Chapter 10
Adaptive Testing in Programming Courses Based on Educational Data Mining Techniques

Vladimir Ivančević, Marko Knežević, Bojan Pušić and Ivan Luković

Abstract Designers of student tests, often teachers, primarily rely on their experience and subjective perception of students when selecting test items, while devoting little time to analyse factual data about both students and test items. As a practical solution to this common issue, we propose an approach to automatic test generation that acknowledges required areas of competence and matches the overall competence level of target students. The proposed approach, which is tailored to the testing practice in an introductory university course on programming, is based on the use of educational data mining. Data about students and test items are first evaluated using the predictive techniques of regression and classification, respectively, and then used to guide the test creation process. Besides a genetic algorithm that selects a test most suitable to the aforementioned criteria, we present a concept map of programming competencies and a method of estimating the test item difficulty.

Keywords Programming competencies · Concept maps · Test creation · Classification of test items · Genetic algorithms

Abbreviations

ARC	Area coverage
C1	Criterion1

V. Ivančević (✉) · M. Knežević · B. Pušić · I. Luković
University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21 000 Novi Sad, Serbia
e-mail: dragoman@uns.ac.rs

M. Knežević
e-mail: marko.knezevic@uns.ac.rs

B. Pušić
e-mail: bpusic@uns.ac.rs

I. Luković
e-mail: ivan@uns.ac.rs

C2	Criterion2
CAT	Computerized adaptive testing
CBA	Computer-based assessment
CR	Correct ratio
DF	Difficulty
DM	Data mining
EDM	Educational data mining
FIT	Fitness
FTS	Faculty of Technical Sciences
GA	Genetic algorithm
GC	Generation count
IC	Item count
IRT	Item response theory
M	Mean
MAX	Maximum
MDF	Mean difficulty
MF	Mean fitness
MGC	Max generation count
MH	Math
MIN	Minimum
MT	Mean completion time
NDF	Natural difficulty
NDFC	Natural difficulty category
OWL	Web ontology language
PAS	Past assignment
PLADS	Programming languages and data structures
PS	Population size
PTS	Past test
RA	Random approach
RDF	Resource description framework
SC	Student capacity
SCR	Student capacity rank
SD	Standard deviation
SDF	Standard deviation for fitness
SGC	Student group capacity
SK	Skewness
SPDF	Specified difficulty
SVM	Support vector machine
TPS	Test pool size
TS	Test
TSR	Test ratio
WRST	Wilcoxon rank sum test

10.1 Introduction

Computerized testing represents an area that has emerged together with the rising popularity of personal computers and their increased availability. With their introduction into schools and universities, a large number of students could be swiftly evaluated and graded using tests administered in computer classrooms. Furthermore, the switch to digital tests provides numerous benefits to both teachers and students.

The teacher's burden of grading each individual test using a same key is greatly reduced because the solution and grading process need to be specified only once while a computer may execute it as many times as necessary. Moreover, computer tests exist only in digital form, which eliminates the need for official storage space, as well as the time and costs associated with the copying of test forms. Additional advantage is that students may retrieve test results immediately after the completion of the test, thus obtaining timely feedback on their performance. Testing based on computers is suited primarily to tests with precisely defined structure and solutions. Multiple choice tests, which are in widespread use across different levels of education and research, fit this format very well.

Although this rigidity may appear to exclude many other forms of educational assessment and impose severe restrictions on the testing process, there have been many studies devoted to "intelligent" approaches to computerized testing that allow for very complex evaluation of student knowledge. The most important activity in the testing process, irrespective of the testing format, is the creation of a test. A test designer is responsible for forming a set of test items that encompass all relevant areas at the prescribed level of knowledge. This activity is sensitive to changes, as seemingly minor test modifications may cause noticeable differences in student performance. The knowledge gap between teachers and students, together with other distinctions between these two groups, may introduce additional difficulties into test creation. A teacher may assume that students have acquired necessary competencies, although the teaching process may not have been completely successful. For these reasons, we devise an approach that could overcome some of the aforementioned problems.

Our goal is to create software infrastructure for the computerized testing of programming knowledge that supports adaptation of tests to the competence of a target student group in a university course. We focus on the automatic creation of static tests for introductory university courses on structured programming, particularly the C language, and data structures. Tests created in this manner could be used as exercises tailored to a student group or even as finely tuned assessments determining the final grade. They should provide good separation of students into different classes according to their level of programming knowledge and be concise yet thorough in the examination. In this manner, generated tests could overcome the following problems:

- The bias of an examiner during the construction of tests.
- Long testing times, which is especially important in settings with limited technical facilities and many students.
- Unsatisfactory correspondence between the test results, and actual student understanding and competencies, which is usually caused by the unadjusted difficulty of the administered test.

The proposed approach is not necessarily restricted to computerized testing, but it may be best applied in such setting, since data required for test generation should be in electronic format. The identified problem is suitable for the application of educational data mining (EDM) [1] because the testing process may yield large data sets and the tuning of the test creation algorithm depends on the analysis of historical data concerning students and test items.

The foundation for the administration of such tests would be the Otisak testing system [2], which is a web-based software solution extensively used at the Faculty of Technical Sciences (FTS) in Novi Sad, Serbia. For this reason, the implementation of the proposed approach is tailored to the testing practice employed in an introductory programming course and information available at FTS. In the Otisak system, assessment of student knowledge is primarily conducted using multiple choice and short answer tests on programming. We intend to utilize assessment logs generated by this system that include student scores and copies of individual tests with recorded student answers. In the analysis of these logs, we may rank and evaluate test items [3] by utilizing classification algorithms.

Moreover, by mapping test items to programming language concepts, we form a concept-based foundation for the automatic creation of comprehensive programming tests. Information about previous student performance on related tests may be used in the additional refinement of tests, i.e., we mine available records with the goal of creating a student model [4].

This would allow for explicit acknowledgement of differences between specific groups of students during the construction of tests. In other words, testing may be considered adaptive because generated tests are suited to the actual competencies of the target groups of students. Therefore, the two key requirements that drive test generation are adequate coverage of the specified areas of competence and specified difficulty, which may be automatically calculated for a group of students.

In order to acknowledge both requirements, we create a genetic algorithm (GA) [5] that is designed to discover the best test with respect to the two criteria. Moreover, we also evaluate the efficacy of the generated tests [6, 7], by comparing them with those created using a random method generator. The research activities that lead to the implementation of the approach include:

- Creation of a formal model of programming concepts and competences that is suitable for a university course on programming (primarily for the C language).
- Mapping of previously used test items to programming concepts and competencies.
- Classification of test items according to their difficulty.
- Formal definition of the structure of a student profile.

- Estimation of current student competency.
- Creation of an algorithm for test generation that acknowledges student competencies (or explicitly specified difficulty), test item classes, and test comprehensiveness.
- Evaluation of tests created in this manner.

The rest of this chapter is organized in the following manner: in Sect. 10.2, an overview of the related work on computerized testing is given; in Sect. 10.3, background information about the testing method and the programming course is provided; in Sect. 10.4, a model of programming concepts and competencies is presented; in Sect. 10.5, we demonstrate how item difficulty and student competence may be predicted; in Sect. 10.6, the genetic algorithm for test creation is presented; in Sect. 10.7, an application of the proposed approach is illustrated; and in Sect. 10.8, the chapter is closed with concluding remarks and ideas for future research.

10.2 Related Work

Automated preparation and administration of tests in programming has become a necessity because competencies related to the computer and programming skills have become the norm in various disciplines and many professionals need to be rapidly trained. However, this field also has a relatively long history, as many custom solutions have been built over the past few decades.

As discussed in [8], computer-based assessment (CBA) brings many benefits to higher education. Contrary to the popular opinion, CBA allows various types of assessment, which are beneficial for students. The author illustrates how multiple choice tests, which are sometimes regarded as suitable for the evaluation of factual knowledge and too simple for advanced assessment, may be used to give complex problems to students. Their integration with randomization techniques, despite high initial costs to set up, could save significant time, especially in environments that are stable. In the context of the course from which the assessment data are retrieved for the study presented herein, multiple choice items have also proved to be a valuable tool when assessing more advanced competencies in programming. Thus, the proposed approach to generation of multiple choice tests could be viewed as a beneficial method of test creation, both in terms of the shortened design time and the reduced possibility of error, but under the condition that the individual test items are carefully designed.

Another group of valuable programming assessments are laboratory exams, where students have to write a working program in the computer laboratory, while being observed by the invigilators [9]. This also represents one of the applied methods in the course that we analyze. However, the authors also employ a special web system that is used in the whole assessment process, from the presentation of program specification, to program testing and storing of the final result.

More information about the rich history of automated assessment systems and techniques in computer sciences may be found in [10]. The authors also identify some of the problems associated with automated assessment and recognize the need for a human assessor, while the assessment systems should act as a support in the education process. A study on more recent advances in automatic assessment for programming exercises is presented in [11]. According to the authors, one of the big problems in this area is the lack of open solutions that may be freely applied by others. This leads to the proliferation of in-house solutions and the need to implement standard system features from scratch. The creation of a student test, together with the selection of adequate test items, is a delicate activity that is not so rarely based on intangible factors guiding the process. An important part of designing a test typically relies on the experience of a teacher/designer to estimate the difficulty of items.

