VYTAUTAS MAGNUS UNIVERSITY
INSTITUTE OF MATHEMATICS AND INFORMATICS
OF VILNIUS UNIVERSITY

Jūratė SKŪPIENĖ

EVALUATION OF ALGORITHM-CODE COMPLEXES IN INFORMATICS CONTESTS

Doctoral Dissertation

Physical Sciences (P 000)
Informatics (09 P)
Informatics, Systems Theory (P 175)

Vilnius, 2010

This research was accomplished in the period from 2006 to 2010 at the Institute of Mathematics and Informatics of Vilnius University.

Scientific Advisor:

prof. habil. dr. Antanas Žilinskas (Institute of Mathematics and Informatics of Vilnius University, Physical Sciences, Informatics - 09 P).

Acknowledgements

To my advisor Prof. Dr. Habil. Antanas Žilinskas and to Dr. Tom Verhoeff, I would like to express my sincere gratitude for your professional support and guidance.

To Dr. Olga Kurasova, Prof. Dr. Habil. Juvencijus Mačys, Janina Kazlauskaitė, Aidas Žandaris, and Mārtiņš Opmanis, thank you for your thorough review of my thesis. Your advice and criticism was invaluable.

To my colleagues at the Informatics Methodology Department of the Institute of Mathematics and Informatics, and Vilnius International School, your encouragement was greatly appreciated.

To my family, my husband Bronius and my daughter, Rusnė, thank you for your help, patience and understanding. You were there for me all hours of the day and night.

Jūratė Skūpienė

Abstract

Informatics contests for high school and undergraduate students is one of the fastest expanding extra-curricula activity. There is a large community of national, regional, international, on-line and on-site contest organisers and participants. During an informatics contest, the contestants get algorithmic tasks, they have to design an algorithm, implement it, and submit it in a form of source code in allowed programming language. The term *algorithm-code complex* stands for a program which contains an implementation of an unknown algorithm designed to solve the given task.

The dominant form of evaluation in informatics contests is black-box testing which does not take into account internal design and structure of an algorithm-code complex. Thus we get a contradiction between the contest goals to challenge contestants in algorithmic problem solving and the dominant form of evaluation which does not take into account the algorithm itself.

The objective of this research was to investigate the background of evaluation in informatics contests, in particular in Lithuanian Informatics Olympiads, and to propose a better motivated evaluation scheme.

First we analyse the background of evaluation of algorithm-code complexes, and concerns regarding the current evaluation practice. Then we overview the experience of evaluation of programming assignments in programming courses with a search for experience to be transferred to informatics contests.

Next we present the multiple criteria decision analysis (MCDA) concepts and processes and present evaluation in informatics contests as an MCDA problem. We also look for suitable MCDA approaches to be applied for evaluation in informatics contests.

In the next part we analyse the alternative possibilities to introduce semi-automated evaluation for tasks with graphs.

Before applying MCDA approaches we analyse evaluation in informatics contests from the point of view of existing quality standards.

In the final part we follow the required MCDA stages, apply fuzzy logic and group decision making methods, and propose the evaluation scheme to use it in Lithuanian Informatics Olympiads.

Contents

List of Figures

List of Tables

Glossary

Informatics Contests

ACM-ICPC ACM International Collegiate Programming Contest (ACM, 2010a). It is a team informatics contest for university students where team of members shares one computer and where *all-or-nothing scoring* is applied.

Algorithm-code complex A program which contains an implementation of an unknown algorithm designed to solve the given task.

All-or-nothing batch soring A scoring scheme, where *all-or-nothing scoring* is applied to a test case (but not to the whole set of tests). See Subsection 2.12.4.

All-or-nothing scoring A scoring scheme which classifies the submissions into two categories: *accepted* or *not accepted* without any intermediate values. See Subsection 2.12.3.

Automated evaluation A form of evaluation in which a computer program aids the evaluator in grading student's work and facilitates the feedback process. See Sections 2.9 and 2.11.

Batch task A task where all the tests used for grading are fixed before the beginning of evaluation and do not depend upon program behaviour. See Section 2.7.

Black-box testing is testing that ignore internal logic of a program and focuses solely on the outputs generated in response to selected inputs and execution conditions (Williams, 2006). See Section 2.12.

Contest Management System (CMS) A group of server applications and modules to support informatics contests by providing submit, test, backup, restore, feedback and automated grading facilities.

Contestant The participant of informatics contests.

Correct algorithm An algorithm which for every valid input instance halts with the correct output (Cormen et al., 1992).

Correctness tests Tests designed with the intention to check algorithm-code complex correctness in order to separate correct algorithm-code complexes from incorrect ones.

Dynamic evaluation A form of evaluation based on observations of program behaviour during its execution.

Efficiency A characteristic of an algorithm-code complex described as run time performance and memory usage and expressed in big O notation. For each task, big O characteristic is associated to a linguistic scale (like *inefficient*, *low efficiency*, *efficient*) and the table of the expected scores. This association is a constructed notion for each task. See Subsection 2.10.2.

Efficiency tests Tests designed with the intention to check the algorithm-code complex efficiency and distinguish between different efficiency categories of solutions.

Evaluation scheme List of evaluation criteria together with the corresponding metrics, and the score aggregation function.

Expected score Expectations (of the jury) about the score of a particular type of solution(s) assigned as the outcome of black-box testing.

Full feedback A form of feedback where black-box testing with grading tests is performed during the contest from the point of view of the contestant. The exact feedback provided to the contestant during the contest is not explicitly defined by this concept.

Grading tests A set of tests used for evaluation (i.e. the score is related to algorithm-code complex performance with these tests) and often they are not disclosed to the contestants before the competition is over.

Informatics contest A task-based problem solving contest with exam sessions where the tasks are such that they have a correct and efficient algorithm, the solutions should be implemented as algorithm-code complexes, tested automatically and the final ranking of the contestants is derived. See Section 2.4. Note, that we narrowed the very general term for the use in the thesis.

International Olympiad in Informatics (IOI) An international individual on-site informatics competition for students in secondary education (IOI, 2010; Verhoeff, 2009). The submission is limited to an algorithm-code complex, the evaluation consists of black-box testing only.

Jury A group of people with background in informatics responsible for performing evaluation in informatics contests. In LitIO *the jury* and *the scientific committee* are the same group of people.

Lithuanian Olympiads in Informatics (LitIO) A national state supported individual informatics contest for students in secondary education in Lithuania. (Lit, 2010).

Manual evaluation Evaluation which is performed by human evaluators.

Partial scoring A scoring scheme, where points are assigned for each grading test, and the score for a task is calculated as the sum of scores for each grading test. See Subsection 2.12.2.

Problem A well-specified computational problem, where the statement of the problem specifies in general terms the desired input/output relationship and which requires to design an algorithm in order to solve it (Cormen et al., 1992).

Programming assignment A task given as an assignment during programming courses.

Scientific Committee A group of people with background in informatics responsible for informatics contest syllabus, choice of tasks and task preparation for the contest. In LitIO, *the Scientific Committee* is the same as *the Jury.*

Scoring function Explanation to the contestants how the points will be distributed for a task. It is announced to the contestants together with the task formulation.

Scoring scheme A scheme for assigning and aggregating points for black-box testing.

Semi-automated evaluation A form of evaluation where human evaluators perform (part of) work, but a computer program simplifies the process.

Static evaluation A form of evaluation which involves analysis of the program without executing it.

Submission The material presented for the evaluation of the jury by the contestant. We assume that each submission consists of an algorithm-code complex and/or other material required by the task.

Task A detailed specification of the problem which determines requirements for the solutions to be submitted and evaluated. See Section 2.7.

Task discrimination The ability of a task to separate contestants who vary in their degree of skills of the material tested.

Test A valid input for the task together with a corresponding output.

Test case A set of one or more tests typically targeting the same well-defined characteristic of the submitted algorithm-code complex.

Multiple Criteria Decision Analysis

Alternatives Different choices available to the decision maker. In case of evaluation in LitIO problem the set of alternatives consists of all the submissions designed to solve a particular task in an exam session of an informatics contest.

Attribute A statement of something that is desired to be achieved. Attributes represent the different dimensions from which the alternatives can be viewed. Attribute specification does not require measure specification. It is possible that attributes are arranged in a hierarchical manner.

Crisp set Any collection of objects from the given universe without regard of their order. For any object from the given universe its membership in the crisp set must be unambiguously defined.

Criteria Each attribute of an MCDA problem is measured in terms of one or more criteria. The same criteria may be used for measuring different attributes. Different criteria might be associated with different units of measure.

Decision weights Weights of importance assigned by the decision makers to each criteria.

Evaluation in LitIO problem By this term we understand an MCDA problem the goal of which is to investigate the context of evaluation in LitIO and other informatics contests, to construct the concept of submission, the list of attributes and criteria, to propose partial value and value functions for score aggregation using MCDA approaches.

Fuzzy set Any set that allows its members to have different degree of membership in the interval [0, 1].

Group decision making (GDM) A decision making process based on the opinions of several individuals.

Interval scale A measurement scale where one unit on the scale represent the same magnitude across the whole range of the scale.

Linguistic variables Variables whose values are linguistic terms and not numbers. They are used to express results of subjective qualitative evaluation or the fuzzy data.

Measurement The assignment of numbers to objects or events in a systematic order.

Objective *(adj.)* Based on observable phenomena and uninfluenced by emotions and personal prejudices.

Partial value function A function assigning a non-negative number to each alternative indicating the desirability (or preference) of the alternative in terms of one or more criteria.

Ratio scale	An interval scale which where *zero* represents the absence of thing being measured.
Scale	A rule using which the measurement is performed (Stevens, 1946).
Value function	A function assigning a non-negative number to each alternative indicating the desirability (or preference) of the alternative.

1 Introduction

1.1 Statement of the Problem and its Relevance

Algorithmic problems and analysis of their solutions and properties are an important part of computer science studies. Informatics contests for high school students also deal with algorithmic problems in a specific way. The contestants are given an algorithmic problem and have to design an algorithm, to implement it, and submit as a working program. However, the students are not required to submit a formal proof of algorithm correctness and efficiency, because they are still at high or secondary schools, and such a task would be beyond their capabilities.

Thus we get the concept of *an algorithm-code complex*. The term was invented in this thesis, and it stands for a program which contains the implementation of an unknown algorithm designed to solve the given task. An algorithm-code complex combines the outcome of both an algorithm design and program development. As a result, it becomes difficult to evaluate the characteristics of algorithm and its implementation in the algorithm-code complex, because it is difficult to distinguish them and tell whether a feature of the algorithm-code complex belongs to the algorithm or to its implementation. Thus, evaluation of qualities of the algorithm-code complex in informatics contests becomes an interesting scientific problem.

Current practice of many contests of this kind is that the dominant part of evaluation (i.e., making a decision on the properties and qualities of the implemented algorithm and implementation in the form of scores) is based on empirical black-box testing of the algorithm-code complex. Task designers have certain expectations about the relationship of the quality and characteristics of solutions to the measure of those qualities expressed by points.

However, the essence of black-box testing is that no knowledge of the key ideas of algorithm, internal logic, and code structure is revealed. Therefore, the conclusions about the qualities of the algorithm-code complexes made in the form of assigned scores pose various educational and scientific questions.

This imbalance between the goal to challenge the contestants to algorithmic problem solving and the commonly accepted practice of evaluation, based on black-box testing, served as the key motive to start this research.

It is considered that at present there is no alternative to the automated evaluation in informatics contests. Therefore it is highly important to conduct a broader more structured research in order to identify the main concerns, priorities, and alternatives, to improve and scientifically investigate evaluation schemes which involve not only black-box testing, but also other types of automated evaluation.

Even though the dissertation investigates evaluation issues in Lithuanian Informatics Olympiads (LitIO), the problem is relevant in a much broader context. LitIO follow the model of International Olympiads in Informatics (IOI). IOI is the most prestigious world contest in programming for individual contestants from various invited countries. There are many other national, regional and international, online and on-site olympiads and contests for secondary school students which follow the IOI model, apply black-box testing as the main evaluation approach and at the same time confront similar challenges related to task design and evaluation.

The research focuses on the evaluation in LitIO since the author has been involved in the jury of LitIO for many years, and is familiar with the evaluation challenges and concerns. Involvement in the organization of LitIO made it possible to use submissions of LitIO for the investigation as well as provided opportunities to initiate practical application of the research results.

1.2 Research Objectives and Tasks

The objectives of this dissertation are: to investigate the evaluation criteria and the evaluation schemes applied in the evaluation of algorithm-code complexes; to develop the evaluation scheme based on the multiple criteria decision analysis methods, suitable to be applied in LitIO.

The tasks of the dissertation are as follows:

1. To present a survey of evaluation practice and problems in informatics contests. To analyse the evaluation of submissions to programming assignments in undergraduate studies. To determine whether this experience could be transferred to informatics contests.

2. To analyse the possibilities of including a semi-automated evaluation in informatics contests using visualisation of algorithm-code complexes (the case of tasks with graphs).

3. To analyse a chosen set of algorithm-code complexes, to determine the precision of measurements of the quality of algorithm-code complexes based on the testing results. To establish whether the the quality of the algorithm implementation in an algorithm-code complex is related to the quality of the programming style.

4. To define the evaluation in informatics contests as a multiple criteria decision problem. To analyse the multiple criteria decision process and methods, and to propose methods suitable for solving the evaluation problem.

5. Using the MCDA methods, to construct the evaluation scheme for LitIO, consisting of the list of components of a submission, the list of its measurable attributes, the list of evaluation criteria for each attribute, and the score aggregation function.

