibration methods can help the system to handle the false negative cases discussed in the previous section.

Each algorithm with its characteristics can be considered to belong to a particular population. We can test the algorithms using, e.g. SPSS to see whether they belong to the same population and if not, by what p-value they differ. A appropriate test for our purpose can be chi-square, for instance.

Clustering can be used to classify the algorithms according to their characteristics.

We should also investigate the possibility of using data mining techniques to see whether it can lead to results.

To evaluate the accuracy and reliability of the results, all appropriate methods, including aforementioned methods can be applied to examine whether an algorithm is of a particular type. If all these methods confirm the result, we can be convinced that results are correct. Alternatively, we can accept the results based on whether the majority of these methods confirmed that the algorithm indeed belongs to that particular population.

The study should be extended to include other well-known algorithms. In the future work, it should be investigated, whether the role of variables and other characteristics can be applied to recognize other algorithms, as well. In addition, more material for analyzing should be collected.

The Analyzer should be developed to be able to detect especially false positive and false negative cases better. The Analyzer should provide the user with more detailed information and tell him, for example, to what extent a rejected algorithm differs from the category that

it is near to. It should also be considered to put the different characteristics into order according to their importance and reliability in recognition process. It could be done by giving a particular value to each characteristic. Moreover, the tolerance of different characteristics should be different according to how critical a particular characteristic is considered to be.

Bibliography

[1] Ahtiainen A., S. Surakka, and Rahikainen M. Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises. In *Proc. of the 6th Baltic Sea Conference on Computing Education Research, Uppsala, Sweden*, pages 141–142, 2006.

[2] Kirsti M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. In *Computer Science Education, Vol. 15, No. 2*, pages 83–102. Routledge, part of the Taylor and Francis Group, 2005.

[3] R. Arnold. *Software Reengineering*. IEEE Press, 1992.

[4] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.

[5] Y. Bar-Hillel, M. Perles, and E. Shamir. *On Formal Properties of simple Phrase Structure Grammaras*. Zeit. Phonetik, Sprachwiss. Kommunikationsforsch. 14, 1961.

[6] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 156–165. ACM, 2005.

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

[8] H.L. Berghel and D.L. Sallach. Measurements of program similarity in identical task environments. In *ACM SIGPLAN Notices archive, Volume 19 , Issue 8*, pages 65–76. ACM Press, 1984.

[9] Ted J. Biggerstaff. Design recovery for maintenance and reuse. In *Computer, Volume 22, Issue 7*, pages 36–49. IEEE, 1989.

[10] Ilene Burnstein and Katherine Roberson. Automated chunking to support program comprehension. In *5th International Workshop on Program Comprehension (WPC '97)*, pages 40–49. IEEE, 1997.

[11] Ilene Burnstein, Katherine Roberson, Floyd Saner, Abdul Mirza, and Abdallah Tubaishat. A role for chunking and fuzzy reasoning in a program comprehension and

debugging tool. In *9th International Conference on Tools with Artificial Intelligence (ICTAI '97)*, pages 102–109. IEEE, 1997.

[12] Irene Burnstein and Floyd Saner. An application of fuzzy reasoning to support automated program comprehension. In *Proceedings of Seventh International Workshop on Program Comprehension, 1999.*, pages 66–73. IEEE, 1999.

[13] J. Carter, J. English, K. Ala-Mutka, M. Dick, W. Fone, U. Fuller, and J. Sheard. ITICSE working group report: How shall we assess this? *SIGCSE Bulletin*, 35(4):107–123, 2003.

[14] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: a taxonomy. In *IEEE Software, Volume: 7, Issue: 1*, pages 13–17. IEEE, 1990.

[15] Richard Clayton, Spencer Rugaber, Lyman Taylor, and Linda Wills. A case study of domain-based program understanding. In *5th International Workshop on Program Comprehension*, pages 102–110. IEEE, 1997.

[16] Paul Clough. Old and new challenges in automatic plagiarism detection. WWW site at `http://ir.shef.ac.uk/cloughie/papers/pas_plagiarism.pdf/`.

[17] Ourston D. Program recognition. In *IEEE Expert, Volume: 4, Issue: 4, Winter 1989*, pages 36–49. IEEE, 1989.

[18] John L. Donaldson, Ann-Marie Lancaster, and Paula H. Sposato. A plagiarism detection system. In *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, pages 21–25. ACM, 1981.