Item Response Theory (IRT) [12] provides a powerful framework for test construction and tuning, in which items have a central role. The probability of a correct response to an item is modelled by a set of item parameters and depends on the examinee's ability. As a result, such approach allows for shorter assessment times, as well as adaptive testing, which represents assessment tailored to the individual ability. It has also boosted the development of computerized adaptive testing (CAT). However, despite the relatively long history and considerable research effort behind this theory, its strong requirements, difficult interpretation, and formal background have most likely been the reasons for its slow adoption among teachers. Owing to the static nature of tests and the lack of a dynamic assessment system in the analyzed course, we could not fully utilize the main benefits provided by IRT. This has partially motivated us to adopt a somewhat different approach to test adaptation that could be more acceptable for some educational settings.

Moreover, as reported in [3], Proportion Correct, a simple item difficulty estimate based on the proportion of correct answers, outperformed other more advanced estimates. By relying on the proportion of correct answers to estimate item difficulty, we defined an easily understandable set of item difficulty classes and a process in which items could be categorized. We also supported addition of new items, which is typical of the analyzed course, and allowed for immediate use of items without having to conduct experimental difficulty estimation. For this purpose, among several variables, we also utilized the expert estimate of item difficulty in the categorization of new items. During the creation of an item difficulty classifier, we corroborated the finding reported in [3] that the expert estimate is not always one of the best methods, since we found that the number of different concepts associated with an item correlates better with the proportion of correct answers as opposed to the expert estimate.

Formalizing the representation of knowledge in some area is another important element in the test design that allows for an approach which is less subjective and, most probably, less error-prone. Ontologies are typically used to represent concepts and their relationships in a domain. An educational ontology for the programming in the C language is presented in [13]. The authors also provide a number of guidelines on creating a “beautiful ontology”, i.e., one that should be clear, symmetrical, and well-organized. The actual ontology is publicly available in [14].

However, ontologies were originally devised to specify concepts that should be shared on the Semantic Web and automatically processed by computers. Such environment implies that a single ontology for some domain should be created, published, and shared by participants that are not necessarily known in advance. Unlike the aforementioned knowledge representations, our model of the programming knowledge is a particular solution, a concept map specifically created to match the requirements in a university course on programming and data structures. The omission of the required concepts would render our map incomplete, while the inclusion of concepts not discussed in the course would further complicate the map without any actual benefit for the people involved. Moreover, its primary users, teachers involved with the course, may not be that familiar with ontologies in general. Another benefit is that a concept map may be more readily comprehended and modified, if needed. More information about the ontologies and concept maps is given in the introductory part of Sect. 10.4.

10.3 Background

The creation of estimations of students and test items requires mining of logs [15] from the student testing system, which have to be parsed and imported into a specially designed database. Since the complexity of student models and item difficulty estimates depends on the richness of the extracted information, the proposed approach is primarily tailored to the quality of the available data. In order to implement and evaluate the approach, we relied on the testing logs collected during the organization of a university course on programming. More information about the environment from which the data originate, as well as about the data set, may be found in the following two subsections.

10.3.1 Environment

The data used in this study represent records associated with student tests that were organized as part of the Programming Languages and Data Structures (PLADS) course. This course is held at FTS (University of Novi Sad, Serbia), as a first year introductory course on programming for students of Computing and Control.

All the programming is taught using the C programming language, with the special attention devoted to the data structures. More information about the structure of the course may be found in [16].

The final grade in the course is determined by the score in pre-exam assessments, which are conducted in practical classes during the semester; and the score in the written theory exam, which is organized at the end of the semester. However, we restrict our analysis to the tests conducted during practical classes, which are held in a computer laboratory. There are two programming assignments, which require writing a C program, and typically three to four tests on programming.

The laboratory was specially designed for courses in computer science [17]. It features a computer-based student testing system named Otisak, which supports multiple choice, multiple response, and fill-in-the-blank test items.

Student activity during testing, together with test items and student answers, is recorded in the form of electronic logs and database entries. These records represent the main source of data to implement and test the approach proposed in this chapter.

10.3.2 Data Set

The data produced by the Otisak system is imported to a separate database that is used solely in the analysis of data about previous tests, student scores, and individual items. The database schema is shown in Fig. 10.1.

The stored data cover the period from 2008 to 2013. They are related to several computer science courses held at FTS, including the PLADS course.

There is basic information about each conducted test (table *Test*), assessed student (table *Students*), test items (table *Question*), test item options (table *QuestionAnswer*), and the corresponding course (table *Course*).

For each test, there are records describing which students took the test (table *StudentTakenTest*) and which test items (questions) were used (table *QuestionTest*). For each test item that a student was answering, there are records about the system events (table *AnswerLog*).

For test items, there are records about the covered knowledge areas (table *QuestionInfo*) and item difficulty (table *QuestionDifficulty*) with respect to some course.

There is information about 1,055 tests (including all courses, and many tests for the debugging and preparatory purposes), while 124 tests are related to the analysed introductory course on programming.

10.4 Modeling Programming Knowledge

In order to generate comprehensive tests about programming, basic information about a test item (e.g., for multiple choices items a designer provides a stem and a set of options) should be extended with the specification of important areas and competencies covered by that particular item.

The simplest solution would be to create an unordered list of areas and competencies, and to assign to each test item one or more list elements. Although the

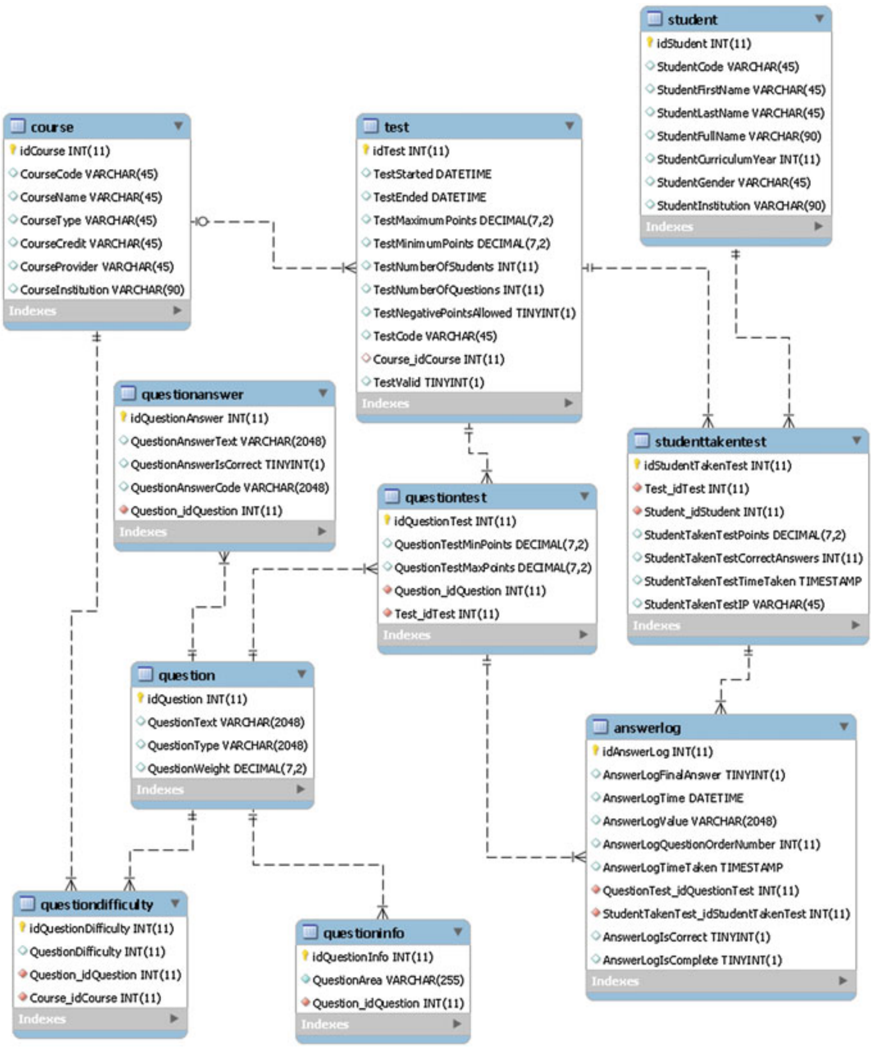
student
idStudent INT(11)
StudentCode VARCHAR(45)
StudentFirstName VARCHAR(45)
StudentLastName VARCHAR(45)
StudentFullName VARCHAR(90)
StudentCurriculumYear INT(11)
StudentGender VARCHAR(45)
StudentInstitution VARCHAR(90)
Indexes
course
idCourse INT(11)
CourseCode VARCHAR(45)
CourseName VARCHAR(45)
CourseType VARCHAR(45)
CourseCredit VARCHAR(45)
CourseProvider VARCHAR(45)
CourseInstitution VARCHAR(90)
Indexes
test
idTest INT(11)
TestStarted DATETIME
TestEnded DATETIME
TestMaximumPoints DECIMAL(7,2)
TestMinimumPoints DECIMAL(7,2)
TestNumberOfStudents INT(11)
TestNumberOfQuestions INT(11)
TestNegativePointsAllowed TINYINT(1)
TestCode VARCHAR(45)
Course_idCourse INT(11)
TestValid TINYINT(1)
Indexes
questionanswer
idQuestionAnswer INT(11)
QuestionAnswerText VARCHAR(2048)
QuestionAnswerIsCorrect TINYINT(1)
QuestionAnswerCode VARCHAR(2048)
Question_idQuestion INT(11)
Indexes
studenttakentest
idStudentTakenTest INT(11)
Test_idTest INT(11)
Student_idStudent INT(11)
StudentTakenTestPoints DECIMAL(7,2)
StudentTakenTestCorrectAnswers INT(11)
StudentTakenTestTimeTaken TIMESTAMP
StudentTakenTestIP VARCHAR(45)
Indexes
questiontest
idQuestionTest INT(11)
QuestionTestMinPoints DECIMAL(7,2)
QuestionTestMaxPoints DECIMAL(7,2)
Question_idQuestion INT(11)
Test_idTest INT(11)
Indexes
question
idQuestion INT(11)
QuestionText VARCHAR(2048)
QuestionType VARCHAR(2048)
QuestionWeight DECIMAL(7,2)
Indexes
answerlog
idAnswerLog INT(11)
AnswerLogFinalAnswer TINYINT(1)
AnswerLogTime DATETIME
AnswerLogValue VARCHAR(2048)
AnswerLogQuestionOrderNumber INT(11)
AnswerLogTimeTaken TIMESTAMP
QuestionTest_idQuestionTest INT(11)
StudentTakenTest_idStudentTakenTest INT(11)
AnswerLogIsCorrect TINYINT(1)
AnswerLogIsComplete TINYINT(1)
Indexes
questiondifficulty
idQuestionDifficulty INT(11)
QuestionDifficulty INT(11)
Question_idQuestion INT(11)
Course_idCourse INT(11)
Indexes
questioninfo
idQuestionInfo INT(11)
QuestionArea VARCHAR(255)
Question_idQuestion INT(11)
Indexes