1.3 Defended Statements

1. It is reasonable to improve the evaluation scheme currently applied in informatics contests for evaluating the quality of algorithm-code complexes.

2. Semi-automated evaluation fastens the evaluation process, and allows introduction of additional evaluation criteria into the evaluation scheme.

3. Use of MCDA facilitates the development of more grounded evaluation schemes.

1.4 Research Methods

Systematisation and *a comparative analysis* were applied when preparing the analytical part of the thesis.

Submissions of the contestants were analysed by applying *the analysis of algorithms* and the obtained results were processed using *the statistical package SPSS* and *descriptive statistics.* The validity of LitIO evaluation criteria was investigated by applying *the methods of software development.*

Approaches and methods proposed by MCDA, in particular, *modelling*, *the Goal/ Question/ Metric framework* and *the expert evaluation*, *Value measurement theory* and *the fuzzy logic*, were applied in the creation of the evaluation scheme for LitIO.

1.5 Research Findings and Results

1. We have investigated the experience of evaluation in informatics contests, analysed and classified the problems related to testing in informatics contests. We have also explored the experience of using testing for evaluating submitted programs in undergraduate courses.

2. We have examined the possibilities of visualisation of the graphs implemented in algorithm-code complexes, and selected the visualisation paradigm. We have analysed and classified the graph implementation methods in 191 algorithm-code complex, solving graph tasks. Basing on the results of the investigation, we have created a tool for semi-automated visualisation of the graphs implemented in the algorithm-code complexes.

3. We have analysed the validity of the evaluation criteria currently applied in LitIO in regard to ISO-9126-1 quality model. We have investigated 290 algorithm-code complexes and determined a deviation from the expected scores of the quality measurements obtained using two testing score aggregation schemes: *partial soring* (20.2% of scores are unjustified) and *all or nothing batch scoring* (8.4% of scores are unjustified). We have calculated the correlation between the quality of programming style and the quality of algorithm implementation: 0.468.

4. We have defined *the evaluation in informatics contests* as an MCDA problem that belongs to *group decision making*, and repeated classes, and to *the ranking problematique* category. We have analysed the MCDA approaches and methods and selected suitable ones for solving this problem: *Goal/Question/Metric* framework, and *the Group decision support algorithm* combined with *the approach of Chen.*

5. By applying the chosen methods, we have constructed the evaluation scheme and suggested to be applied in LitIO.

6. We have piloted the suggested evaluation scheme with four tasks during a small contest. We have summarised the piloting results and feedback, and have presented proposals how to adapt the evaluation scheme to concrete tasks.

1.6 Scientific Novelty

- The thesis provides the first extensive analysis of the problem field of evaluation in informatics contests (to our knowledge).
- A novel idea has been suggested and investigated to apply visualisation in a semi-automated evaluation of algorithm-code complexes solving graph tasks.
- The MCDA theory including the fuzzy logic and group decision making approaches was applied to developing of the evaluation scheme for LitIO.

1.7 Approbation and Publications

The results of the dissertation were presented and discussed in the following national and international conferences:

- The Fourteenth International Scientific Conference "Computer Days – 2009", 2009, Kaunas, Lithuania, Kaunas University of Technology.
- The Ninth International Conference on Teaching Mathematics "Retrospective and Perspectives", 2008, Vilnius, Lithuania, Vilnius Pedagogical University.
- The Thirteenth International Scientific Conference "Computer Days – 2007", 2007, Panevėžys, Lithuania, Panevėžys Institute of Kaunas University of Technology.
- The First International Olympiads in Informatics Conference "Country Experiences and Developments", 2007, Zagreb, Croatia.
- The XLVIII'th Conference of the Lithuanian Mathematical Society, 2007, Vilnius, Vilnius Gediminas Technical University.
- The Second International Conference ISSEP "Informatics Education – the bridge between using and understanding computers", 2006, Vilnius, Lithuania.

- The Fourth E-learning Conference “Computer Science Education”, 2007, Istanbul, Turkey.
- The Third E-learning Conference “Computer Science Education”, 2006, Coimbra, Portugal.
- 2’nd International Conference on Information Technology “Research and Education”, 2004, London, UK.
- Scientific Conference “Information Technologies’2004”, 2004, Kaunas, Lithuania, Kaunas University of Technology.

The main results of the dissertation were published in the following papers:

1. Skūpienė, J. (2010). Score Calculation in Informatics Contests using Multi Criteria Decision Methods. *Informatics in Education*, accepted for publication.
2. Skūpienė, J. (2010). Improving the Evaluation Model for the Lithuanian Informatics Olympiads. *Informatics in Education*, ISSN 1648-5832, 9(1):141-158.
3. Skūpienė, J. (2009). Lietuvos informatikos olimpiados darbų vertinimas programinės įrangos kokybės modelio požiūriu. *Informacijos mokslai*, ISSN 1392-0561, 50: 153–159.
4. Skūpienė, J. (2009). Credibility of Automated Assessment in Lithuanian Informatics Olympiads: One Task Analysis. *Pedagogika*, ISSN 1392-0340, 96: 143–151.
5. Skūpienė, J. (2007). Assumptions for Automated Grading of Programming Style in Informatics Olympiads. *Lithuanian Mathematical Journal*, ISSN 0132-2818, 47: 273–278.
6. Skūpienė, J. (2007). Development and Perspectives of Automated Grading in Informatics Olympiads. *Information Sciences*, ISSN 1392-0561, 42–43: 43–49.
7. Skūpienė, J., and Žilinskas, A. (2007). Automated Grading of Programming Tasks Fulfilled by Students: Evolution and Perspectives. In *Proceedings of the 4’th E-learning’07 Conference.* Istanbul Turkey.
8. **Skūpienė, J.** and Žilinskas, A. (2006). Evaluation in Informatics Contests: Aids for Tasks Involving Graphs. *The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, ISSN: 0773-182, 23(1–4): 39–46.
9. Skūpienė, J. (2006). Programming style – Part of Grading Scheme in Informatics Olympiads: Lithuanian Experience. In *Information Technologies at School, Proceedings of the Second International Conference “Informatics in Secondary Schools: Evolution and Perspectives”.* Vilnius, Lithuania, 2006, pages 545 – 552, (ISI Proceedings List).

10. Skūpienė, J., and Žilinskas, A. (2006). Evaluation of programs in Informatics Contests: Case of Implementation of Graph Algorithms. In *Proceedings of The Third E-learning conference "Computer Science Education"*. Coimbra, Portugal, 2006.

11. Skūpienė, J. (2004). Automated testing in Lithuanian Informatics Olympiads. In *Informacinės technologijos 2004, Konferencijos pranešimų medžiaga*. Kaunas, Technologija, pages 37–41.

1.8 Synopsis

The dissertation consists of seven chapters.

The first chapter is the introduction. It contains the problem statement and its relevance, research objectives and tasks, methods applied in the research, the findings and results, scientific novelty of the research, its practical importance as well as the list of publications which served as a basis for this dissertation.

The second chapter presents an overview of the problematics of evaluation of algorithm-code complexes. It presents the main terms and concepts in order to provide terminology for discussion. The investigation of evaluation in informatics contests cannot be conducted without understanding of the origin, scope, goals, syllabus and structure (which defines limitations and resources available for evaluation) of informatics contests. The chapter also presents the current LitIO evaluation scheme, and a survey of scoring schemes applied in other informatics contests. Black-box testing is a dominant form of evaluation in LitIO as well as in other informatics contests. The issues regarding black-box testing were identified and structured in the second chapter.

The analysis of experience in the evaluation of programming assignments with a view to transfer it to the evaluation in informatics contests is presented.

In the *third chapter*, we describe the evaluation in LitIO problem as a multiple criteria decision analysis (MCDA) problem. Therefore we survey the stages of the MCDA process and various MCDA approaches for solving MCDA problems, among them the methods that apply the fuzzy logic and group decision making methods. We elicit the methods that are most suitable for solving the evaluation in LitIO problem.

The fourth chapter investigates the possibilities for visualisation of the graphs implemented in the algorithm-code complexes, designed by the contestants. Our intention was to broaden the possibilities for a semi-automated evaluation in informatics contests. The chapter presents the analysis of a set of submissions for three different graph tasks. Different graph implementations that were encountered in the algorithm-code complexes have been analysed and categorised.

In *the fifth chapter*, the life cycles of a submission and of software are compared using the waterfall life cycle model. The current LitIO evaluation scheme is analysed from the point of view of existing quality standards, in particular, the ISO-9126-1 software quality model. We motivate there why reliability, usability and portability are not included into the evaluation scheme, and separately analyse the evaluation of the other three quality characteristics.

The concern of the current evaluation is the question whether the results of the automated testing of functionality presented in the form of scores, correspond to the intentions of the task designers. To establish this, a set of submissions was analysed manually, in order to identify whether the expected scores deviated from the scores obtained during automated testing. Another set of submissions was investigated with a view to check the hypothesis that a good programming style has an influence on a successful implementation of algorithms, designed by the contestants.

The aim of the *the sixth chapter* was to develop an evaluation scheme for LitIO using MCDA. First, the problem structuring phase was performed, it included the work with experts and was performed using the Goal/Question/Metric framework. The work done with experts is presented in detail, illustrating the way to the final outcome - a refined concept of the submission and a detailed list of evaluation criteria. By combining the outcome of this chapter with that of the third chapter, we get the evaluation scheme for applying it in LitIO. Finally, we present a description of the experiment where four tasks are given to a number of students, evaluated using the proposed evaluation scheme, and the values of the sensitivity parameters have been calculated.

2 Problematic of Evaluation of Algorithm-Code Complexes

In this chapter we overview the problematic of evaluation in informatics contests, and in particular, in *Lithuanian Informatics Olympiads* (LitIO). We define the main concepts, look at LitIO goals and structure, domain of problems, present the current LitIO evaluation scheme, identify the cuts of looking at the evaluation problematic, identify and structure the main evaluation problems in informatics contests.

The evaluation in informatics contests might have common points with the evaluation of programming assignments in programming courses, transferable to evaluation in informatics contests. Therefore, in this chapter we will look at the development and the experience of the automated evaluation of programming assignments.

2.1 Introduction

"The programming contests offer a unique environment for research in several areas of computer science, in particular, computer science education" (Trotman and Handley, 2006). Informatics contests are introduced as the fastest expanding co-curricular activity related to computer science which is seen as a good model of competitive learning (Revilla et al., 2008). Overviews of different aspects of informatics contests can be found in (Cormack et al., 2006; Pohl, 2006; Skienna and Revilla, 2003; Trotman and Handley, 2006; Vasiga et al., 2008; Verhoeff, 2009).

Informatics contests as an object of scientific interest is rather a new topic. Most of the early contest related papers were more descriptive rather than presenting a deeper scientific analysis of different aspects of informatics contests (Bryson and Roth, 1981; Comer et al., 1983; Deimel, 1984, 1988; Metzner, 1983; Ryan and Deimel, 1985; Salniek and Naylor, 1988). One of the recent events that has initiated treatment of informatics contests as the object of scientific interest in the area of computer science education was a workshop on informatics contests held in Germany in 2006 (Per, 2006).

Informatics contests are contests of algorithmic problem solving. The contestants are given an algorithmic problem and have to design an algorithm, to implement it, and submit as a working program. The proof of the algorithm correctness and efficiency is not required. One of the reasons is that it is too dfficult for the the contestants who are still at high school.

Thus we get the concept of *an algorithm-code complex.* The term was invented in this thesis, and it stands for a program which contains an implementation of an unknown algorithm designed to solve the given task. An algorithm-code complex combines the outcome of both an algorithm design and program development. As a result, it becomes hard to evaluate characteristics of the algorithm, and of its implementation in the algorithm-code complex, because it is hard to separate them and identify whether a feature of an algorithm-code complex belongs to an algorithm or to its implementation. Thus, evaluation of qualities of an algorithm-code complex in informatics contests becomes an interesting scientific problem.

Current practice of many such contests is that the dominant part of evaluation (i.e. making decision about properties and qualities of implemented algorithm and the implementation in a form of scores) is based on empirical black-box testing of algorithm-code complex. Task designers have certain expectations about the relationship of the quality and characteristics of solutions to the measure of those qualities expressed in points.

However, the essence of black-box testing is that no knowledge of key ideas of algorithm, internal logic and code structure is revealed. Therefore, the conclusions about the qualities of the algorithm-code complexes made in the form of assigned scores raise various educational and scientific questions.

In the earlier papers we already found concerns about such a form of evaluation (Struble, 1991). Recent papers present a much deeper analysis of various black-box evaluation related concerns and aspects. In this chapter we overview and structure these issues.

Typically, static and/or dynamic evaluation is performed to evaluate programs which are given as an assignment or an exam task in undergraduate courses of computer science education. Much research has been completed in that area (Douce et al., 2005; Helmick, 2007; Rahman et al., 2007; Woit and Mason, 1998). A lot of emphasis was put on the grading tools and their features, (Ahoniemi and Reinikainen, 2006; Foxley et al., 2004; Harris et al., 2004; Spacco et al., 2005; von Matt, 1994). In this chapter we look at the experience of evaluating programming assignments and will identify whether that could be transferred to informatics contests.

We have found references that informatics contests were already organised in the early sixties (Comer et al., 1983). Currently informatics contests span over a huge number of participants and scientists involved in organising the contests. Around 300 participants in total come to represent their countries and to compete in IOI (IOI, 2010), the leading international contest for students in secondary education. However, IOI is at the top of the pyramid. The bottom part consists of national contests(Anido and Menderico, 2007; Kolstad and Piele, 2007; Wang et al., 2007). No more than four contestants can represent their country in IOI. Around 2000 participants join Lithuanian Informatics Olympiads (LitIO) each year to compete for awards and to be selected as those four to represent Lithuania.