[19] Bruce S. Elenbogen and Naeem Seliya. Detecting outsourced student programming assignments. In *Journal of Computing Sciences in Colleges*, pages 50–57. ACM, 2007.

[20] Johnson W.L. Erdem A. and Marsella S. Task oriented software understanding. In *Proceedings. 13th IEEE International Conference on Automated Software Engineering*, pages 230–239. IEEE, 1998.

[21] David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 266–270. ACM, 1999.

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

[23] N.K. Gupta and R.E. Seviora. An expert system approach to real-time system debugging. In *Proc. First Conf. Artificial Intelligence Applications, CS Press, Los Alamitos, Calif., 1984*, pages 336–343. IEEE, 1984.

[24] Stephen h. Edwards. Improving student performance by evaluating how well students test their own programs. In *ACM Journal of Educational Resources in Computing, Vol. 3, No. 3, Article No. 1, September 2003*. ACM, 2003.

[25] M.T. Harandi and J.Q. Ning. Knowledge-based program analysis. *Software IEEE*, 7(4):74–81, January 1990.

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

[27] Mary Jean Harrold and Brian Malloy. A unified interprocedural program representation for a maintenance environment. In *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 19, NO. 6*, pages 584–593. IEEE, 1993.

[28] 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*, pages 46–50. ACM Press, 2002.

[29] Vesa Hirvisalo. *Using Static Program Analysis to Compile Fast Cache Simulators*. Phd thesis, Department of Computer Science and Engineering, Helsinki University of Technology, Finland, 2004. Available online at `http://lib.tkk.fi/Diss/2004/isbn9512270137/isbn9512270137.pdf`.

[30] Lester J. Holtzblatt, Richard L. Piazza, Howard B. Reubenstein, Susan N. Roberts, and David R. Harris. Design recovery for distributed systems. In *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 23, NO. 7*, pages 461–472. IEEE, 1997.

[31] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proceedings of 28th ACM SIGCSE Symposium on Computer Science Education*, pages 335–339, 1997.

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

[33] W.L. Johnson and Soloway E. Proust: Knowledge-based program understanding. In *IEEE Transactions on Software Engineering, Volume SE-11, Issue 3, March 1985*, pages 267–275. IEEE, 1984.

[34] J.K. Joiner, W.T. Tsai, X.P. Chen, S. Subramanian, J. Sun, and H. Gandamaneni. Data-centered program understanding. In *Proceedings of International Conference on Software Maintenance*, pages 272–281. IEEE, 1994.

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

[36] Mike Joy, Nathan Griffiths, and Russell Boyatt. The boss online submission and assessment system. In *ACM Journal on Educational Resources in Computing, Vol. 5, No. 3, September 2005. Article 2.* ACM, 2005.

[37] Mike Joy and Michael Luck. Plagiarism in programming assignments. In *IEEE TRANSACTIONS ON EDUCATION, VOL. 42, NO. 2*, pages 129–133. IEEE, 1999.

[38] Young-Chul Kim and Jaeyoung Choi. A program plagiarism evaluation system. In *Computational Science and Its Applications - ICCSA 2005*, pages 10–19. Springer Berlin / Heidelberg, 2005.

[39] Wojtek Kozaczynski, Jim Ning, and Tom Sarver. Program concept recognition. In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference*, pages 216–225. IEEE, 1992.

[40] Mikko-Jussi Laakso, Tapio Salakoski, Ari Korhonen, and Lauri Malmi. Automatic assessment of exercises for algorithms and data structures – a case study with TRAKLA2. In *Proceedings of Kolin Kolistelut / Koli Calling – Fourth Finnish/Baltic Sea Conference on Computer Science Education*, pages 28–36. Helsinki University of Technology, 2004.

[41] Ronald J. Leach. Using metrics to evaluate student programs. In *ACM SIGCSE Bulletin Volume 27 , Issue 2*, pages 41–43. ACM, 1995.

[42] Y. Limpiyakorn and I. Burnstein. Applying the signature concept to plan-based program understanding. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pages 325–334. IEEE, 2003.

[43] Wills L.M. Flexible control for program recognition. In *Proceeding of Working Conference on Reverse Engineering*, pages 134–143. IEEE, 1993.

[44] Lauri Malmi. *Pascal-ohjelmien samankaltaisuuden tutkimisesta*. Licenciate's thesis, Helsinki University of Technology, Finland, 1989.