Fig. 10.1 The test log database schema

creation of such list would not require a special set of skills except possessing domain expertise, the final product would be largely impractical as any non-trivial domain or subdomain typically features at least several hundred concepts.

As a result, navigating a large unordered list in search for an adequate concept would be very time-consuming and tiring for a domain expert. Furthermore, such representation of the domain may not adequately transfer domain knowledge to non-experts, namely students who could only benefit from the access to the domain model.

On the other hand, ontologies and standards associated to the Semantic Web, such as Resource Description Framework (RDF) [18], Web Ontology Language (OWL) [19], and OWL2 [20], could be utilized as a formal basis for the description of a domain and its concepts with the added benefit of having functional semantic reasoners. Approaches based on the Semantic Web have many proponents as numerous applications, tools and new versions of standards are continuously being developed.

The OWL ontology specification language, which actually includes three sublanguages (Owl Lite, OWL DL, and OWL Full), is a formally defined text-based language. Although the availability of the three sublanguages was expected to offer significantly different levels of expressiveness, numerous problems persisted, which led to the creation of OWL 2 [21]. However, the formality, textual nature, and web orientation of these languages may be the biggest obstacles that a domain expert should overcome before successfully using them. The domain expert should be knowledgeable about classes, properties, and data types, as well as be aware of semantic implications associated with the concrete ontology design. Moreover, the textual syntax may be quite cumbersome for human users.

There are many visualization solutions for OWL ontologies [22–24], however, the OWL languages were not primarily created to be human readable. One possible solution to the problem of flexible domain description may be the use of concept maps [25], which are diagrams featuring concepts and relationships between them. The graphical nature of concept maps represents a benefit when describing knowledge, as these maps were invented in order to reflect mental processes and associative relations between concepts.

Some positive effects of directly using concept maps as means of student assessment in schools have been observed [26], but numerous challenges to their general adoption in assessments still remain [27]. Moreover, CmapTools [28] is a software tool that was developed to allow knowledge modeling and sharing using concept maps. It supports map export to text propositions, CXL format (an XML representation of the concept map diagram), numerous image formats, etc.

One of the benefits of using concept maps is that they may be freely created to capture relevant knowledge, unlike OWL ontologies, which are restricted by numerous rules as they are expected to be interpreted by computers. This freedom to create arbitrary concepts and relationships without the overhead caused by many formalisms also facilitates communication between domain experts who are creating a concept map, as well as between students who are exploring the modeled domain.

The structure of the map diagram is not limited to tree structures but allows for any graph, i.e., several connections may point to a single concept. In case the map needs to be programmatically processed, its CXL representation may be parsed. For these reasons, we have opted for a concept map to express the programming knowledge. We primarily require that the knowledge representation may be relatively easily created and understood by a layperson, as well as programmatically accessed and analysed when estimating individual test items. In the remainder of this section, we propose a model of programming competencies and concepts, which was created using the CmapTools software.

10.4.1 Programming Knowledge Overview

In order to simplify the concept map and its parsing, a tree structure of concepts was created by a teaching assistant from the analyzed course, together with a set of auxiliary links that are not parent–child links, which are known as cross-links. In the resulting knowledge model, excluding the cross-links, each concept may have zero or more child concepts. On the other hand, each concept has exactly one parent concept, save for the root concept, which does not have a parent. A link between concepts (typically marked with a noun) includes a linking phrase (marked with a verb) and a set of connections between the concepts and the linking phrase.

The resulting concept map was created to match the knowledge outcomes of the PLADS course organized at FTS. It includes 309 concepts, 93 linking phrases, and 401 connections. It has a single root concept labeled *Programming in C*. All of the relevant concepts are divided into two groups: one containing concepts related to general competencies associated with structured programming and the other containing specificities of the C language. The root concept of the first group is *Programming Competencies*, while the second group starts from the concept *C Language Elements*. Between the two subtrees starting from the two abovementioned concepts, there are many crosslinks matching programming competencies with concrete constructs and keywords of the C language. The first two levels of the concept map are presented in Fig. 10.2. The concept map represents a solution that has been created for a concrete university course, as well as the test generation problem which we are attempting to solve.

This is mainly evident from the structure of the *Programming Competencies* subtree, which includes some common programming principles embodied in *Algorithmic Thinking* and *Code Styling*. Although these principles are universal, the exact structure and level of detail with which they are presented to students may differ between institutions. Moreover, teachers are expected to design a course according to the restrictions imposed by a study program, such as class duration and frequency.

For these reasons, some teachers may find the concept map too detailed or even limited in scope, depending on the objectives of the programming courses that they teach. On the other hand, the *C Language Elements* subtree contains the technical

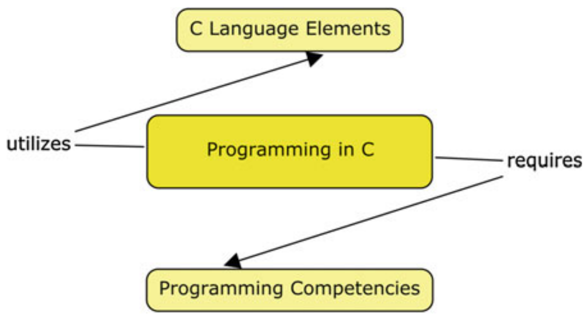

Fig. 10.2 The two initial levels of the programming knowledge model

terms from C that are standardized, which makes this portion of the map reusable in different institutional and educational contexts.

In the following subsection, some of the general concepts from the two subtrees are presented. In the model, each concept that denotes a subtree corresponding to a module graded in the practical portion of the programming course has an underlined label—there are 19 such concepts in total. In order not to clutter the diagrams with numerous connections, cross-links are excluded from the provided map excerpts.

10.4.2 Modeling Programming Competencies

Programming competencies are organized by areas, such as *Variable Manipulation*, *String Manipulation*, *Flow Control*, *Command Line Arguments*, *Numeral Systems*, and *Data Structures* (see Fig. 10.3). They typically correspond to the mastery of a basic command (or a set of them) that have similar outcomes in different programming languages. The presented portion of the concept map includes skills that are typical of the structured programming paradigm and mostly transferable between various languages, e.g., C, C++, and Java. The listed concepts are further decomposed. For example, the *Data Structures* concept encompasses 29 other concepts.

The majority of these competencies are graded (13 in total), while student knowledge in areas *Algorithmic Thinking* and *Code Styling*, which are taught and encouraged throughout the course, is not explicitly assessed during computerized testing.