ACM International Collegiate Programming Contest (ACM-ICPC) is a team contest for university students around the world (ACM, 2010a). The team consists of up to three members and shares one computer. In ACM-ICPC the algorithm-code complexes submitted for evaluation are either accepted or rejected without any intermediate values, i.e. *all-or-nothing* scoring is applied. Currently it is the leading team contest for post-secondary students. It is possible to observe the growth of this contest with less than 500 teams and universities in 1989 and more than 5,000 teams and 1,500 universities in 2005 (Patterson, 2005). The teams have to pass a tight competition in regional quarter-finals and semi-finals to proceed to the finals. 103 teams from over 200 regions competed in the world finals of ACM–ICPC in Harbin in north-east China in 2010. There were some 22,000 participants in the preliminaries, and they represented 1,931 universities in 82 different countries (ACM, 2010b).

At present, the leading on-line informatics contest is *TopCoder algorithm competition* established in 2001. It is an open privately run on-line contest with a large electronic community of more than 150 thousands registered users (Top, 2010).

2.2 Concepts

The key concepts of the thesis are *an algorithm-code complex*, *informatics contests*, and *evaluation*. Here we introduce them together with other relevant concepts:

Problem. Well specified computational problem, where the statement of the problem specifies in general terms the desired input/output relationship and which requires an algorithm in order to solve it (Cormen et al., 1992).

Algorithm. Well defined computational procedure that takes some value as input and produces some value as output (Cormen et al., 1992).

We only consider the algorithms that are designed with purpose to solve the given problem.

Correct algorithm. An algorithm which for every valid input instance halts with the correct output (Cormen et al., 1992).

Algorithm efficiency. A characteristic of an algorithm described as run time performance and memory usage and expressed in big O notation.

Note, that when we use linguistic terms *inefficient, low efficiency, efficient*, etc, we mean that the linguistic terms are associated with big O characteristic taking into account the context in which the problem is being solved.

For example, if we give a task to sort n numbers, and want to make it an easy problem, we may decide that an $O(n^2)$ algorithm is *efficient*. If we want to make the problem difficult, we may define that the same $O(n^2)$ algorithm is *inefficient*, and an *efficient* algorithm should have $O(nlog(n))$ complexity.

Task. A detailed specification of the problem which determines requirements for the solutions to be submitted and evaluated. We only consider the tasks that require to implement the algorithm solving the problem in approved programming language. The structure and typical format of tasks is discussed in Subsection 2.7.

Informatics contest. A task-based problem solving contest with exam sessions where the tasks are such that there exists a correct and efficient algorithm solving the problem, the solutions are implemented in approved programming languages, tested, assigned score and the final ranking of the contestants based on the scores is delivered.

We introduced this concept here shortly, so that we could start talking about the problematics of the thesis. However, the wider background of the concept and how we arrived to it will be discussed in Subsection 2.4.

Contestant. The participant of informatics contests. We assume that the contestants are either in secondary or in high schools, except for ACM-ICPC contests for university students.

Algorithm-code complex. A program, submitted as a (part of) solution to a task in informatics contest in a form of its source code.

It is expected that the program contains an algorithm intended to solve the task, presented in a form of implementation in approved programming language.

Instead of this concept we could have used the term *program*. However, by defining an algorithm-code complex, we wanted to emphasise that we are in a situation when we have both an algorithm and its implementation in one place and we can not easily measure the qualities of each item separately, or decide whether a certain feature belongs to the algorithm or to the implementation.

This situation occurs *de facto* in many informatics contests, like IOI. One reason may be the willingness of the contest organisers to use black-box testing as the only form of evaluation. On the other hand, the contestants are students in secondary education, and providing proof of their algorithm correctness may be beyond their capabilities.

Submission. Solution to task, submitted for evaluation. The exact specification of submission (i.e. what a solution should consist of) is written in the task. We assume that an algorithm-code complex is always the required part of submission.

Evaluation. A systematic determination of qualities of something against a set of standards.

In informatics contest we are interested in evaluation of submissions. Informatics contest as an event is not related to any controlled successive teaching and learning process. Therefore much reasoning which is common when evaluation take place in the middle or at the end of some course is not valid here. There is no context arising from the learning process and its goals. Besides, the evaluators may not have a direct contact with the contestant if this is an on-line contest.

Therefore we decided to focus on *the quality of a submission* in the evaluation.

Test. A valid input for the task together with a corresponding output. Note, that for some tasks there are many correct outputs to the same input.

Black-box testing. Testing that ignores internal logic of a program and focuses solely on the outputs generated in response to selected inputs and execution conditions (Williams, 2006).

2.3 Goals of Informatics Contests

Many different informatics contests for high school and undergraduate students are organised every year. Each contest has its own goals. We looked at the goals of most popular informatics contests (Cormack et al., 2006; Trotman and Handley, 2006), and noticed that the list of goals is similar in many informatics contests. However, different contests put different weights and importance to various goals.

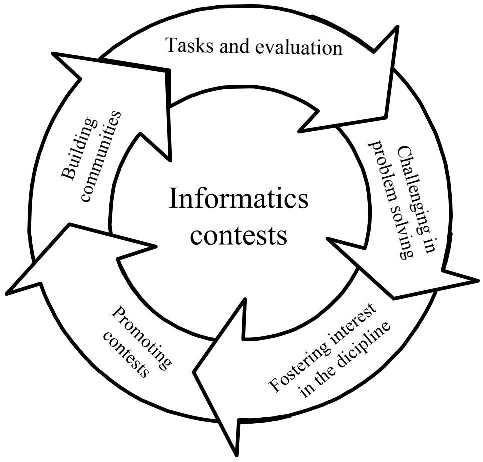

Figure 2.1: Relationship of goals of informatics contests

Next we look at the goals that we consider most important in LitIO and in the context of this research.

"Bringing the discipline of informatics to the attention of young people" (IOI, 2008), illustrating the nature of the discipline, exposing students to the interesting breath of computer science are the goals emphasised in many informatics contests (Lie, 2008; Astrachan et al., 1993; Kearse and Hardnett, 2008; Kolstad and Piele, 2007; Pohl and Polley, 2006). Computer science typically is not included (or included at a low level) into the informatics teaching curricula at the gymnasium level (Blonskis and Dagienė, 2008; Dagienė, 2008). Informatics contests serve as a stimulator for the students to get interested in the subject, explore it, develop analytical skills, challenge themselves and consider informatics as the choice for their future career.

In the last few years the media announces that interest in computer science decreases both among gymnasium graduates and undergraduates already enrolled in informatics. Informatics contests attract to the discipline (Patterson, 2005; Shilov and Kwangkeun, 2002). High school informatics contests organised by universities have become an important component of recruiting efforts (Kaz, 2010; Bowring, 2008; Myers and Null, 1986; Sherrel and McCauley, 2004). Analysis of surveys of the contestants has corroborated that the contests increased interest in computer science of more than 70% of respondents and had other positive effects on the contestants (Sherrel and McCauley, 2004).

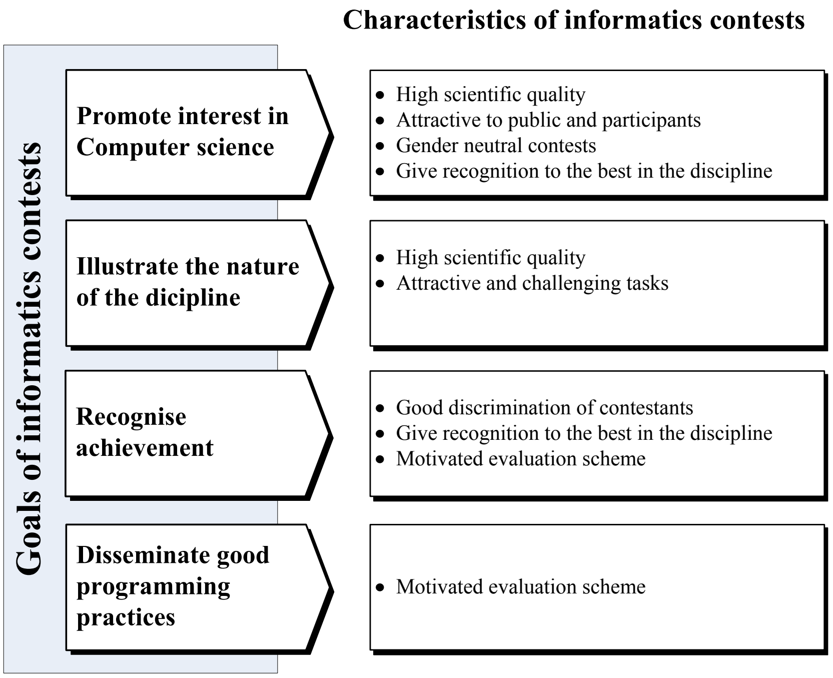
Characteristics of informatics contests
Goals of informatics contests
Promote interest in Computer science
Illustrate the nature of the dicipline
Recognise achievement
Disseminate good programming practices
• High scientific quality
• Attractive to public and participants
• Gender neutral contests
• Give recognition to the best in the discipline
• High scientific quality
• Attractive and challenging tasks
• Good discrimination of contestants
• Give recognition to the best in the discipline
• Motivated evaluation scheme
• Motivated evaluation scheme

Figure 2.2: Most important goals and characteristics of informatics contests

A high scientific quality of informatics contests is an important goal of LitIO and of IOI (Verhoeff, 2006). Universities that take into account the awards obtained in informatics contests during enrollment are also interested in a high scientific quality.

Another important role is recognising achievement of each contestant, not only of those who are on the top (Cormack et al., 2006). Contests attract a large community of contestants with diverse skills. The contestants with lower abilities should feel that they can also succeed and that their efforts are recognised. There are many ways how to try to achieve this goal especially through task selection and evaluation. This goal defines the direction of contests.

Recognising achievement is closely related to discrimination of the contestants. *Discrimination* refers to how well the task differentiates between high and low scorers (Dis, 2010). It should be good over a broad range of ability levels (Kemkes et al., 2006). In each contest the gold, silver, and bronze winners have to be nominated. If discrimination is poor among the top contestants, then it becomes difficult to make that decision. The contestants may doubt whether their achievement or luck and other random factors play the most important role in identifying the winners. A good discrimination among the bottom part of contestants is important in order to keep those contestants motivated.

LitIO emphasise educational goals of informatics contests, especially dissemination of good programming practices. An important reason for that is that many teachers in Lithuania take LitIO evaluation practice as a model for teaching students of programming. However, this goal may not be treated as a goal in informatics contests in general.

A high scientific quality of informatics contests, good discrimination of contestants, contests attractive to the general public, new and current contestants, recognition of the best ones in the discipline, making the contests more gender neutral and thereby more attractive to the female students are important characteristics of informatics contests to be achieved (Boersen and Phillips, 2006; Cormack et al., 2006; Pohl and Polley, 2006). Fig. 2.1 and 2.2 provide an overview of goals important in LitIO.

2.4 Background of Concept of *Informatics Contest*

Informatics contest is one of the key concepts in the thesis. The purpose of this section is to provide the background for this concept. Note, that *informatics contest* is a very general term. However in this thesis we define and use the very general term for specific purposes.

(Pohl, 2006) suggested the definition of *a typical informatics contest*:

"The typical informatics contest is a task-based contest with short-time exam sessions, where task solutions are submitted as source code only and evaluated automatically" .

Automated evaluation is a wider concept than *black-box testing.* However since black-box testing is the dominant form of evaluation in informatics contests, some sources use these words as synonyms in the context of informatics contests.

We modified the definition of *the informatics contest* for the use in this research:

- It is a problem solving task-based contest.
- The contest is organised in a form of short term exam session(s).
- The tasks are such that there are known correct and efficient algorithms for solving the tasks.
- An algorithm-code complex is an obligatory part of submission.
- Black-box testing is included into evaluation.
- Each submission is graded, the score is assigned, and the final ranking based on the scores is delivered.

This is a generalised concept and each concrete contest has its own peculiarities.

The extended definition (illustrated in Fig. 2.3) used in this dissertation is wider than that of Pohl's mainly in two aspects: it requires to include an algorithm-code complex in submission, however allows additional materials to be included into submission. Even though black-box testing remains obligatory in the evaluation, it makes room to introduce other forms of evaluation as well.

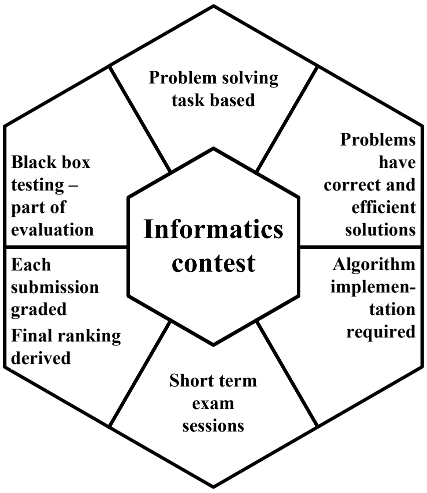
Problem solving task based
Problems have correct and efficient solutions
Algorithm implemen-tation required
Short term exam sessions
Each submission graded
Final ranking derived
Black box testing – part of evaluation
Informatics contest

Figure 2.3: Definition of the informatics contest

The obligatory requirement for a problem to have fully correct and efficient solution means that heuristic tasks are not part of the investigation.[1] The definition of a task is adapted to the scope of investigation of this dissertation, i.e., it does not consider other types of tasks which would be natural to discuss if informatics contests were analysed in a broader context.