[45] Lauri Malmi, Ari Korhonen, and Riku Saikkonen. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *Proceedings of The 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*, pages 55–59, Aarhus, Denmark, 2002. ACM Press, New York.

[46] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *16th IEEE International Conference on Automated Software Engineering*, pages 107–114. IEEE, 2001.

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

[48] Department of Information Technology at University of Turku. Ville - visual learning tool. `http://verkkoopetus.cs.utu.fi/Research/VILLE/index_en.html`.

[49] Karl J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. In *ACM SIGCSE Bulletin, Volume 8, Issue 4 (December 1976)*, pages 30–41. ACM, 1976.

[50] Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw. Theories and techniques of program understanding. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 37–53. IBM Press, 1991.

[51] Alex Quilici. A memory-based approach to recognizing programming plans. In *Communications of the ACM, Volume 37 , Issue 5*, pages 84–93. ACM, 1994.

[52] Alex Quilici. Reverse engineering of legacy systems: a path toward success. In *Proceedings of the 17th international conference on Software engineering*, pages 333–336. ACM, 1995.

[53] Alex Quilici and David N. Chin. Decode: a cooperative environment for reverse-engineering legacy software. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 156–165. IEEE, 1995.

[54] Michael J. Rees. Automatic assessment aids for pascal programs. In *ACM SIGPLAN Notices Volume 17, Issue 10*, pages 33–42. ACM, 1982.

[55] Sally S. Robinson and M. L. Soffa. An instructional aid for student programs. In *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*, pages 118–129. ACM, 1980.

[56] Spencer Rugaber. The use of domain knowledge in program understanding. In *Journal Annals of Software Engineering, Issue Volume 9, Numbers 1-4*, pages 143–192. Springer Netherlands, 2000.

[57] Letovsky S. and Soloway E. Delocalized plans and program comprehension. In *Software, IEEE, Volume: 3, Issue: 3*, pages 41–49. IEEE, 1986.

[58] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of The 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*, pages 133–136, Canterbury, UK, 2001. ACM Press, New York.

[59] Jorma Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 37–39. IEEE Computer Society, 2002.

[60] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[61] S.D.Benford, E.K.Burke, and E.Foxley. Courseware to support the teaching of programming. In *Proceedings of the Conference on Developments in the Teaching of Computer Science*, pages 158–166. University of Kent, April 1992.

[62] Robert L. Sedlmeyer, William B. Thompson, and Paul E. Johnson. Knowledge-based fault localization in debugging: preliminary draft. In *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on high-level debugging*, pages 25–31. IEEE, 1983.

[63] R. E. Seviora. Knowledge-based program debugging systems. In *IEEE Software, Volume 4 , Issue 3*, pages 20–32. IEEE, 1987.

[64] M.-A.D. Storey, K. Wongb, and H.A. Müller. How do program understanding tools affect how programmers understand programs? In *Science of Computer Programming 36 (2000)*, pages 183–207. IEEE, 2000.

[65] Eleni Stroulia and Tarja Systä. Dynamic analysis for reverse engineering and program understanding. In *ACM SIGAPP Applied Computing Review archive, Volume 10 , Issue 1*, pages 8–17. ACM, 2002.

[66] Scott R. Tilley, Dennis B. Smith, and Santanu Paul. Towards a framework for program understanding. In *4th International Workshop on Program Comprehension*, pages 19–28. IEEE, 1996.

[67] A. Turing. On computable numbers with an application to the entscheidungsproblem. In *Proc. London Math. Soc. 42 (1936). Corrections appeared in: ibid., 43 (1937), pp. 544-6.*, pages 230–265, 1936.

[68] Michael J. Wise. Detection of similarities in student programs: Yap'ing may be preferable to plague'ing. In *ACM SIGCSE Bulletin*, pages 268–271. ACM, 1992.

[69] 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.

[70] S. Woods and Qiang Yang. The program understanding problem: analysis and a heuristic approach. In *18th International Conference on Software Engineering (ICSE'96)*, pages 6–15. IEEE, 1996.

[71] Steven Woods and Qiang Yang. Program understanding as constraint satisfaction. In *Seventh International Workshop on Computer-Aided Software Engineering (CASE'95)*, pages 318–327. IEEE, 1995.

[72] Steven G. Woods and Qiang Yang. Constraint-based program plan recognition in legacy code. In *Working Notes of the Third Workshop on AI and Software Engineering: Breaking the Toy Mold (AISE-95)*, 1995.