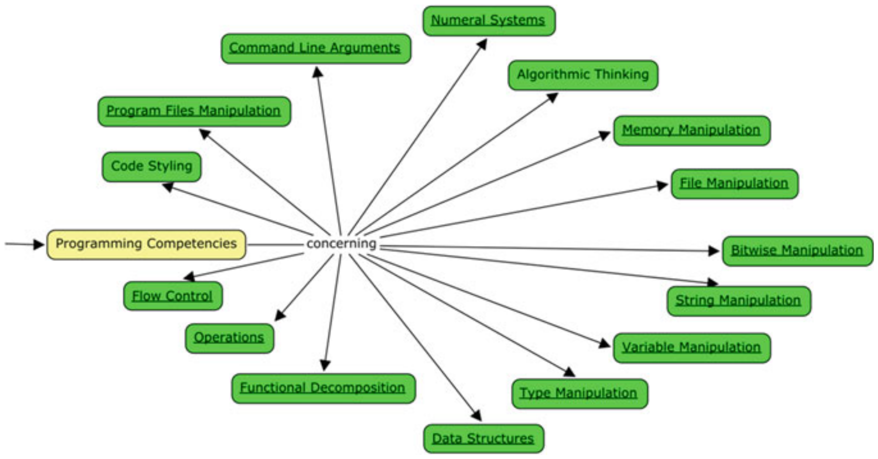

Fig. 10.3 The two initial levels of the programming competencies subtree

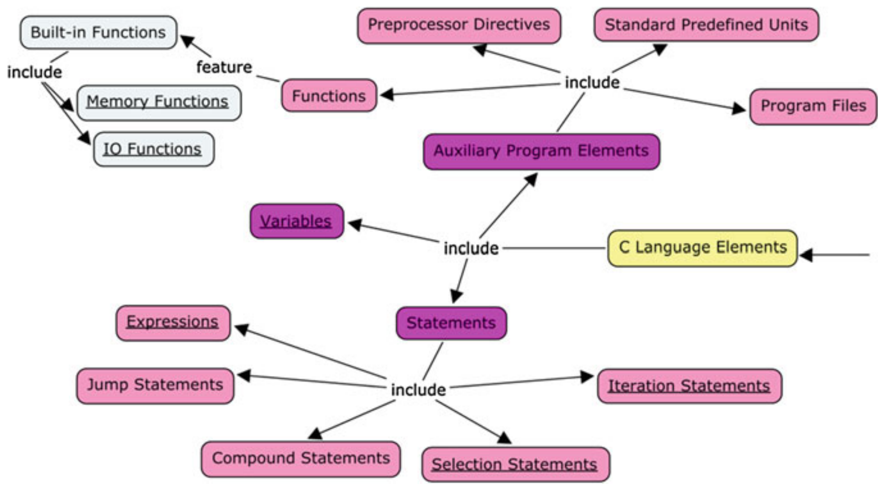

Fig. 10.4 The most important concepts in the C Language elements subtree

10.4.3 Modeling Programming Concepts of the C Language

We organized the key constructs and capabilities of the C language into three broad categories: Variables, Statements, and Auxiliary Program Elements. Some of the most prominent members of these categories are presented in Fig. 10.4.

There are in total 6 key areas belonging to the *C Language Elements* sub tree of the knowledge model (underlined in Fig. 10.4), which are thoroughly covered in student assessments: *Variables*, *Expressions*, *Selection Statements*, *Iteration Statements*, *IO Functions* and *Memory Functions*.

The suggested "taxonomy" should not be viewed as a strict scientific overview of a generic programming language, but as a model of the features of the C language, as presented in the educational context of an introductory course on programming. Many of the directly linked concepts do not conform to the "is-a" relationship. They rather represent arbitrary associations between concepts, in the way they may be mentally formed by students during classes and individual study. As a result, a concept may be further linked to other concepts that designate subclasses, properties, or behaviour of their parent concept. An example of these relationships may be observed for the *Variable* concept (see Fig. 10.5).

For instance, program variables may be classified as plain, pointer, array, or matrix variables. Each variable has several properties: scope, address, value, name, and type. A variable may be declared or initialized. However, not all information about variables in C is covered by the *Variable* subtree of the model.

There may be a cross-link to the *Statements* subtree, namely the *Expressions* concept, whose subtree is the most populous in the model. *Expressions* in C may involve variables, thus specifying additional operations applicable to variables. Due to the complexity of many concepts, information about a single concept

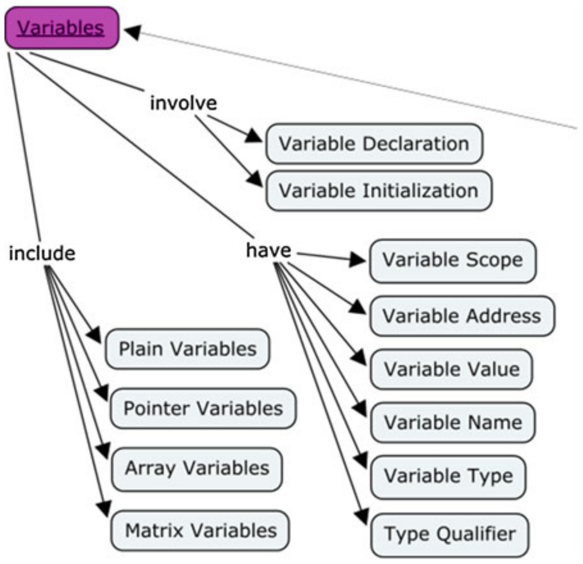

Fig. 10.5 The most important concepts in the variables subtree

cannot be contained within a single subtree. In order to create an intuitively understandable model, the model designer has to choose global demarcation lines between concept groups, which may be connected using cross-links when needed.

10.5 Estimating Test Difficulty

Student performance in an assessment is primarily influenced by the requirements level of the assessment (test difficulty) and the competence of the assessed students. When creating a new test, both factors should be taken into account. The test should cover all topics relevant for the particular assessment and have an adequate level of difficulty, as determined by the difficulty of individual test items. However, these two requirements are not easy to fulfill in practice. The testing time is often limited by the available time during regular classes, which restricts the number of items in the test and, consequently, the test comprehensiveness.

Furthermore, teachers may not always correctly estimate the difficulty of a single item. They may misjudge the knowledge of students, thus creating a test in which students perform very bad or very well. An algorithm for automatic test creation, which attempts to overcome these problems, utilizes available data about items, students and previous tests, in order to generate a test satisfying the aforementioned conditions. By performing educational data mining, we estimate the difficulty of items and future performance of a group of students who need to be assessed.

As a result of the process, the predicted values may be passed to the test generation algorithm, which further uses them as guidance during the automatic selection of items for a test. Once the selection of items is finished, the resulting test may be administered to the target students. In this manner, we obtain a test with the required number of fixed items that is tailored to the estimated ability of a student group as a whole and not to the individuals. In the following two subsections, we demonstrate how item difficulty and student performance may be represented and predicted in the context of the PLADS course at FTS.

10.5.1 Estimating Test Item Difficulty

Tests used in the practical assessment of students in the analyzed course primarily include multiple choice questions as items. In order to build a test of the required difficulty, for each possible item there should be a numerical estimate of its difficulty. With this information, the test difficulty could be calculated as the arithmetic mean of the difficulty values matching the pertaining items.

The individual item difficulty is neither explicitly modeled nor evaluated differently for each student, because, in the analyzed course, a test is always administered to a student group and each student receives the same set of items that cannot be modified once the test has started. The simplest solution would be to calculate item difficulty using the percentage of correct answers for an item in the past tests. However, there are two problems with such solution.

The first problem is that the proposed solution is possible only when there are sufficient records in the assessment log about each item. For the analyzed course, we observed that the percentage of correct answers for an item may increase if that item is often repeated in different assessments, most probably because assessed students readily share information about the completed tests with their peers. For this reason, new items are constantly being added to the item pool. Nonetheless, these items do not have their own percentage of correct answers and, hence, their difficulty cannot be estimated. In case that the item's difficulty is unknown, there is a non-negligible risk that a new test item might be too difficult (or easy) for students, which may lead to an unjustified change in scores and overall student performance. The second problem is related to the meaning of the difficulty estimate. The percentage of correct answers for an item is an exact value, but for teachers this value alone may not be sufficient to understand the difference between items with respect to their difficulty or know exactly for which percentage of correct answers the item becomes difficult. In other words, there is no suitable interpretation of these values for the purpose of test creation.

In order to remedy the identified problems, in this subsection, we present how the difficulty of test items may be represented to offer a more manageable interpretation for teachers. Furthermore, we also demonstrate how the newly defined difficulty may be estimated for an item in two scenarios: when the item has been extensively used in past assessments and when the item is previously unknown or rarely used. For the purpose of illustrating the estimation process, as well as the identification of important variables related to item difficulty, we have selected a sample of 172 items from the test records database.

All of the items were annotated with the matching programming concepts defined in the model from Sect. 10.4. As demonstrated in this section, concepts mapped to an item also provide valuable information about the item's difficulty. The individual items were chosen so that they reflect different type of programming questions which typically appear in student assessments. For each item, there is also the percentage of correct answers (*CorrectRatio*, *CR*) recorded during past assessments, which is the key variable in the estimation of item difficulty.