There is a variety in naming of *informatics contests*, used both in the scientific papers and the competition communities. The word *informatics* sometimes is replaced by *programming*, *algorithmic*, *computer science* or *computing science.* The word *contest* sometimes is replaced by *olympiad* or *competition.* Naming of this kind of events is more a matter of a tradition, scope, prestige, but not of the content.

The above defined understanding of informatics contest will be used throughout the thesis and will not assume any other types of informatics contests unless explicitly stated otherwise. We did not try to define an ideal informatics contest, we just generalised the concept in view of the currently existing contests and adopted it to this research.

[1]In reality in LitIO the use of such tasks is highly avoided or at least is the subject of serious discussions.

2.5 Structure of LitIO

The contests have many dimensions, like coverage, location, divisions, audience, rounds, duration, evaluation, score announcement, ranking factors, and other dimensions which form taxonomy of the contests (Pohl, 2006; Trotman and Handley, 2006). The structure of each contest has its own peculiarities. In this section we present the structure of LitIO. A detailed description of LitIO structure can be found in (Dagienė and Skūpienė, 2007).

We describe here the structure of the final round, because namely this round corresponds to the definition of the informatics contest accepted in the thesis. The contest is organised in a form of three five-hour exam sessions, held on separate days. The contest is individual and each contestant is assigned a separate computer for the time of the contest. The regional winners participate in the first session which is held on-line. Only the best contestants of the first session are invited to participate in the two on-site sessions (called finals).

At the beginning of a session the contestants get from two to four tasks to be solved during the session. Each task requires to design and implement an algorithm in one of the allowed programming languages. Currently the languages allowed in LitIO (as well as in IOI) are Pascal, C/C++. Other types of tasks are rare in LitIO (as well as in IOI) and not part of this investigation.

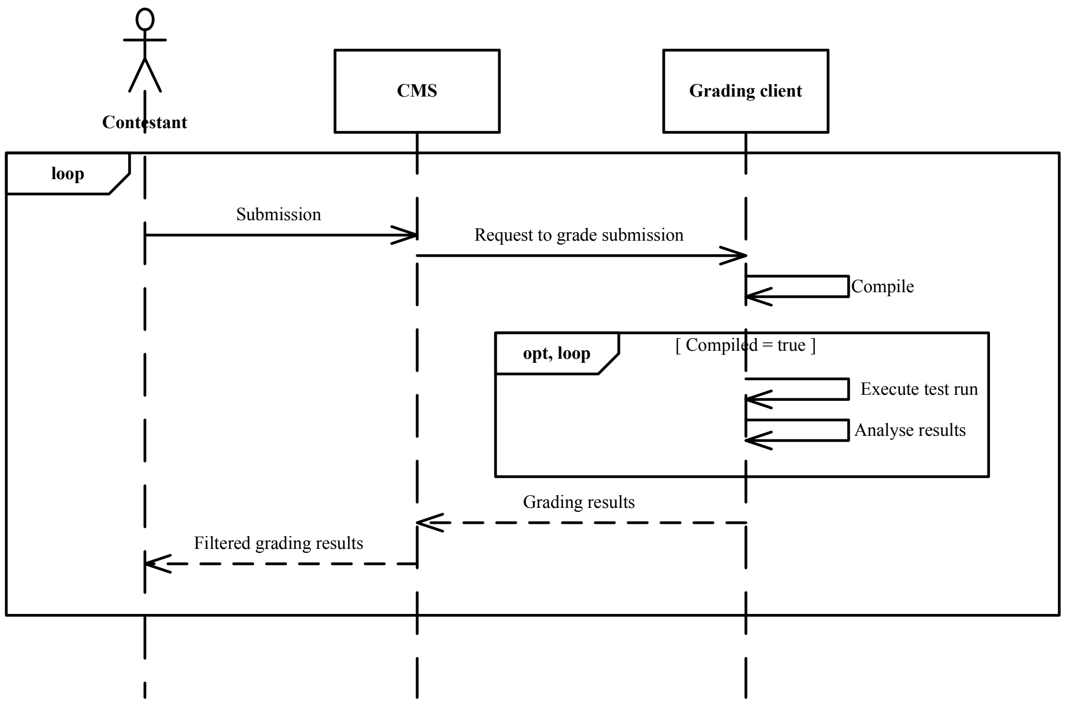

Figure 2.4: Sequence diagram of the submission process

The contestants have to submit their solutions for evaluation using *the Contest Management System* (CMS). CMS is a group of server applications and modules to support informatics contests. The main functions of CMS are to support the contest by providing submit, test, print, backup, restore facilities during the contest and to support automated grading of contestants'submissions (IOI, 2002; Mareš, 2007). Modern CMSs provide more facilities.

After the submission has been submitted, the program is compiled and executed with sample tests. An immediate feedback is provided to the contestant, and they can modify their programs and resubmit (Fig. 2.4). No penalty is given for resubmission. Submission consists of the algorithm-code complex, and the algorithm idea description if required by the task.

Full feedback tasks have been introduced recently to LitIO following IOI developments. For those tasks all the grading tests are public and the contestants can see the points for black-box testing immediately[1]. Evaluation of submissions to other tasks from the point of view of contestants takes place after the contest. The scores of a contest session have to be announced within few hours after the session is over. Official rankings and the winners are announced after the contest.

LitIO is an IOI type contest and its structure corresponds to the IOI structure with one major difference. In IOI, the submission consists of the source code only and no additional materials are required. There are more differences between LitIO and ACM-ICPC type contests. Those are team contests. Each resubmission in ACM-ICPC is penalised, evaluation takes place during the contest, the team scores are announced on a live scoreboard.

2.6 Domain of Problems in Informatics Contests

The domain of problems of informatics contests can be roughly characterised as *"programming problems involving college-level computing and mathematics, as well as associated fields such as operations research"* (Cormack et al., 2006).

Russian Informatics Olympiads have a syllabus (Kiryukhin, 2007). However, many informatics contests do not have a formal syllabus. It is not always possible to find domain of contest problems in the contest documentation or publications. We found just general reasoning, (e.g., *"problems should be interesting, novel, should require solutions demonstrating interest in computer science or they must be computer science problems and not require knowledge from other disciplines"* (Trotman and Handley, 2006)) and considerations about topics which should be excluded from the contests (e.g. *"problems in obscure application areas, numeric problems involving extensive computations with floating-point values"* (Deimel, 1984)).

An informal syllabus of informatics contests can be found in (Skienna and Revilla, 2003) where the problems from a variety of previous contests are analysed and categorised into thirteen groups including arithmetics and algebra, graphs, and dynamic programming.

[1]The contestants do not get information about the points scored by their submissions for full feedback tasks in IOI unless the submission solved the task completely.

IOI seems to be the only contest among the well-known international informatics contests that made the efforts towards an official syllabus. The authors of the recently proposed IOI syllabus tell that *"the competition problems are algorithmic in nature"*. They present a proposal for an IOI Syllabus, divided into four main areas: mathematics, computing science, software engineering, and computer literacy (Verhoeff et al., 2006). LitIO officially follows the proposed IOI syllabus.

The syllabus includes the following areas of computing science: fundamental programming constructs, algorithms and problem solving, fundamental data structures, recursion, basic algorithmic analysis, algorithmic strategies, fundamental computing algorithms, advanced algorithmic analysis, and geometric algorithms (Verhoeff et al., 2006).

Mathematics is inseparable from computing science and it plays several roles, in particular, the role of a language for expressing formalised models, that of reasoning about models, computations, algorithms, data structures and their implementations (Verhoeff et al., 2006). *"Concepts that lie beneath informatics problems are often mathematical in nature and for harder problems students need a sound mathematical mindset to succeed"* (Burton, 2007). It can also play the role of a problem domain, however, it is still a computational problem in its nature.

Note that software engineering and computer literacy are not the domains for the problems of informatics contests. The skills in software engineering are needed in order to implement an algorithm. The syllabus states that *"the application of software engineering concerns the use of light-weight techniques for small, one-off, single-developer projects under time pressure"*. The elements of computer literacy, included into the syllabus, refer to the basic skills needed to use a standard computer with a graphical user interface and the provided program development tools.

2.7 Structure of a Batch Task

The most common, "classical" type of tasks in informatics contests is batch tasks. In a batch task, all the tests are designed and fixed before the beginning of evaluation and do not depend upon the program behaviour. A batch task has the following typical components:

The task story. That gives a detailed background of the task, e.g., a precise description of the model. It is often wrapped up in some kind of story.

The task overview. It contains the statement that explicitly tells which problem the algorithm should solve.

Input and Output. These explain a detailed format of input and output files the contestants program must deal with. Sticking to the input/output formats enables automated testing.

Sample input and output. These illustrate simple input scenarios and their solutions. Their purpose is to help the contestants understand the task and the input/output format. However, they are considered as a supplementary material for the task, i.e., the task should be understandable without sample tests.

Data constraints. These provide constraints on input in general, and concrete upper and lower boundaries to each input item. In some cases, input constraints are provided in the form of output constraints (e.g., input will be such that output will not exceed the value x).

Technical constrains. These contain the memory and run-time limits. All this combined with data constraints give an opportunity for the contestant to estimate the expected performance of his solution.

Deliverables. These explain what should appear in a submission.

Scoring function. It explains to the contestants how the points are to be distributed. For example, if the task contains sub-tasks, in this section, the points for each sub-task are explicitly defined. In the case of LitIO, this section also contains a distribution of points for each evaluated item (verbal algorithm description, black-box testing, programming style).

An example of a batch task can be found in Appendix A.1.

Batch tasks are much more common than other types of tasks in LitIO and IOI. In LitIO during 1989-2010, 243 tasks were prepared and used in the final round. Out of them 226 tasks (93%) were batch tasks. In IOI, 81% of tasks (87 out of 107) were batch tasks since 1989 (i.e., in the years 1989-2009). In other informatics contests which are modeled after IOI, e.g., Baltic Olympiads in Informatics, batch tasks also dominate (Poranen et al., 2009).

Batch tasks do not cover all the types of tasks that are used or considered to be included into informatics contests. Other types of tasks used in LitIO and IOI are theoretical, interactive, and output-only tasks. Interactive tasks are tasks where the program has to interact with some libraries and some output must be produced before the new input comes. In terms of evaluation, interactive tasks are very close to the batch tasks.

Detailed analysis of types of tasks, used in LitIO an IOI, as well as considerations on different types of tasks can be found in (Burton, 2007; Dagienė and Skūpienė, 2003; Kemkes et al., 2007; Pohl, 2007; Verhoeff, 2006, 2009)

In the thesis we focus on the evaluation of batch tasks assuming that most of it will be applicable to interactive tasks, but will not go into the peculiarities of interactive tasks.

2.8 Concept of Quality

In this section we explore the concept of quality. This is needed in order to be able to define quality of a submission.

There are two different trends how the concept of quality can be treated (Hoyer and Hoyer, 2001). Either quality is conformance to specification, or correspondence to the needs of the customer. In the first case, the product is considered to be qualitative if its measurable characteristics correspond to the requirements defined in advance. In the latter case, the quality is the ability to conform to the needs of a customer and is not related to any measurable characteristics. An extensive overview of different views about quality and software quality can be found in (Lundberg et al., 2005).

The quality is neither luxury, nor elegance. The quality is a strict conformance to specifications. The quality standards should be precise and the view ”almost good” is not acceptable (Crosby, 1979). E. W. Deming (Deming, 1988) suggests that quality is conformance to the demands of the user, however a difficulty arises when we have to define the level of quality, i.e., to describe it using measurable characteristics. He suggests that the quality can only be described in terms of an agent, i.e., a concrete judge of the quality (Deming, 1988). A similar view is suggested by A. V. Feigenbaum (Lundberg et al., 2005) who supports the opinion that only the user, during a real use of the product can determine, its quality. However, the problem is that the needs of the user are changing, therefore the concept of quality changes as well. Ishikawa also defines quality as conformance to the requirements of the user and emphasises that international standards (like ISO, IEEE) have some drawbacks and do not respond to the changing needs of the user fast enough (Lundberg et al., 2005).

W. A. Shewart indicates two understandings of quality, i.e., either as an objective reality independent of the existence of the users, or as a subjective perspective dependent upon the thoughts and feelings of individuals which occurred due to the objective reality (Hoyer and Hoyer, 2001).

To sum it up, the concept of quality is not absolute. It is a constructed and changing concept. Therefore it is not possible to talk about the absolute quality of a submission (or an algorithm-code complex). When we speak about the quality of a submission in informatics contests, the two main understandings of quality (correspondence to specification and conformance to the needs of the users) intertwine. The same group of people (scientific committee) both determine the specification and, at the same time, are the only users of the submission (i.e. the jury).

2.9 Different Points of View of Evaluation of Algorithm-Code Complex

The informatics contest is a problem solving contest where the solutions have to be presented as working programs (algorithm-code complexes). This implies that two types of skills are important in the contest and have to be evaluated: problem solving skills (i.e., designing a correct and effective algorithm) and program development skills (Fig. 2.5).

By problem solving we mean *“the use of creative, intelligent, original ideas in combination with prior knowledge when applied to a new situation”* (Vasiga et al., 2008). Informatics contests test the general problem solving skills. In particular, understanding the problem, determining the requirements, planning and designing a solution, implementing the solution and putting it to a test (Salniek and Naylor, 1988).

In informatics contests the problem solving skills and program development skills are interrelated. The submission in LitIO is presented as an algorithm-code complex. We introduced this term to emphasise that it is not always possible to separate those skills and evaluate them separately. A lot of burden in ensuring that the problem solving component were properly included goes to task designers (Vasiga et al., 2008). The task itself should require an original and intelligent approach without which the contestant would not be able to solve the problem completely. Thus, a part of burden for evaluating the problem solving skills is transferred from the jury to task designers. We guess that this might be one of the reasons why black-box testing has a strong position in the evaluation in the informatics contests despite its limitations.