In order to add the meaning to the difficulty of an item that has been extensively used, we decided to first simplify the estimation problem, by discretizing the percentage of correct answers. The percentage range for the analysed sample is automatically divided into three intervals using the Jenks natural breaks classification method [29], whose implementation is available in R environment for statistics and data analysis [30]. As a result, we obtain the categories for the difficulty of test items (*NaturalDifficultyCategory*, *NDFC*). The generated partition, including the meaning for each category, is shown in Table 10.1, while the histogram for *CR* is presented in Fig. 10.6.

As our aim was to provide a limited number of difficulty levels that are sufficiently separated but enclose a comparable number of items, according to the distribution of *CR* values, we defined exactly three readily interpretable categories, where easy items are denoted by 1, items of moderate difficulty by 2, and difficult items by 3.

In the automatic calculations involving item difficulty, we are using more precise (and more informative) values in addition to the three integers. For an item *it* with a known value of *CR*, the exact value of *NaturalDifficulty* (*NDF*), which lies inside the interval [0.5, 3.5], may be calculated using the formula (10.1):

$$NDF(it) = ndc - 0.5 + (U_{ndc} - CR(it))/(U_{ndc} - L_{ndc}), \tag{10.1}$$

where ndc is the natural difficulty category of the item it, L_{ndc} the lower bound of the item's CR for the category ndc, and U_{ndc} the upper bound of the item's CR for the category ndc. In this manner, if the CR value of an item is the midpoint of the interval defined by L_{ndc} and U_{ndc}, the corresponding NDF is equal to the item's $NDFC$, while the NDF for the other CR values typically falls somewhere between the integer values matching the three categories.

Table 10.1 Natural segmentation of the percentage of correct answers

Natural difficulty category	Correct ratio range		Number of items in the category
	Lower bound	Upper bound	
1 (Many correct answers)	0.76 (exclusive)	1 (inclusive)	50
2 (Majority of correct answers)	0.4688 (exclusive)	0.76 (inclusive)	65
3 (Minority of correct answers)	0.0323 (inclusive)	0.4688 (inclusive)	57

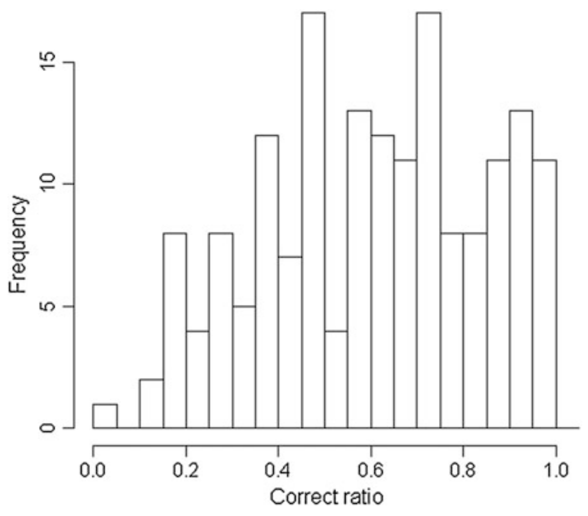

Fig. 10.6 Histogram for the ratio of correct item response

In the estimation of the difficulty of a new or rarely used item, i.e., when the item's *CR* value is unknown or not representative of the item's difficulty, we predict *NDFC* of an item using other item-related variables. In this scenario, the exact difficulty (*NDF*) is not predicted but only the corresponding category (class) due to the predictive quality of the available variables.

The simplest solution would be to have an expert give a difficulty estimate for each new test item (*ExpertDifficulty*). For this purpose, a teaching assistant from the analysed course rated all available test items on an integer scale from 1 to 3, where 1 denotes an easy item while 3 denotes a difficult item.

The principal criterion for assigning labels was the complexity of the problem presented in a test item. An item is considered easy (label 1) if it requires basic reproduction of some piece of information presented in the course, or an analysis of a simple statement written in C. A moderate item (label 2) is the one that typically combines from two to four language constructs (or a combination of practical and theoretical ideas) and requires an analysis of their interaction or relationship.

A difficult item (label 3) requires a careful analysis of a program code featuring a combination of advanced concepts (typically including pointers) with complex flow controls and strong dependencies between presented code sections.

An overview of the expert classification of test items is given in Table 10.2. For each difficulty category, we present the label, number of items from the sample that are assigned to that particular label, and examples of what is typically being evaluated in that category.

However, the expert estimate of the difficulty of an item has a weak-to-moderate correlation with *CR*, which is closely related to *NDFC*, and the items are less evenly distributed between the categories as opposed to *NDFC*. Therefore, we included other item-related variables in the estimation, for which we hypothesized

Table 10.2 Overview of categories associated with expert difficulty estimation

Expert difficulty category	Number of items in the category	Examples
1—Easy	42	• Use of terminal • Basic knowledge of types • Analysis of simple selection statements • Analysis of simple iteration statements • Understanding of memory addresses • Basic command of memory functions • Basic operations on arrays • Basic operations on strings • Basic operations on files • Basic understanding of C structures • Calculation of memory requirements for concrete data storage • Understanding of numeral systems (*binary*, *octal*, *decimal*, and *hexadecimal*)
2—Moderate	99	• Use of pointers • Advanced use of expressions (in selection and iteration statements) • Advanced string operations • Bitwise operations (operators and masks) • Functions (declaration, definition, calling) • Advanced knowledge on types (duality of certain types and type conversion) • Use of command line arguments • Use of memory functions for custom data structures • Analysis of moderately complex code
3—Difficulty	31	• Analysis of very complex code featuring competencies from all categories

that there should be a positive relation to item difficulty. For each item in the analyzed sample, we calculated three additional values:

- AreaCoverage$_{\mathrm{all}}$ (ARC$_{\mathrm{all}}$). A number of *all* 19 relevant areas from the knowledge model (underlined concepts in the model from Sect. 10.4) that are at least partially covered by the item, i.e., the item is associated to a concept within one of the 19 relevant subtrees.
- StemLength. A number of characters in the stem, where an occurrence of multiple consecutive whitespace characters counts as a single character.
- KeywordCount. A number of the reserved words in the C language that appear in the stem.

In Table 10.3, for the three aforementioned variables and the expert difficulty estimate, we present mean (*M*), standard deviation (*SD*), minimum (*MIN*), maximum (*MAX*), skewness (*SK*), and Pearson's correlation coefficients with respect to *CR*. With respect to correlation with *CR*, the best variable is AreaCoverage$_{all}$,

Table 10.3 Summary statistics and correlation for predictor variables

Variable	M	SD	MIN	MAX	SK	Correlation to CR
ExpertDifficulty	1.936	0.65	1	3	0.063	−0.276
$\text{AreaCoverage}_{\text{all}}$	0.142	0.069	0.1	0.26	0.238	−0.392
StemLength	175.878	55.32	74	325	0.464	0.032
KeywordCount	4.395	2.306	0	12	−0.105	−0.150

which surpasses even the expert estimate (*ExpertDifficulty*), the second best variable.

On the other hand, *KeywordCount* exhibits weak correlation, while *StemLength* appears to be linearly unrelated to CR. As a result, the estimation of the difficulty of an item without its *CR* value is performed using a classifier that is created including the following independent variables:

- ExpertDifficulty.
- $\text{AreaCoverage}_{\text{all.}}$
- StemLength.
- KeywordCount.

The only dependent variable is:

- NDFC.

Although the preliminary analysis of variables *StemLength* and *KeywordCount* does not indicate that they have significant predictive quality, they are included in the classifier because their presence offered small improvements in the prediction rate, as discovered in the initial experiments with the classifier.

For illustrative purpose, after the data preparation phase on the set of the 172 test items, we tested four different types of classifiers, which are available as part of the R environment. The classifier performance was evaluated with respect to the training error, cross-validation error (for 10 folds and 100 experiments), and Fleiss' kappa.

The results are presented in Table 10.4. The support vector machine (SVM) classification using the Crammer-Singer native multi-class method [31] has the lowest errors overall and the top kappa value. The best results for this algorithm are obtained using the radial basis function (Gaussian) kernel with parameters $C = 80$ (the parameter in the cost function) and $sigma = 80$ (the kernel function

Table 10.4 Classifier performance on the testing set

Classifier	Training error	Cross-validation error	Kappa
Support vector machine	0.081	0.490	0.877
K-nearest neighbor	0.081	0.515	0.877
Decision tree	0.267	0.531	0.597
Naive Bayes	0.467	0.506	0.277

parameter), as indicated by the results of a two-step grid search for good parameter values.

The first error value, which most probably is lower than it could be expected for a set of new items, illustrates how much the model is misclassifying the training data. The second error value, which is calculated when performing the k-fold cross validation, generally provides a more realistic perspective on the performance of the classifier on new data. However, this error describes a classifier trained without using one kth of the potentially valuable data. The kappa value indicates the agreement with the used data over the one expected by chance.

Given all the aforementioned information, the difficulty estimate (*Difficulty*, *DF*) for any available item may be made in the following manner (10.2):

$$DF(it) = \begin{cases} 3, & valid\ CR(it) \wedge NDF(it) > 3 \\ NDF(it), & valid\ CR(it) \wedge 1 \leq NDF(it) \leq 3 \\ 1, & valid\ CR(it) \wedge NDF(it) < 1 \\ NDFC(it), & not\ valid\ CR(it) \end{cases} . \tag{10.2}$$

When there is a representative *CR* value for an item, we may calculate a more precise estimate, while, for all the other cases, the trained SVM classifier is utilized to estimate the item difficulty by predicting the item's *NDFC*.