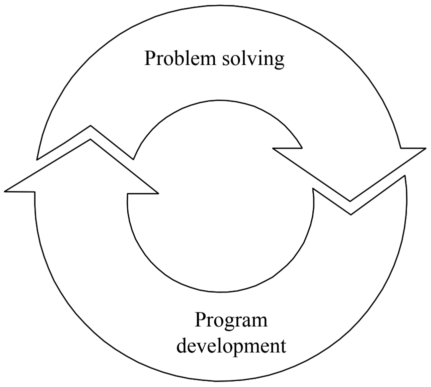

Figure 2.5: The skills measured in informatics contests

Another point of view about evaluation in the contests is the form of evaluation. There are two major approaches to the form of evaluating programming assignments: static and dynamic. The dynamic analysis is based on the observation of program behaviour during its execution. The static analysis involves the analysis of the program without executing it (Ala-Mutka, 2005). Once the measures obtained from the static and dynamic analyses are associated with the scores, we obtain the static and dynamic evaluation.

Another approach distinguishes the following categories: automated and manual evaluation. *Manual evaluation* is evaluation which is performed by human evaluators. *The automated evaluation* is a method in which a computer program aids the teacher in grading student's work and facilitates the feedback process. It can be *semi-automated* where the teacher does (part of) work, but the tool simplifies the process (Jackson, 2000). The basic requirement for the automated evaluation is measurability of evaluation targets (Ala-Mutka, 2005).

It should be noted that the term *automated evaluation* is a much broader term if used outside the context of evaluating programming assignments. For example, automatically processing multiple choice questions is also an example of automated evaluation.

In the dissertation, we assume that each form of evaluation falls under the corresponding category (Fig. 2.6), i.e. it is either dynamic or static, and either automated or manual. It is more difficult to decide on a semi-automated approach. By formal definition, evaluation of idea description and programming style is semi-automated in LitIO, however, conceptually it should be considered as manual evaluation. Therefore in each case we will specify its meaning.

We felt that we had to present one more point of view about evaluation. We will look at the main parts of programming assignments, the evaluation of which we have found discussed in the related publications. At this point we do not make any references about *should be*. We just claim that we have found these parts discussed in the publications and it will be easier to follow the discussions if we know the list of items in advance. The main items are:

- *Written documentation.* It might include reasoning on algorithm correctness and efficiency and the comments on program design.
- *Algorithm implementation.* Evaluation of implementation might be split into the following attributes: *correctness, efficiency* and *programming style.*
- *Set of Tests.*

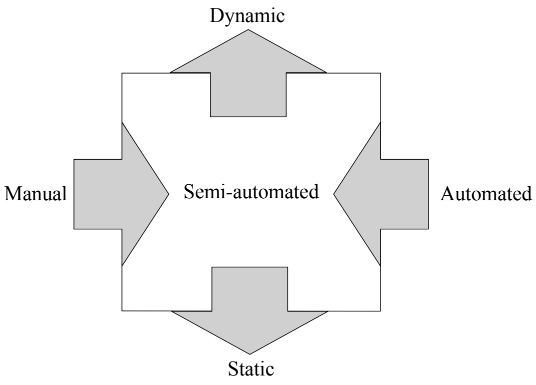

Figure 2.6: Forms of evaluation of programming assignments

We listed four different perspectives: skills that are to be evaluated, the extent of evaluation automation, dynamic or static evaluation, and the list of the main parts of the algorithm-code complex that are evaluated. Two out of four perspectives refer to the implementation of evaluation and the other two refer to the conceptual parts of evaluation. We will look at evaluation from the latter perspective. One section will be devoted to the automated evaluation, because its development and availability of tools play a significant role in the choice of the scheme of evaluation.

2.10 Current Evaluation Scheme in LitIO

The current evaluation scheme has been applied in LitIO since 1994 without significant changes. Historically in the first few olympiads some contest sessions were conducted without computers and manual grading dominated there. About fifteen years ago LitIO moved to a combination of semi-automated and automated evaluation. The same scheme with minor changes (which emerged when the CMS were introduced in LitIO) is applied until now.

In informatics contests, two terms might be used to refer to bodies, responsible for task preparation and evaluation. *Scientific committee* is group of people with background in informatics, responsible for informatics contest syllabus (at least informal), the choice of tasks and task preparation for the contest. The term *jury* stands for a group of people with background in informatics responsible for carrying out evaluation in informatics contests. In LitIO we have only one body which performs functions of both scientific committee and the jury. Therefore these words are synonyms if we speak about LitIO. The choice of the term will depend upon the responsibility we want to emphasise.

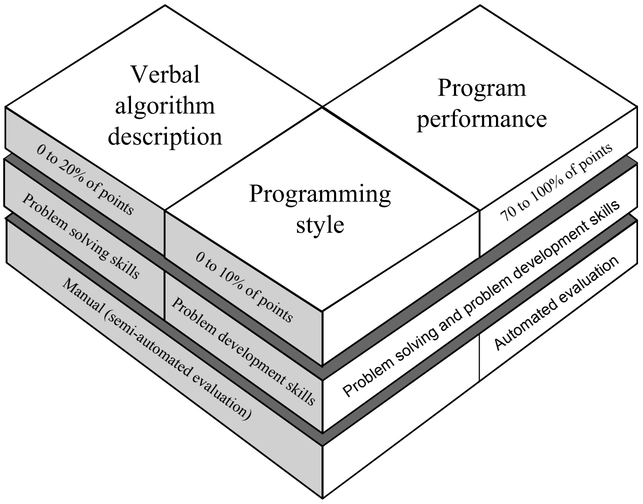
Verbal algorithm description
Program performance
Programming style
0 to 20% of points
70 to 100% of points
0 to 10% of points
Problem solving skills
Problem development skills
Problem solving and problem development skills
Manual (semi-automated evaluation)
Automated evaluation

Figure 2.7: Current evaluation targets in LitIO

We will look at evaluation in LitIO from the point of view of current evaluation targets (Fig. 2.7). A submission in LitIO consists of a verbal algorithm description and the algorithm-code complex. The evaluation is split into three parts: evaluation of the verbal algorithm description (0 to 20% of points), black-box testing of the algorithm-code complex (70%–100% of points), and evaluation of the programming style (0 to 10% of points).

Testing is performed automatically (dynamic automated evaluation), while the evaluation of idea description and the programming style is performed by human graders, using a special tool implemented in the CMS to support this (static semi-automated [manual] evaluation).

The exact scoring function for each task is fixed before the contest and announced to the contestants at the beginning of a contest session. The ranges allow flexibility in the evaluation. It means that there might be tasks where either the verbal algorithm description, or the programming style is not graded. Next we overview each part separately.

2.10.1 Evaluating the Verbal Description of an Algorithm

The verbal description of an algorithm is evaluated using static and semi-automated (manual) evaluation (Fig. 2.8). By evaluating that the judges evaluate algorithmic problem solving skills.

The contestant is expected to provide a short description of his algorithm and (if needed) a mathematical model. The contestant has to decide himself whether the model should be provided. For example, sometimes modeling task data as a graph is not obvious and the modeling procedure has to be described before describing the algorithm itself. The algorithm should be unambiguously clear according to this description. A strict scientific proof is not required. Because that can not be expected at the secondary education level.

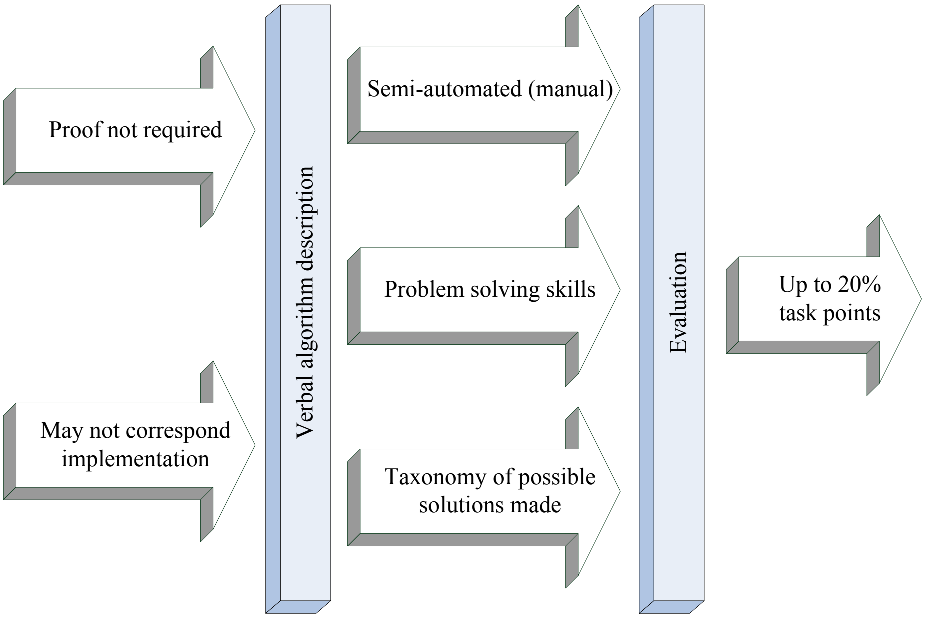
Proof not required
May not correspond implementation
Verbal algorithm description
Semi-automated (manual)
Problem solving skills
Taxonomy of possible solutions made
Evaluation
Up to 20% task points

Figure 2.8: Evaluation of verbal algorithm description

Verbal descriptions are graded by human evaluators. Taxonomy of possible solutions is made in advance for each task. The solutions are sorted into several categories, depending upon the algorithmic strategy, e.g. greedy strategy, full-search, dynamic programming, etc. The jury associates each category with some range of points, depending upon the correctness and the big O characteristic of the algorithm. Thus the concepts of *efficient* or *inefficient* solution for this task is defined. Typically incorrect solutions for verbal algorithm description are awarded no more than 30% of points, correct, but inefficient – 30%-60%. Correct and efficient solutions are awarded 60%-100% of points. However, other distributions are also possible.

Sometimes there arises a situation where without a scientific proof, it is impossible to decide about the correctness of the described algorithm. In this case, the jury still has to decide on the score, sometimes taking a more attentive look at the implementation and testing results (if that corresponds to the written description).

Another common situation is when algorithm descriptions are not clear enough to identify the algorithm unambiguously. In that case the jury try to identify which parts of algorithm are clearly described and base their score on that.

There is no requirement to the implementation to correspond to the verbal description. There are two reasons for that. One of them is that checking whether the algorithm described matches the implementation is not obvious and time-consuming. Another reason is that the contestants are allowed to submit algorithm descriptions without implementations. The contestants who did not have enough time to solve all the tasks can present just the ideas how to solve the problem and get points if their ideas are reasonable.

2.10.2 Testing Functionality and Efficiency

Functionality of an algorithm-code complex is defined as the feature of an algorithm-code complex to terminate and provide correct output for every valid test given as an input.

Efficiency is a characteristic of an algorithm-code complex described as run time performance and memory usage in the worst case and expressed in big O notation. Task designers associate big O characteristic to a linguistic scale (like *inefficient*, *low efficiency*, *efficient*) and the table of the expected scores for each task. This association is a constructed notion.

Testing the functionality and efficiency of a submission is performed using the dynamic automated evaluation. Both problem solving and program development skills are evaluated that way.

Task designers develop a taxonomy of possible solutions and associate it with the expected ranges of points. After they design set of *correctness tests* and a set of *efficiency tests*.

Correctness tests are tests designed with the intention to check algorithm-code complex correctness in order to separate correct complexes from incorrect ones. Typically the size of correctness tests is limited so that reasonably inefficient correct solutions would pass these tests. Various modifications of input data are taken into account during the test design in order to ensure better testing. *Efficiency tests* are designed with the intention to check the algorithm-code complex efficiency and distinguish between different efficiency categories of solutions

The points for each test are distributed in advance, based on the task taxonomy. It is expected that testing will distinguish different classes of solutions. Grading tests often are not disclosed to the contestants before the contest is over.

The contestants submit their source code through the CMS web-interface during the contest. The source is compiled and if the compilation is successful, the program is executed with sample and sometimes with additional tests and the output is checked for correctness. Thus, it is verified that the program performs something reasonable and that input/output format is correct. In the case of use of additional tests, the contestant gets more feedback about the functional correctness and efficiency of his algorithm and implementation. The feedback is given to the contestant immediately. In the case of errors, the contestant can fix and improve his program and resubmit it again. Only the last submission is forwarded for the formal grading.

Testing from the point of view of a contestant takes place after the contest. The only exception is recently introduced full feedback tasks. The purpose of such tasks is a better differentiation of contestants. Such a task should differentiate the contestants who have good programming skills, but are weaker in problem solving. Typically such tasks do not require strong problem solving skills. Full feedback tasks are tested and graded during the contest in LitIO.

During the grading each algorithm-code complex is compiled and executed with each test run (input). The test run is considered to be passed if and only if: the program does not raise any exceptions while being executed with this test run, the execution is completed within the pre-defined time and memory constraints, the program output satisfies the output format requirements, and the output is correct for the given input.

There is no requirement for an algorithm-code complex to pass all the correctness tests in order to be tested with efficiency tests.

Until 2008, grouping was not used in LitIO. It means that partial scoring (adding up points for each test) was used for score aggregation. Test grouping together with all-or-nothing batch scoring was introduced in 2008. Tests are grouped into test cases, each test case targeting at some specific feature. The points for a test case are only assigned if all the tests from that test case are passed successfully.

However, among the jury of LitIO there is no unanimous opinion on test grouping. Because sometimes it is difficult to define the disjoint correspondence between the domain of input and the groups of possible solutions (including incorrect ones). Overlapping might lead to the consequences where a submission is punished several times for the same mistake. Thus, the points awarded for black-box testing, fall out of the range defined in the task taxonomy. Therefore, for some tasks partial scoring is still applied.

2.10.3 Evaluating the Programming Style

The programming style is evaluated using static semi-automated (manual) evaluation. In the evaluation, the program development skills are evaluated.

The programming style has been part of the evaluation scheme in LitIO for many years. After the set of tasks for a contest session has been prepared, the jury discusses and approves the scoring scheme for each task. A part of this process is deciding whether to include the programming style into the scoring scheme of a concrete task. When making the decision, the jury takes into account the amount of solutions to be graded, available resources for grading, and the nature of a task. On the average, the programming style is included into the scoring scheme of half of the tasks.

If the programming style is evaluated, then the following scoring scheme is used. 90% of points are given for the testing results and verbal description of an algorithm, the remaining 10% points – for the programming style, i.e., program elegance, structure, and simplicity. Those points can only be awarded if the program scores $\geq 50\%$ of points for automated testing, in order to avoid awarding points for programs that do not even try to solve the task. Grading is performed by human graders (jury) semi-automatically, using a special module of the CMS.

Some formal criteria are developed in LitIO, that give guidelines for the evaluators. The criteria also serve as guidelines for the contestants. However, there is no formal evaluation formula relating metrics (of the programming style quality according to each criterion) to the points for the programming style. Therefore the current grading practice should be considered as holistic.

One of the main requirements is consistency everywhere in the program: in text formatting, naming, processing data, control structures, etc. Other basic requirements are: neat and clear text formatting, text indentation revealing the program structure, spaces used to give more clarity and suggest grouping, appropriate use of comments, descriptive names, reasonably selected and used data structures, and the structural program (separate groups of computational steps are separated into procedures or functions). These requirements were based not only on the general requirements to the programming style in (Kernighan and Pike, 1999; Miara et al., 1983), but also on the most common offenses to the programming style the contestants make in LitIO.

2.11 Automated Evaluation of Submission to Programming Assignments in Programming Courses

Originally an automated evaluation was developed at universities for evaluating submissions to programming assignments given in the programming courses, and a lot of research is designated to that. We will analyse it in this section.

We use the term *programming assignment* when we refer to the tasks given in the programming courses.

Technically both in informatics contests and in the programming courses we deal with an algorithm presented in the form of implementation. However, a task in an informatics contest is conceptually different from programming assignment. The tutor evaluating the submission to programming assignment does not have to concentrate on eliciting what kind of algorithm is implemented. On the one hand, the programming assignments correspond the course curricula. On the other hand, the materials submitted for evaluation in programming courses, are treated differently because of the nature of the event (learning process versus exclusive contest event).

2.11.1 Development of the Automated Evaluation of Programming Assignments

The roots of necessity of the automated evaluation are similar both in informatics contests and in programming courses. Programming problems and assignments are considered essential elements of software engineering and computer science education courses (Douce et al., 2005). Academic institutions face the challenge of providing their students with a better teaching quality. Simultaneously they need to decrease the amount of additional work for the staff. As a consequence of that, huge numbers of programming assignments (programs) have to be evaluated and provided with feedback in a short period of time (Joy and Luck, 1995).

The schedule is even tighter in informatics contests. It can be estimated that over a thousand of submissions have to be evaluated in a few hours in IOI. Therefore at present it is considered that there is no alternative to automated evaluation neither in informatics contests, nor in programming courses (Kemkes et al., 2006).

Automated evaluation tools of programming assignments share common features like speed, consistency, and availability of evaluation (Ala-Mutka, 2005). A comprehensive overview of automated evaluation systems was presented in (Colton et al., 2006; Douce et al., 2005).

The earliest examples of the automated evaluation system can be found in (Hollingsworth, 1960). The next step was a system, where automated evaluation was applied in testing beginner's student programs written in Algol. Routines had to be written for each task. The tests were randomly generated and the programs were executed with those tests. The output was checked for correctness (Forsythe and Wirth, 1965). New ideas were introduced in (Hext and Winings, 1969). This system could already compile and run programs without a human intervention and it was testing each program with two tests. It had implemented the scoring policy, i.e., assigned points for a successful compilation, a short running time, etc. Those earliest first generation automated evaluation systems already had the most important features and demonstrated the power of automated evaluation. Using automated evaluation tools of the first generation required a certain qualification and experience.

Automated evaluation systems of the second generation were tool-oriented systems (Douce et al., 2005). They were developed using existing tools. The focus of such systems was the same as in the earlier systems, i.e., functional correctness of submitted programs. Some second generation systems already had already implemented a remote submission and use of a network (Benford et al., 1995; von Matt, 1994). The automated evaluation systems started to support grading with generation of grading reports, that allow the tutors to assign the weights to the tests.

The third generation automated evaluation systems can be called as web-oriented tools. They use web-technology, adopt more sophisticated testing approaches (e.g. "diagram" evaluation in *CourseMarker* (Cou, 2010)), support many programming languages, automatically evaluate the program design, provide a rich feedback for the student, introduce plagiarism detection, etc. (Douce et al., 2005).

Note that there were developed many informal grading systems, where it is difficult to transfer the results among institutions and even among course instructors (Edwards, 2003).

Development of automated evaluation in informatics contests has some parallels. We have not found a published overview of the development of evaluation systems in informatics contests. However we observed the appearance of tool-oriented evaluation systems and their development into web-based CMS in LitIO and IOI (Skūpienė, 2004). Earlier systems were more limited technically. For example, they did not provide real-time feedback during the contest. The contestants had no aid in detecting errors related with format specification (e.g., the wrong file name or extra space at the end of the line). As a result, those errors had more weight in informatics contests than they were supposed to.

Modern web-based contest management systems (IOI, 2002; Mareš, 2007) are supplied with many features like real-time feedback during the contest, contest management features, analysis mode after the contest, etc. They have improved the quality of contests in many aspects. However, the main concern of applying black-box testing to the evaluation in informatics contests (i.e., detecting all incorrect submissions and validity of assigning scores to such submissions) remains.

2.11.2 New Role of Automated Evaluation in Programming Courses

In recent years the role of automated evaluation tools in computer science and programming courses has changed significantly.

Looking at the history of automated evaluation in programming courses in universities, we observe a shift of emphasis. The ability to automatically compile, run, and test a student's program and provide the score was most important in the early systems. These remain important issues both for the contests and for the programming courses. However, the emphasis was shifted in different directions in the informatics contests and in the evaluation of programming courses.

A course on a subject (programming) is a lengthy process which involves many assignments, submissions and resubmissions, deadlines, observation of student's performance, progress, and feedback from the evaluator. The grading tool components that support the course management became important (Benson, 1985). Even though the main role remains measuring student's knowledge and skills, the role of a grading tool as a learning device became very significant. The students need supporting learning (and evaluation) environments, because the learning environments help to achieve better learning outcomes (Roberts and Verbyla, 2002). Designing a course and comfortable monitoring of the learning and evaluation process became important features of learning environments and tools.

Automated evaluation systems try to solve a number of other issues that are outside the scope of this investigation, for example, plagiarism detection, evaluating programs with graphical interfaces, performing the formative assessment of programming assignments, evaluating the automated programming assessment with respect to the stated objectives, student's knowledge of language constructions, analysing the program structure in order to identify whether the program followed the given skeleton, supporting different types of assignments, parameterising programming problems, and peer-assisted automated evaluation. These other issues were discussed in (Amelung et al., 2006; Benson, 1985; Carter et al., 2003; Douce et al., 2005; Lewis and Davies, 2004; Malmi et al., 2002; Pardo, 2002; Saikkonen et al., 2001; Woit and Mason, 1998).

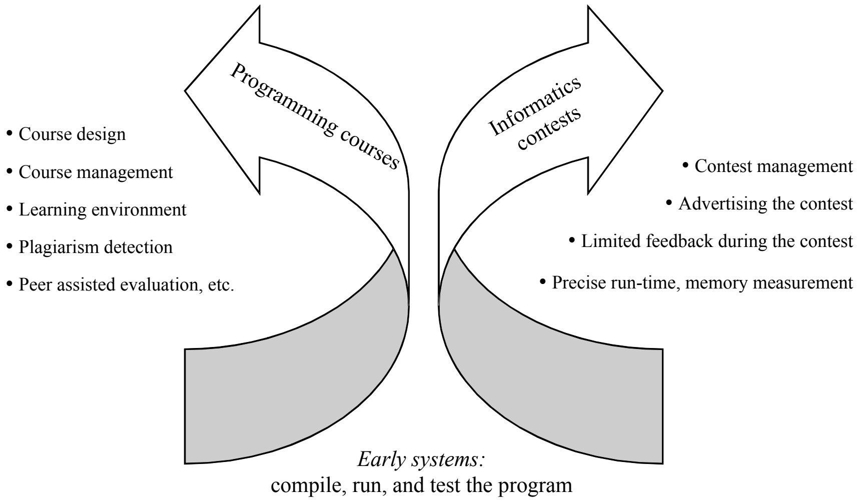
Programming courses
Informatics contests
• Course design
• Course management
• Learning environment
• Plagiarism detection
• Peer assisted evaluation, etc.
• Contest management
• Advertising the contest
• Limited feedback during the contest
• Precise run-time, memory measurement
Early systems:
compile, run, and test the program

Figure 2.9: The directions of development of automated evaluation in programming courses and in informatics contests

The new roles require additional features of the environments and they have become an active topic of research in the area of computing science education. On the contrary, that did not become an active topic in informatics contests and they are not relevant in the context of this research.

The CMS does not have to perform the role of a learning tool. Some other features (e.g., evaluating programs with graphics) might become relevant if the format of the contest were different. The attitude towards the support and feedback for the contestants is different. The contestants are provided support from the CMS on the issues that might distract them from concentrating on the algorithm. For example, the CMS detects output formatting errors or provides run-time information. Providing too much feedback might conflict with the nature of the contest as an event. With the growing speed of processors, precise measurement of program execution time becomes highly important in informatics contests as this is directly related to the ability of the system to distinguish between different efficiency classes of solution. Advertising the contest (e.g. the ability to demonstrate the scores of the contestants on a live score board for the spectators during the contest) is another new expected feature of CMS.

In this section, we have showed that the research of automated evaluation in computer science education is relevant and active. However, the direction of the research is different from that in informatics contests (Fig. 2.9). The number of recent publications directly related to the evaluation of programming assignments is very limited. That was also mentioned in (Ala-Mutka, 2005).

In the subsequent subsections, we will look over the automated evaluation experience in informatics contests and programming courses. We discovered the experience of automated evaluation of three items of programming assignments, and a separate subsection will be devoted to each. They are: automated evaluation of correctness and efficiency (Subsection 2.11.3), automated evaluation of the programming style (Subsection 2.11.4), and automated evaluation of test sets (Subsection 2.11.5).

2.11.3 Evaluating Programming Assignments by Testing

In this subsection we will look through some aspects of applying black-box testing in the evaluation of programming assignments. We located just a few sources and the most extensive reference is (Ala-Mutka, 2005).

We did not find any extensive discussions in the published papers based the fact that testing cannot be used to prove the program correctness (in our case the algorithm-code complex correctness) (Dijkstra, 1972).

We suggest that one of the reasons is the difference in the difficulty of tasks. The tasks at high level informatics contests might be a real challenge even for graduates of computer science studies. Therefore heuristic approaches are common among the submissions of contestants. They are incorrect, and it is rather difficult to detect all of them by black-box testing (Verhoeff, 2006).

The situation is different with the course assignments. Much research was devoted to the automated evaluation of introductory programming assignments (Califf and Goodwin, 2002), which are much easier if compared to the contest tasks. The assignments have to reflect the syllabus and should be solvable after taking the course.

Despite inability to prove program correctness, testing still can show the absence of *known errors* (Leal and Moreira, 1998). For simple assignments (e.g. sorting an array) that are routinely given to the students, it is much easier to decide on the *known errors* and make tests against them. This might be the reason why we did not find discussions about the ability of black-box testing to detect errors in programming assignments.

In the informatics contests each task is (expected to be) original, requires problem solving skills and more complicated techniques. Therefore the concept of *known errors* remains rather vague in the informatics contests.

Assigning the score to incorrect solutions is a questionable issue both in evaluating the programming assignments and submissions (Verhoeff, 2006). The concept *how close* the algorithm-code complex is to the correct solution is a subjective judgement. The subjectivity arises either from the human grader or from the nature of black-box testing. In general, the black-box testing it does not expose neither the nature, nor the scope of error). The practice of applying all-or-nothing scoring with a possibility of resubmission is acceptable for regular programming assignments

(Colton et al., 2006). In the case of failure, the students might be given the failed test and have to fix their solutions. This is the essential difference from the practice of informatics contests. In the contests, the test set is fixed before the contest and the same set is applied to every submission to ensure the same testing conditions. While in the evaluation of programming courses, it is usual to apply randomly generated tests for evaluation and different students might be tested by different sets of tests (Colton et al., 2006). Such a practice is not directly applicable in the informatics contests.

We have found one more interesting approach that correlates with informatics contests. It deals with measuring the solution complexity (efficiency). This is the experience of assessing individual procedures rather than programs. The automated evaluation system *Scheme-robo* was developed and the experience of assessing simple assignments in introductory programming courses was presented (Saikkonen et al., 2001). (Hansen and Ruuska, 2003) have implemented it by giving the students an input/output module for the assignments that concentrate on efficient data processing algorithms. Calculating the number of times, certain structures inside the program were executed, and comparing the results to model the solution was implemented on *CourseMaster* and *Assyst* systems (Foxley et al., 2004; Jackson and Usher, 1997).