By using the difficulty estimates of available items, we may evaluate the student capacity to do well in assessments with respect to item difficulty, as well as parameterize the test generation algorithm to construct a test matching the capacity of a group of students.

10.5.2 Estimating Student Capacity

In the proposed approach, the term student capacity denotes the ability of a student to perform well in a given course, i.e., achieve good scores in the assessments conducted by course teachers. There are three primary requirements when estimating student capacity for the purpose of test creation. First, there should be a predictive model for student scores in programming. Second, there should be a measure of student capacity that is related to student scores and has meaning for teachers, similarly to the case of item difficulty from the previous subsection. Third, there should be a clearly defined relationship between the measure of student capacity and measure of item difficulty because such relationship would allow direct comparison between the capacity of a student group and the difficulty of a test with its pertaining items. In this manner, the quality of an automatically generated test could be evaluated with respect to the capacity of the target student group. In the remainder of this subsection, we present our solutions to the three issues.

It is difficult to predict the exact performance of students in tests that are held during the PLADS course because the course is organized in the winter semester (the first semester) of the first year of study, when there are still almost no data about the academic achievement of students. For the initial portion of the semester, the only available information includes student scores from the university entrance exam. Given the fact that the entrance exam is used to evaluate the proficiency in mathematics, the resulting student score is not directly related to the score in a programming test. However, there is moderate positive correlation between the score in the entrance exam on mathematics and score in the programming tests in the analyzed course.

The value of the Pearson's correlation coefficient for the two variables in the academic year 2011–2012 is 0.477 (0.475 for the Spearman's correlation coefficient). Moreover, once the initial programming assessments are completed, it is possible to improve capacity prediction by utilizing student scores from the already completed programming tests and assignments.

Owing to the preliminary findings, we opted for the multiple linear regression as means for estimating student score in some of the n_a programming tests that are conducted during a semester. The independent variables include the entrance exam score in mathematics (*Math*, *MH*), the total score in the first i programming assignments (*PastAssignment*, *PAS*), and the total score in the first i programming tests (*PastTest*, *PTS*), while the dependent variable is the total score in the $n_a - i$ remaining tests in the programming course (*Test*, *TS*). The i value is an integer from $[0, n_a - 1]$. For a student s, the regression model is of the following form (10.3):

$$TS(s,i) = \beta_3(i) \cdot PTS(s,i) + \beta_2(i) \cdot PAS(s,i) + \beta_1(i) \cdot MH(s) + \beta_0(i), \quad (10.3)$$

where β_3, β_2, and β_1 are the regression coefficients for the predictors *PTS*, *PAS*, and *MH* respectively, while β_0 is the intercept. The regression coefficients and intercept depend on the number of already completed tests in a semester, which is marked by i. Because each test marks a milestone during a semester, a semester part is defined as a period between two consecutive tests, including the special case of a period before the first test.

Therefore, an i value also denotes the $(i + 1)$th part of the semester. As students complete programming tests, i.e., progress from one semester part to the next one, there is more information about students' programming knowledge. This information, which is present in the cumulative test scores, may be used to predict performance in the remaining tests. However, it also changes throughout the semester.

As a result, for each part of the semester, we utilize a different regression formula with its own set of values of the coefficients and intercept. For illustrative purpose, in Table 10.5, we present information about all regression formulae for 144 students enrolled in the winter semester of the academic year 2011–2012, when there were three programming tests ($n_a = 3$).

The values of the regression coefficients, together with the allowed ranges for *TS*, *PTS*, *PAS*, and *MH*, indicate that, as a semester progresses (i increases), the *PTS* variable becomes more important and the variables *PAS* and *MH* less

Table 10.5 Details about regression formulae for student programming performance

i	Allowed range				β_3	β_2	β_1	β_0	R^2	p-val
	TS	PTS	PAS	MH						
0	0–40	/	/	0–60	/	/	0.230	21.253	0.198	$<10^{-7}$
1	0–30	0–10	0–15	0–60	0.387	0.342	0.087	13.507	0.271	$<10^{-7}$
2	0–10	0–30	0–15	0–60	0.198	0.090	0.022	−0.147	0.228	$<10^{-7}$

important in the prediction. This is evident primarily from the more rapid decrease in the values of β_2 and β_1 as opposed to β_3.

Other regression models were also evaluated with respect to Eq. (10.3): models excluding the *MH* predictor, models with the added squared term for *PTS* and/or *PAS*, and models with the added interaction between *PTS* and *PAS*. Nonetheless, when compared to the original model in terms of the residual standard error, the other models performed worse or, for certain *i* values, equally well but with the added burden of unnecessary terms.

In order to link the predicted *TS* value for a student to the matching difficulty of test items, we introduce auxiliary variables. For (10.4) a non-empty set of n_{items} test items *Items*, let:

$$rank : Items \rightarrow \{1, \ldots, n_{items}\}, \tag{10.4}$$

be a ranking function that assigns to each item a different integer, so that an item *it* has a lower rank value when compared to all the other items with a higher *CR* value, while the ranks of items with equal *CR* may be ordered according to items' identifiers within the database presented in Sect. 10.3.2. The formula (10.5) for calculating *TestRatio* (*TSR*) is:

$$TSR(s,i) = TS(s,i)/TS_{\max}(i), \tag{10.5}$$

where $TS_{max}(i)$ is the maximum allowed value for *TS*(*s*, *i*), which is given in Table 10.5 for different *i* values.

The formula (10.6) for *StudentCapacityRank* (*SCR*) is

$$SCR(s,i) = \begin{cases} ceil(n_{items} \cdot (1 - TSR(s,i))), & TSR(s,i) \neq 1 \\ 1, & TSR(s,i) = 1 \end{cases}. \tag{10.6}$$

For each student, we may estimate the DF value corresponding to the student's capacity by using the Eq. (10.2) in the Eq. (10.7) for *StudentCapacity* (*SC*):

$$SC(s,i) = DF(rank^{-1}(SCR(s,i))). \tag{10.7}$$

For a non-empty set of n_s students *Students*, the *StudentGroupCapacity* (*SGC*), which is the difficulty of a matching test for that student group (10.8), is within the [1, 3] range and calculated as the arithmetic mean of individual values of *SC*:

$$SGC(Students, i) = (1/n_s) \cdot \sum_{s \in Students} SC(s, i). \tag{10.8}$$

Equations (10.4–10.8) formally describe a process responsible for matching student capacity to item difficulty. The predicted student performance is expressed as a ratio (*TSR*) between the predicted score and the maximum score in Eq. (10.5). For a given student performance, there is a matching rank *SCR* from 1 (best) to n_{items} (worst). The process of transforming the student performance ratio to its rank is started by sorting existing test items in the decreasing level of difficulty, as expressed by *CR*, using the rank function from Eq. (10.4). Next, the bottom *TSR* percentage of ranked items is removed and the rank of the least difficult item remaining (or the most difficult item removed) becomes the rank matching the student capacity, which is expressed in Eq. (10.6).

Finally, the student capacity is mapped to the difficulty of the item with the same rank in Eq. (10.7). For example, if a student's predicted score ratio in a test is 0.9 (90 %), then the student is expected to give a correct answer for 90 % of items, on the average. For this scenario, the student's capacity equals the difficulty of the item that separates the top 10 % of items from the rest. In this manner, for each student, the corresponding capacity is defined as the difficulty of the most difficult item for which that student is expected to give a correct answer.

10.6 Test Generation Algorithm

The test generation algorithm is designed as a genetic algorithm that searches for a combination of test items, which, as a group, should cover as many specified areas as possible, but at the same time be as close to the specified mean difficulty as possible. As the stated problem belongs to the field of multi-criteria optimization, in this case exhibited by the need to attain the specified coverage and difficulty, genetic algorithms are chosen as the method of test construction. Therefore, if a set of available test items, together with a set of arguments, is passed to the algorithm, the output is a combination of test items matching the aforementioned criteria and argument values.

Once the test generation process is finished, we may conduct an assessment, in which all target students have to give answers for the same items within the generated test. As with any genetic algorithm, there are several common steps. First, a random group (*population*) of solutions (*individuals*) is generated and set as the current population. Next, the population evolves through the specified number of iterations (*generations*), with the possibility of terminating the process early if the population becomes homogenous (becomes entirely composed of similar or same solutions) or a solution of a desired quality is found.

In each generation, some individuals are selected (*selection*) according to their *Fitness* (*FIT*), which represents the quality of an individual and is calculated using a custom fitness function. The selected individuals enter the phases of *crossover* (two or more individuals are combined to generate a new individual) and then *mutation* (selected new individuals are randomly modified), thus producing new individuals, which, as a group, are generally expected to be better than the current population. The new individuals comprise a new population, which replaces the current population and enters the next iteration. The concrete elements that have to be specified in this generic procedure include:

- Algorithm arguments.
- Structure of a solution.
- Selection process.
- Crossover process.
- Mutation process.
- Fitness function for the solution.