A similar suggestion to use such a metric in the informatics contests was presented by (Ribeiro and Guerreiro, 2009). They address the difficulties related to measuring the efficiency. As the computer power increases, the size of input has to increase in order to separate the solutions of different complexity. Data increase causes other problems. Measuring behaviour of some structures within the program might be a solution in this case. The paper suggests asking to submit functions (procedures) rather than programs and repeating the same function call several times to increase clock precision. Thus input size, which nowadays has become too large and started causing problems, is decreased. Curve fitting analysis is proposed to be used to estimate program complexity rather than referring to the number of passed test cases. However experiments and the corresponding software are required before the proposal can be included into the evaluation scheme.

In the subsection, we presented a few examples of similar issues which occur in both contexts. On the one hand, this shows that concerns about black-box testing are not so active and severe in the evaluation of programming assignments. We did not discover the experience that could be directly transferred to informatics contests. The suggested different measurement of the algorithm-code complex efficiency is interesting and potentially applicable in informatics contests. However, the implementation and piloting require a separate study and therefore it falls outside the scope of this thesis.

2.11.4 Automated Evaluation of Programming Style

From the observations in the previous sections we have concluded that much of research in the area of automated evaluation in the programming courses is outside the interest of evaluation in the informatics contests. However, we have found an area where the experience of automated evaluation in the mass programming courses might be transferred to the informatics contests.

It is the automated evaluation of the quality of program design. This involves performing a static analysis and checking the program source against a set of characteristics. Many grading tools were designed that perform a static analysis and use software metrics to check readability, maintainability, and complexity of the source code (Ala-Mutka et al., 2004; Hirch and Heines, 2005; Jackson, 2000; Leal and Moreira, 1998; Spacco et al., 2005). Such tools were applied in evaluating programming assignments in universities. However, we found no evidence of such automation being applied in informatics contests.

The ability to write nice and elegant programs is already a skill and a very important skill which is not usually the focus of computer science and programming courses (Kernighan and Pike, 1999). It is easy to make a small program working despite a bad style. Students often treat the programming style as secondary, not part of the program development process (Schorsch, 1995). That was also noticed both in the context of programming courses and in the context of informatics contests (Douce et al., 2005; Grigas, 1995; Struble, 1991).

In order to measure the programming style, we need a common understanding of programming style. *"A programming style is understood as an individual's interpretation of a set of rules and their application to the writing of source code in order to achieve the aim"* (Mohan and Gold, 2004) that the source code is readable and understandable. It can be said that everything that is related to program clarity, simplicity and generality, is understood as programming style. These types of definitions together with guidelines (e.g. as in subsection 2.10.3) can be applied for holistic approach to evaluate the programming style by human evaluators.

However, in order to introduce the automated evaluation, the elements of the style should be identified and concrete metrics for each element must be defined (Fig. 2.10) and associated with ranges of the expected values (Hirch and Heines, 2005).

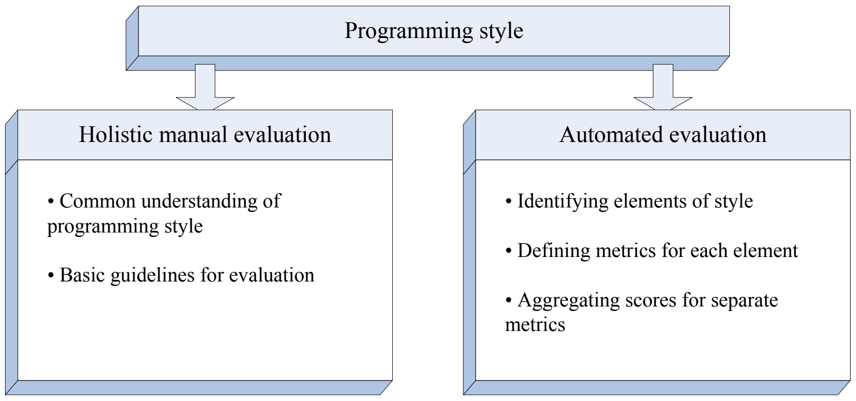
Programming style
Holistic manual evaluation
• Common understanding of programming style
• Basic guidelines for evaluation
Automated evaluation
• Identifying elements of style
• Defining metrics for each element
• Aggregating scores for separate metrics

Figure 2.10: Holistic versus automated evaluation of programming style

The first tools for the automated evaluation of programming style were created in the eighties. Then the basic guidelines for the programming style were created.

One of the early systems was a *Style* system (Rees, 1982) for automated evaluation of the programming style of Pascal programs. The system had ten style measures that could be easily calculated. They were: average line length, percentage of comment lines, numbers of *goto's*, average length of identifiers, use of blank lines as separators, etc. The scoring scheme included five parameters for each metric which defined the conversion curve presented in Fig. 2.11. The parameter *max* specifies the maximum score to be awarded for each measure. If the value of the measure lies in the range defined by *lotol* and *hitol* then the maximum score for that measure is given. If the value of the measure is lower than *lo* or higher than *hi*, then the score is zero. For example, it the percentage of comment lines in the source is either very low (basically no comments) or very high (nearly every line is commented) then the score for this measure would be very low. To get a maximum score the amount of comments should be within some reasonable range.

The total score was an aggregate of the scores of separate metrics.

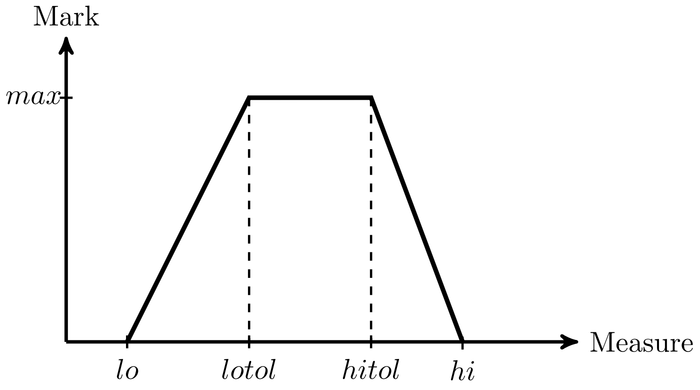

Figure 2.11: Programming style marking scheme suggested by (Rees, 1982)
max – maximum score for the programming style. If the value of a concrete style measure is lower that *lo* or higher than hi then the score is zero.

Later on, a C analyser was developed for evaluating the programming style. It served as a basis for developing other automated evaluation tools (Benford et al., 1995; Berry and Meekings, 1985). The feature of those systems was that the course designers could configure the parameter values for the metrics. The new tools foresee other programming languages (Jackson and Usher, 1997; Leal and Moreira, 1998; Redish and Smyth, 1986) and more metrics. For example, (Dromey, 1995) incorporated 99 metrics for automated evaluation of C programs. A variety of tools and the increasing number of metrics resulted in the classification of measurements and developing taxonomy for the programming style (Oman and Cook, 1990). The taxonomy proposed four stylistic factors. These were: general programming practices, typographic style, control structure style, and information structure style.

Among later systems we could mention *Checkstyle* for automated evaluation of Java programs (Burn, 2003). This is an open source tool that provides an extensive analysis of the source code programming style. The feature of this tool is its modularity. The *Checkstyle* consists of a variety of checks and additional checks can be written to include new metrics.

Earlier available systems either did not cover important features of object-oriented programming or used some obsolete checking which is currently performed by compilers. Therefore a *STYLE++* tool has been created for automated evaluation of the programming style of C++ programs (Ala-Mutka et al., 2004). The tool covers 64 different measures. Metrics were developed that meet the software quality requirements. They also included non-functional quality requirements, such as reliability and efficiency. Four programming style categories have been introduced: transportability, understandability, modifiability, and readability. They were decomposed into nine smaller categories until measurable features of a concrete level have been reached. Scoring is based on the ideas of (Rees, 1982), however, since different courses may require to emphasise different style aspects, the system allows much tailoring, irrelevant measures may be switched off and different weights might be associated with different measures.

Program documentation (which includes proper commenting) can be considered as a separate part of programming style. We discovered efforts to create an automated evaluation tool for evaluating the quality of program code documentation. Even though currently there are no guidelines (and no measurable standards) how to perform such an evaluation, there exist tools that help creating such a documentation. Given those tools students, should be required to produce a qualitative documentation (Hirch and Heines, 2005).

The interest in the automated programming style has lowered if compared to the eighties. The efforts to find modern program style development tools or programming style evaluation guidelines for C++ evaluation for educational purposes were unsuccessful (Ala-Mutka et al., 2004). The accessible guidelines are industrial high-level recommendations for object-oriented program design.

There is one significant difference between the evaluation of programming style in the programming courses and that in the informatics contests. Universities sometimes develop their own standards, they might ask the students to follow some specific programming style standards, while the informatics contests should be open to a variety of programming styles. The contestants do want precision when they deal with getting or loosing points (Grigas, 1995). To ensure equal conditions for the contestants, the evaluation of programming style should be language independent.

Note that evaluation of the quality of program design is supported not by all educators. Design is important if the program works. There is an opinion that, once the students have learned to program, it is easy to teach them good design, but not vice versa (Daly and Waldron, 2004).

It can be concluded that much research has been done in the area of evaluating the programming style of programming assignments in the programming courses, and we found no evidence of any of that being applied in the informatics contests. The main idea of the automated evaluation of programming style is performing the static analysis, calculating different metrics, and associating the expected ranges. We feel that this experience can be transferred to the informatics contests. In order to apply the experience in the informatics contests, the research should be conducted in two areas. Metrics should be chosen and tailored so that they would not favour some programming styles and disfavour the others. Another direction of research is to ensure the evaluation compatibility between different programming languages. This is a potential area for research, but, due to its scope, it could not be conducted within the framework of this thesis.

2.11.5 Automated Evaluation of Test Sets

The ability to create comprehensive and effective test data is part of program development skills. Testing skills are considered as an important skill while performing the programming assignment. Writing a test case enforces a student to model and think how his program will behave.

Even though testing skills are important in software development, most computer science curricula cover it minimally and it is considered to be poorly suited for a course topic (Edwards, 2003). Therefore an alternative approach to teaching testing skills is to re-frame the concept of programming assignment and to require providing a test case together with implementation.

Automated evaluation tools were created that evaluate the quality of test sets. The evaluated items that we met were the test set validity, test completeness (i.e., measuring how comprehensively a test set designed by a student, covered its own program or different execution paths of other programs), and the correctness score (i.e. running test sets against own, others or buggy programs) (Ala-Mutka, 2005; Allowatt and Edwards, 2005; Chen, 2004; Edwards, 2003; Jackson and Usher, 1997).

2.12 Black-Box Evaluation in Informatics Contests

Black-box testing (also referred to as *testing to specification*) is testing that ignores the internal logic of a program and focuses solely on the outputs generated in response to the selected inputs and execution conditions (Williams, 2006).

Black-box testing is a form of automated and dynamic evaluation. It is considered that at present there is no alternative to black-box testing in informatics contests (Kemkes et al., 2006). The procedure of black-box testing, applied in LitIO, was already described in Sections 2.5 and 2.10.2. Variations of the procedure in other informatics contests are also possible. However, in our opinion, the differences are not essential in terms of this research.

The outcome of black-box testing is a vector (test run failed/passed), based on which the score aggregation is performed and the final score is calculated:

$T = (t_1, t_2, \cdots t_g)$, where T is a vector representing outcome of black-box testing, $g > 0$ is the total amount of grading test runs, and $t_i \in \{0, 1\}$ is the Boolean outcome of a concrete test run (i.e., *failed* or *passed*). In some cases t_i might be a record instead of a Boolean value.

It should be noted that the scoring related terminology in some sources might differ from that used in this dissertation.

The structure of this section will be as follows. First we will look over the concerns related to dominant use of black-box testing in informatics contests. The search for better score aggregation schemes is performed with a view to preserve the dominance of black-box testing and, at the same time, to get rid of some of its drawbacks. Therefore we will look through different black-box score aggregation schemes applied or suggested to be applied in informatics contests.

2.12.1 Concerns about Black-Box Evaluation

It was acknowledged long ago, that *program testability*, i.e., the qualitative features of tasks that allow comprehensive and definitive testing of solutions, is important and should be taken into account while selecting problems for informatics contests (Deimel, 1988). It is assumed that now there is no alternative to black-box testing in informatics contests (Kemkes et al., 2006).The main goal of testing might be formulated in the following way: to distinguish correct and incorrect solutions and to distinguish different classes of correct solutions (that are of different difficulty), regardless of the programming language used to implement them (Diks et al., 2007).

However, the actual role performed by black-box testing is described in a different way. *"Hence, if a submission achieves full score, it can be said to reproduce the input-output relation given by the test data – no more, no less. The contestant ... cannot count on a full score to confirm the solution to be perfect"* (Pohl, 2008). This is seconded by (Ernst et al., 2000) who in their paper advise future contestants: *"you do not have to submit a correct program. It only has to produce the right output for the jury input."* Describing the ACM-ICPC ranking procedure, (Skienna and Revilla, 2003) indicate that the winner is the team which correctly solves the most problems and immediately mentions *"at least correctly enough to satisfy the judges".*

The concerns of the use of automated testing to prove the program correctness were known long ago. (Dijkstra, 1972) wrote: *"program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence".* (Forišek, 2006) theoretically shows that there is no (known) way to perform testing efficiently and with a 100% accuracy as this is an NP-hard problem. (Leal and Moreira, 1998) stress that automated testing does not prove program correctness, but the absence of *known errors.*

We identified the concerns related to use of black-box testing for evaluation in informatics contests:

- *Black-box testing is incapable of identifying interesting and original problem solving approaches.* (Leeuwen, 2005; Verhoeff, 2006) indicate the cases where the contestants designed interesting original algorithms (not assumed by task designers), but this was not discovered because of automated grading.