The following algorithm arguments are required in the proposed approach:

- Items. The set of potential test items, where each item has an identifier, difficulty, which is calculated using Eq. (10.2) from Sect. 10.5.1, and list of knowledge areas that it covers.
- ItemCount (IC). The exact number of chromosomes (test items) in individuals, i.e., the desired number of items in the generated test.
- SpecifiedDifficulty (SPDF). The student group capacity of students who would take the generated tests, which may be manually set to a value from [1, 3] or calculated using Eq. (10.8) from Sect. 10.5.2.
- ConceptMap. The concept model containing all knowledge areas (presented in Sect. 10.4).
- Required. The set containing knowledge areas (presented in Sect. 10.4) that needs to be covered by the generated test.
- PopulationSize (PS). The size of the population, i.e., the exact number of individuals within the population (preferably an even number).
- MaxGenerationCount (MGC). The maximum number of generations, after which the algorithm should terminate.
- FitnessThreshold. The minimum fitness measure that leads to the termination if observed in an individual (the observed individual is considered the best solution).
- Convergence. The maximum deviation in mean population fitness over a specified number of latest generations that leads to the termination.
- CrossoverCount. The number of chromosomes that will be exchanged between individuals during the crossover phase.
- MutationChance. The chance that a mutation will occur in an individual.
- Elitism. Indicator about whether to allow elitism, which is a process when best fitted individuals (elites) skip the crossover phase and directly enter the new population.
- ElitismMutation. The indicator about whether to allow elitism mutations, which is a case when an elite individual, who directly passes to the new population, also undergoes the mutation phase.

Each individual contains an array of different test item identifiers. The length of the array is equal to the chromosome count specified before algorithm execution. Before the crossover phase, pairs of individuals are randomly formed, where the number of pairs is equal to one half of the population size. However, since the proposed algorithm employs roulette wheel selection, fit individuals, which have a high fitness value, are more likely to transit to the crossover phase and, consequently, propagate their chromosomes (a set of test items) to the next generation.

During the crossover, the specified number of chromosomes (item identifiers) is swapped between the individuals who were coupled in the selection process. In the mutation phase, given the initially specified mutation chance, a randomly selected value corresponding to a valid test item identifier is set as a new value of a single chromosome. The target chromosome is either selected randomly or it represents an item whose difficulty varies the most from *SPDF* within the individual. The fitness function is one of the key elements in the algorithm. For a set of test items encompassed by the individual, the fitness function provides a numerical indicator of the quality of the test that would contain these items. It takes as input an individual *ind* and calculates to which extent the two criteria are satisfied:

- Criterion1 (C1). The required knowledge areas (*Required*) are covered by the items within the individual *ind.*
- Criterion2 (C2). The mean difficulty of the items within the individual *ind* matches the specified student group capacity (*SPDF*).

The first criterion is expressed by the following formula (10.9):

$$C1(ind) = ARC_{required}(ind)/n_{required}, \tag{10.9}$$

where *ind* is an individual (a set of items), $ARC_{required}$ the number of the required areas from the *Required* set (an argument passed to the algorithm) that are covered by the individual *ind*, and $n_{required}$ the total number of the required areas (the cardinality of the *Required* set). The second criterion is expressed by the following formula (10.10):

$$C2(ind) = (2 - |MDF(ind) - SPDF|)/2. \tag{10.10}$$

where *MeanDifficulty* (*MDF)* is the mean difficulty for the test items enclosed within the individual *ind*, and *SPDF* the desired difficulty of the generated test (an argument passed to the algorithm). For both criteria, the allowed range is [0, 1], where 0 denotes the worst fitness and 1 the best fitness of an individual. The fitness function is of the following form (10.11):

$$FIT(ind) = 0.5 \cdot C1(ind) + 0.5 \cdot C2(ind). \tag{10.11}$$

For some typical use scenarios in practice, with the presented fitness formula, area coverage may have greater influence on the fitness measure of an individual than the distance between the obtained and specified test difficulty. However, this may be a case more acceptable than the opposite situation because good knowledge coverage of the test is one of the primary goals in the analyzed course. In case the opposite criterion may need to be encouraged, the constant factors in the two addends from the fitness formula may be modified.

Moreover, as evidenced in the Sect. 10.5.1, area coverage of a test item is positively correlated with the difficulty of the items. If a great coverage of all possible areas is required together with a less demanding test featuring just a few items, which may be one of the typical scenarios in practice, the proposed approach is generally expected to discover a solution matching both criteria only to some extent. However, this tradeoff may be somewhat avoided by increasing the number of required items or extending the pool of potential items with those that individually satisfy such requirements.

10.7 Application and Results

For teachers who are primarily interested in obtaining a test for student assessment, there are three especially important algorithm arguments: the exact number of test items (*IC*), student group capacity of target students (*SPDF*), and knowledge areas that need to be covered by the test (*Required*). For the purpose of illustrating the use and performance of the algorithm, we formulate three distinct assessment scenarios:

- S1. The creation of a 5-item test of low difficulty ($IC = 5$, $SPDF = 1.5$), which is an example of a short assessment that may be frequently administered.
- S2. The creation of a more difficult test with 10 items ($IC = 10$, $SPDF = 2.5$), which is an example of an assessment that requires greater concentration and competence from students.
- S3. The creation of a very difficult test with 20 items ($IC = 20$, $SPDF = 3$), which is an example of an assessment useful for discerning between the best students.

In all three cases, the target test was created from a set of 172 items and expected to cover all 19 knowledge areas from the analyzed course. The proposed approach is compared to the random approach incorporating a random generator that randomly chooses items to create a specified number of tests (*TestPoolSize*, *TPS*) and then selects the best one as the solution. The random approach represents a benchmark, as its variant is currently used to generate tests in the analyzed programming course.

The tests from the two groups were evaluated using the metrics built within the fitness function of the genetic algorithm. For each scenario, an experiment was conducted in $N = 10$ iterations using the two approaches. In each iteration, two tests were automatically generated, one using the random approach (RA), the other using the proposed genetic algorithm approach (GA). In all three scenarios, the most important settings for the GA approach were:

- FitnessThreshold = 1
- CrossoverCount = 0.4 × *IC*
- MutationChance = 20 %
- Elitism = true
- ElitismMutation = false

In order to facilitate the comparison of the two approaches, we chose such a *TPS* value for the RA approach so that it leads to the mean completion time which approximately matches the one of the GA approach. The results of the evaluation are presented in Table 10.6. For both approaches, there are the mean fitness of the solution (*MF*), standard deviation for the fitness of the solution (*SDF*), and mean completion time (*MT*) in seconds. For the GA approach, there are also the argument *PS*, and mean generation count (*GC*). For the RA approach, there is also *TPS*. The comparison of solution fitness between the two approaches is done using the Wilcoxon Rank Sum test (*WRST*). The obtained results indicate that, for each of the three scenarios, the GA approach significantly outperforms the RA approach for similar completion times, as evidenced by the differences in *MF* values and the *p*-values for the significance test. Moreover, in each scenario, the GA approach always yielded solutions with the same fitness value, i.e., the GA approach produced consistently good solutions.

The GA convergence, as represented by the change of mean population fitness across generations, is illustrated in Fig. 10.7. The scenario *S1*, which required the

Table 10.6 Comparison of the GA and RA approach for N = 10 iterations

S	GA					RA				WRST (N = 10)
	PS	GC	MF	SDF	MT	TPS (k)	MF	SDF	MT	
S1	100	149.7	0.843	0	4.43	39	0.82	0.008	4.45	W = 95 $p \ll 0.01$
S2	400	22.9	1	0	9.58	48	0.895	0.012	9.83	W = 100 $p \ll 0.01$
S3	500	73.7	0.95	0	93.5	281	0.842	0.012	93.88	W = 100 $p \ll 0.01$

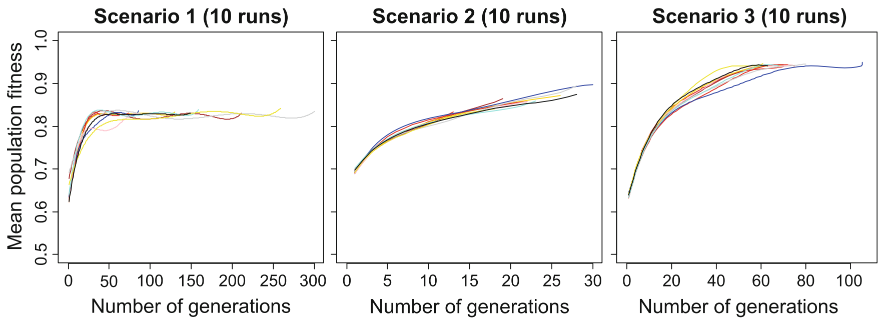

Fig. 10.7 The genetic algorithm convergence for three scenarios

most generations and resulted in the suboptimal solution, was the most demanding most likely because of the insufficient number of test items for the posed requirements. On the other hand, with the increased number of items in *S2*, the GA search ended prematurely after fewer generations due to the discovery of the perfect solution, which had maximum fitness. However, when the further increased number of items was coupled with the need for the extreme difficulty, as in the case of *S3*, the search could not yield the perfect solution but the population managed to converge.

These findings suggest that the proposed GA approach may provide solutions for different assessment scenarios and generate tests in reasonable time. As a result, it should be considered for use in practice in the analyzed course.

10.8 Conclusion and Future Work

In the application of the proposed approach to generation of computer based tests on programming, we utilize the EDM techniques to estimate the difficulty of a test. The difficulty estimate is one of the two important pieces of information that is used to guide the search for a good test, which is conducted using a genetic algorithm. Four variables are used to train a multi-class SVM classifier for the estimation of the difficulty of a new test item, while a regression model is used to estimate the competency of students that should take the test. With this information, the proposed algorithm for test creation attempts to find a test with the minimum difference between the test difficulty and student competency.

The other important piece of information is the test coverage of important programming areas. For the purpose of automatic calculation of coverage, an extensive concept map of the programming competencies and C language elements was designed. Furthermore, each test item was annotated with the matching knowledge areas specified in the concept map. As a result, the algorithm also favors tests that cover more of the required knowledge areas. The benefits of the proposed approach are demonstrated in an evaluation where the presented algorithm is compared to a solution that randomly searches within the item space to find an adequate test.

Our primary goal was not to make a contribution to the field of data mining (DM) but to use existing open implementations of DM algorithms suitable to our needs. The proposed approach includes student modeling, item difficulty estimation and creation of tests for programming assessments. It is modular as it features a separate component for each activity, which may be reused or improved without severely affecting the rest of them. Its primary setting is a university course on programming that features computerized testing of students, i.e., advanced software solution for testing students.

However, the approach could also be applied in any environment where automatic construction of programming tests may provide benefits, including assessments within distance learning systems, as well as within the primary and secondary education. On the other hand, the performance of the approach may be significantly influenced by the quality of the assessment data set. It is required that previous records feature a comprehensive set of student data and items corresponding to various levels of competence and difficulty, respectively. The proposed predictive models should be trained and used on such representative data sets.

There are several possible directions for the future research on the presented issue. The quality of estimates of item difficulty and student competence is also tightly related to the structure of available data. For the purpose of improving these predictions, we may deploy additional mechanisms that would record additional variables related to student performance in programming tests. By merging the accessibility of the proposed approach with the formality and power of IRT, we could further enhance the precision in the assessment process and reduce testing times.

Moreover, we may also create an ontology matching the presented concept map, as a way of providing additional means to its use in different environments. In some scenarios, it may be needed to generate a similar test for different groups of students, where each test should have as few common items with other tests as possible. For this purpose, the algorithm may be extended to generate a set of non-overlapping tests, while the fitness function would have to be modified to include this additional criterion.

Acknowledgments The research was supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, Grant III-44010, Title: Intelligent Systems for Software Product Development and Business Support based on Models. The authors are very grateful to their colleagues from the Chair of Applied Computer Science at the Faculty of Technical Sciences (University of Novi Sad, Serbia), who have contributed to the Otisak testing system and participated in the organization of the Programming Language and Data Structures course, thus making the presented study possible.

References

1. Romero, C., Ventura, S.: Educational data mining: a review of the state of the art. IEEE Trans. Syst. Man Cybern. Part C Appl. Rev. **40**(6), 601–618 (2010)
2. Živanov, Ž., Rakić, P., Stričević, L., Pušić, B., Suvajdžin, Z., Hajduković, M.: Computer aided student examination. Info M **7**(25), 45–53 (2008)
3. Wauters, K., Desmet, P., Noortgate, W.V.D.: Acquiring item difficulty estimates: a collaborative effort of data and judgment. In: Pechenizkiy, M., Calders, T., Conati, C., Ventura, S., Romero, C., Stamper, J. (eds.) 4th International Conference on Educational Data Mining, pp. 121–127. International Educational Data Mining Society, Eindhoven (2011)
4. Peña-Ayala, A., Sossa-Azuela, H., Cervantes-Pérez, F.: Predictive student model supported by fuzzy-causal knowledge and inference. Expert Syst. Appl. **39**, 4690–4709 (2012)
5. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press, Cambrigde (1992)
6. Barla, M., Bieliková, M., Ezzeddinne, A.B., Kramár, T., Šimko, M., Vozár, O.: On the impact of adaptive test question selection for learning efficiency. Comput. Educ. **55**(2), 846–857 (2010)

7. Feng, M., Heffernan, N.: Can we get better assessment from a tutoring system compared to traditional paper testing? Can we have our cake (Better Assessment) and eat it too (Student Learning During the Test)? In: Alven, V., Kay, J., Mostow, J. (eds.) Intelligent Tutoring Systems. LNCS, vol. 6095, pp. 309–311. Springer, Heidelberg (2010)
8. Thelwall, M.: Computer-based assessment: a versatile educational tool. Comput. Educ. **34**(1), 37–49 (2000)
9. Daly, C., Waldron, J.: Assessing the assessment of programming ability. ACM SIGCSE Bull. **36**(1), 210–213 (2004)
10. Douce, C., Livingstone, D., Orwell, J.: Automatic test-based assessment of programing: a review. J Educ. Resour. Comput. **5**(3), 4 (2005)
11. Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O.: Review of recent systems for automatic assessment of programming assignments. In: 10th Koli Calling International Conference on Computing Education Research, pp. 86–93. ACM, New York (2010)
12. Baker, F.B.: The Basics of Item Response Theory. ERIC Clearinghouse on Assessment and Evaluation, Washington (2001)
13. Sosnovsky, S., Gavrilova, T.: Development of educational ontology for C-programming. Int. J. Inf. Theor. Appl. **13**, 303–308 (2006)
14. Sosnovsky, S.: C Programming Language Ontology, http://www.sis.pitt.edu/~paws/ont/c_programming.rdfs
15. Zhou, M., Xu, Y., Nesbit, J.C., Winne, P.H.: Sequential pattern analysis of learning logs: methodology and applications. In: Romero, C., Ventura, S., Pechenizkiy, M., Baker, R.S.J.d. (eds.). Handbook of Educational Data Mining, Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, pp. 107–121. CRC Press, Boca Raton (2010)
16. Faculty of Technical Sciences in Novi Sad, Accreditation of the Study Programme; Computing and Control Engineering, http://www.ftn.uns.ac.rs/_data/planovi/2012/engleski/osnovne/ftn_e2.pdf
17. Rakić, P., Stričević, L., Živanov, Ž., Suvajdžin, Z., Hajduković, M.: Computer classroom: deployment and exploitation. Info M **6**(21), 9–13 (2007)
18. Klyne, G., Carroll, J.J., McBride, B.: Resource description framework (RDF): concepts and abstract syntax. W3C Recommendation **10** (2004)
19. McGuinness, D.L., Van Harmelen, F.: OWL web ontology language overview. W3C Recommendation **10** (2004)
20. Motik, B., Patel-Schneider, P.F., Parsia, B., Bock, C., Fokoue, A., Haase, P., Smith, M.: OWL 2 Web ontology language: structural specification and functional-style syntax. W3C Recommendation **27,** 17 (2009)
21. Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., Sattler, U.: OWL 2: the next step for OWL. Web Semant.: Sci., Serv. Agents World Wide Web **6**(4), 309–322 (2008)
22. Falconer, S.: OntoGraf, http://protegewiki.stanford.edu/wiki/OntoGraf (2010)
23. TopBraid Composer, http://www.topquadrant.com/products/TB_Composer.html
24. Krivov, S., Williams, R., Villa, F.: GrOWL: a tool for visualization and editing of OWL ontologies. Web Semant.: Sci., Serv. Agents World Wide Web **5**(2), 54–57 (2007)
25. Novak, J.D., Cañas, A.J.: The theory underlying concept maps and how to construct and use them. Technical report, Florida Institute for Human and Machine Cognition (2008)
26. Novak, J.D.: Learning, Creating, and Using Knowledge: Concept Maps as Facilitative Tools in Schools and Corporations. Taylor & Francis, New York (2010)
27. Ruiz-Primo, M.A., Shavelson, R.J.: Problems and issues in the use of concept maps in science assessment. J. Res. Sci. Teach. **33**(6), 569–600 (1996)
28. Cañas, A.J., Hill, G., Carff, R., Suri, N., Lott, J., Eskridge, T., Carvajal, R.: CmapTools: a knowledge modeling and sharing environment. In: Concept maps: Theory, Methodology, Technology 1st International Conference on Concept Mapping, vol. 1, pp. 125–133. Universidad Pública de Navarra, Pamplona (2004)
29. Bivand, R.: ClassInt: Choose Univariate Class Intervals. R package version 0.1-19 (2012), http://CRAN.R-project.org/package=classInt

30. R Core Team.: R: A Language and Environment for Statistical Computing, Manual. R Foundation for Statistical Computing (2013)
31. Karatzoglou, A., Smola, A., Hornik, K.: Achim Zeileis, A.: Kernlab—an S4 package for kernel methods in R. J. Stat. Softw. **11**(9), 1–20 (2004)