- *Incorrect algorithm-code complexes might not be identified,* i.e., they still may be awarded a full score. (Leeuwen, 2005; Verhoeff, 2006) write that during the analysis of the IOI'2000 task *Median*, 10 out of 226 submissions were not identified as incorrect by black-box testing and for task *Phidias* 57 out of 295 submissions were not identified as incorrect during testing.

- *The expected score differs from the actual score.* An algorithm-code complex where a correct and reasonably efficient algorithm is correctly implemented should score more than an algorithm-code complex which contains the implementation of an incorrect algorithm or incorrect implementation of either correct or incorrect algorithm. However, this is not always achieved with black-box testing (Forišek, 2006).

- *After the error has been discovered, black-box testing does not reveal the nature and the dimension of the error.* The error could have occurred in understanding or analysing the problem, designing an algorithm or implementing it (Burton, 2008). (Verhoeff, 2006) writes that over 30% of task *Median* scores and over 50% of task *Phidias* scores have not been properly justified. (Forišek, 2006) analysed the tasks of the three past IOI's and for most of the tasks detected incorrect solutions that are easy to find, easy to implement and that would score more than expected by the jury. (Kemkes et al., 2006) point out that there is no method to measure the number of cases (i.e., the coverage of input domain by an algorithm-code complex) solved by the algorithm-code complex of the contestant as it depends entirely on the authors of the test cases. This raises a pedagogical question how to assign scores to an incorrect solution and how to justify that one incorrect solution scores more than another. Assigning a partial score is considered rather subjective despite the use of testing (Colton et al., 2006).

- *Too large penalties for minor mistakes.* This problem was addressed in many papers (Burton, 2008; Forišek, 2006; Pohl, 2008). Black-box testing leads to severe punishment of submissions that implement correct ideas, but show slight implementation mistakes (e.g., do not implement border cases properly). (Vasiga et al., 2008) state that time pressure in the contest is high, the challenge to implement and properly test the solution is overwhelming, and, thus, even minor errors can be a reason for nearly a correct solution to obtain very few points. The goal of evaluation should be to achieve that only a small amount of points could be taken away because of small mistakes. Grouping test cases makes this problem even more severe.

- *Scoring of algorithm-code complexes with different efficiency may also be inadequate.* In (Leeuwen, 2005) some programs were discovered that received 60 points instead of expected 100 due to a particular combination of the programming language and solution technique.

- *Black-box scoring does not identify the quality of program design.* Even though that is a problem solving contest, the solutions are communicated as programs. Therefore it is important to encourage good programming practices and make a distinction between programs of different design quality (Andrianoff and Hunkins, 2004; Bowring, 2008; Fitzgerald and Hines, 1996; Sherrel and McCauley, 2004; Struble, 1991).

Concerns regarding the black-box testing based evaluation scheme can be assigned to one of the two categories.

The first category is *the concerns that cannot be expected to solve* by black-box testing. The source for such concerns is the absence of other forms of evaluation. That is why they were expressed in the context of discussions on the suitability of black-box testing. Black-box testing is not supposed to perform certain things such as identifying original algorithmic ideas or giving a feedback on the quality of programming style. This cannot be considered as a drawback of black-box testing as such.

It is a drawback of the evaluation scheme that it does not include any other grading than black-box testing. If these characteristics are considered to be very important in informatics contests the only way to take them into account is to incorporate other forms of evaluation into the scoring scheme.

The second category is *the concerns that the jury expects that black-box testing solves on a satisfactory level.* Despite the warning issued by (Dijkstra, 1972), we often observed expectations that black-box testing should identify all incorrect algorithm-code complexes, provide adequate scoring of efficiency and give hints how to assign scores among the incorrect algorithm-code complexes. (Forišek, 2006) summarises the actual situation in informatics contests by saying that sometimes an incorrect solution scores far too many points, sometimes an asymptotically better solution scores less points than a worse one, and sometimes a correct solution with a minor mistake (e.g. a typo) scores zero points.

There can be different ways how to try to tackle this issue [1]:

- *Include other forms of evaluation.* Providing the reasoning for a design might help identify incorrect algorithms thus ensuring that such algorithm-code complexes are not assigned a full score. This might also help to see the scope of deviation from the expected score.

- *Work towards improving the testing by improving the quality of tests and score aggregation schemes.* (Pohl, 2008) emphasised that the quality and choice of test data influence the score of automated testing dramatically. (Kemkes et al., 2006) proposes ideas how to improve the design of test data. Search for improved score aggregation schemes will become visible in the next subsection, that presents an overview of scoring schemes applied in informatics contests.

- *Search for other options.* Providing more feedback during the contest might help the contestants to minimize the amount of small and subtle errors which result in big loss of points (Vasiga et al., 2008).

Next we will overview different scoring schemes for black-box testing. Searching for improved scoring schemes is one of the ways to improve evaluation, especially in the contests where the evaluation is limited to black-box testing.

[1]There also might be other approaches (e.g., to consider new types of tasks or a different contest format), but they are outside the scope of this dissertation.

2.12.2 Partial Scoring

Partial scoring is a scoring scheme for black-box testing where points are assigned for each test run independently, and the score S for a task is calculated as the sum of scores for each test run.

$$S = \sum_{i=1}^{g} w_i t_i \tag{2.1}$$

Here w_i $(i = 1 \cdots g)$ are the weights of the test runs determined by the task designers in advance.

The weights are such that $\sum_{i=1}^{g} w_i = P$, where P is the maximum amount of points for testing for the task.

Such an approach has been applied in IOI and in many other contests, including LitIO, for many years. This scoring scheme received a lot of criticism (Kemkes et al., 2006; Verhoeff, 2006) as partial scoring adds more random noise rather than addresses the issue, i.e., than evaluates the quality of an algorithm-code complex and results in reasonable discrimination. This scoring scheme was considered especially unsuitable for tasks, where a solution (or a specific answer) could be easily guessed.

(Forišek, 2006) investigated the tasks of three IOI's. The goal was to check whether there is *"an incorrect algorithm that is easy to find, easy to implement (in particular, easier than a correct algorithm) and scores significantly inappropriate amount of points".* They succeeded to discover such tasks.

The reason for that could be the task itself or inappropriate choice of tests. However arguments were presented that partial scoring strengthens this problem. We will present our own research later. We have not found any recent publications in favour of this scoring scheme.

2.12.3 All-or-Nothing Scoring

All-or-nothing scoring is a Boolean scoring scheme which classifies the solutions into two categories: *accepted* (considered to be correct and efficient) or *not accepted* (incorrect or inefficient) without any intermediate values.

This type of evaluation is applied in ACM-ICPC type contests (ACM, 2010a). ACM-ICPC contests are team contests for younger college students where a team of at most three students share one computer to solve a set of 8 to 12 tasks. A submission consists of the program source only. The only form of evaluation applied in ACM-ICPC is black-box testing. The run time and memory limits are typically more generous than that in the IOI type contests.

Each submitted solution is immediately evaluated. In the case of failure on at least one test, the submission is rejected and the team is given some limited feedback about the reason of the failure. However, the contestants do not know the exact number of tests and might not know the number of tests their submission passed successfully. The teams can correct and resubmit their solution.

When the solution is accepted, the team gets a penalty of 1 point for every minute since the beginning of the contest till the moment it has been accepted and

20 penalty points for each rejected submission of this task. Penalty points for all the accepted tasks are summed up.

A submission is accepted if $\prod_{i=1}^{g} t_i = 1$.

The score for an accepted submission for the task is a two-parameter vector $S = (1,\ 20r + m)$ where r is the number of previously rejected submissions and m is the number of minutes since the beginning of the contest till the acceptance of the submission.

The total score S_{Total} for all the tasks is also calculated as a two-parameter vector:

$$S_{Total} = (\sum_{accepted} 1,\ \sum_{accepted} (20r_{task} + m_{task})) \tag{2.2}$$

The ranking is based on two criteria. The primary criteria are the number of accepted solutions. The secondary criteria, used to break ties, are the amount of penalty points (Cormack et al., 2006).

This approach when submission is accepted only if all the tests are passed successfully is called *all-or-nothing.* It avoids one most severe concern causing many discussions – validity of assigning scores to incorrect solutions.

On the other hand it is very strict, adds pressure and might discourage contestants especially in individual contests or those with lower skills (Boersen and Phillips, 2006; Fisher and Cox, 2006). The same approach, but presented in a more positive way was applied in the Computer Science Olympiad for High School Students organised by Northwest Missouri State University. The penalty was replaced for bonus for each minute till the end of the contest (Myers and Null, 1986).

There was conducted research on IOI submissions which showed that vast majority of submissions fail on at least one test (Kemkes et al., 2006). The same holds for LitIO. Under all-or-nothing scoring scheme in the on-line exam session of LitIO finals in 2010, only 14 out of 249 (5.6%) would have scored points. The differences in grading schemes between LitIO and IOI in our opinion would not have changed the scores significantly.

Under current LitIO format (individual contest, two-to four tasks of different difficulty, average level of big part of contestants) such scoring scheme would not be appropriate. The majority of contestants would get zero scores. That would decrease their motivation because they would feel that their efforts were rejected as their skills are not that low.

2.12.4 All-or-Nothing Batch Scoring

After it has been acknowledged (Kemkes et al., 2006; Verhoeff, 2006) that partial scoring scheme included large component that can be viewed as a noise, in IOI tests were replaced by test cases. Each test case consists of a set of test-runs, and each test case is intended to assess a well-defined characteristic of a submitted algorithm-code complex. This characteristic can be related to correctness and/or efficiency.

If each test run has a binary (*pass* or *fail*) outcome, then a test case is passed if and only if all test runs are passed. The total score for the task is the sum of separate scores for test cases. In the thesis we will call this scoring *all-or-nothing batch scoring.* It can be expressed with the following formula:

$$S = \sum_{i=1}^{g_c} w_i \prod_{j=1}^{g_{c_i}} t_j \tag{2.3}$$

Where g_c is the total number of test cases, w_i $(i = 1 \cdots g_c)$ are weights for each test case, and g_{c_i} is the number of test runs in test case c_i $(i = 1, \cdots g_c)$. The weights are such that $\sum_{i=1}^{g_c} w_i = P$ where P is the maximum amount of points for the task.

All-or-nothing batch scoring scheme is expected to reduce arbitrariness in scores. The scheme is considered to be more positive approach than all-or-nothing scoring (Revilla et al., 2008). Nevertheless the scoring scheme is much more rigorous to the contestants than partial scoring. This was acknowledged and there appeared suggestions to introduce more real time feedback together with this scoring so that the contestants could detect and correct trivial mistakes (Forišek, 2006; Kemkes et al., 2006; Opmanis, 2006).

The main concern which prevents implementing all-or-nothing batch scoring at full pace in LitIO was discussed in Subsection 2.10.2, i.e. the difficulty in defining the disjoint correspondence between the domain of input and the groups of possible solutions. Similar concerns were expressed at (Helmick, 2007) who emphasises that functional testing produces Boolean answer and an important goal in automated testing is to segment the tests such that pieces of functionality are isolated and tested as independently as possible.

2.12.5 Other Black-Box Scoring Possibilities

We did not found enough material in the published papers on informatics contests, that would reveal if there is a variety of scoring schemes in informatics contests. Different scoring typically involved different contest format (Cormack, 2006). Here we list several interesting scoring ideas that we came across.

- *Scoring related to real-time feedback.* Real-time feedback is considered as a mechanism which reduces competitive pressure and allows the contestants easier track minor errors (or more serious errors depending upon the amount of feedback). Different forms of feedback were analysed in (Cormack et al., 2006). Feedback can be integrated into scoring scheme. For example, full feedback can be provided on correctness tests and a requirement might be introduced that only the solutions that pass all the correctness tests are tested with the efficiency tests.

- *Graduated difficulty (multi-part tasks).* Assigning scores to partially correct (i.e. incorrect) solutions is one of the most questionable issues in informatics contests. Multi-part problems are problems where a task is formulated as a sequence of sub-tasks. Each subsequent sub-task is a straightforward derivative of the previous one. We suggest that for some tasks that can be properly decomposed into several sub-tasks even all-or-nothing scoring can be applied to each sub-task. We believe that such an approach could be best applied if, it is possible to construct a solution to each sub-task by augmenting the solution of the previous sub-task. If there is no such possibility, then some contestants might be trapped on a decision: whether to risk to implement a complicated solution to the whole task or choose a simple approach that will yield some points, but will not lead to the solution of the whole task. Some experience of applying sub-tasks in informatics contests was presented in (van der Wegt, 2009).

- *Speed of execution as an element of the score.* The current LitIO and IOI practice does not include the speed of execution into the scoring scheme. The run-time limit is fixed in advance and the test is considered to be passed within the run-time limits as long as the algorithm-code complex does not exceed the time limit while executing the test. The speed of execution is not considered a good metric because it may often depend also on the compiler/programming language implementation, especially in the IOI where run-time limits are very tight. However we found an example of including the speed of execution into the scoring scheme in (Diks et al., 2007). If the submitted program fluctuates over the time limit, then the result may vary in subsequent evaluations. To solve this, the function which maps the running time to points is made continuous. The function is flat from zero to half the time limit. Then it linearly descends to zero at the point of time limit, thus avoiding sharp changes in the number of points (Fig. 2.12).

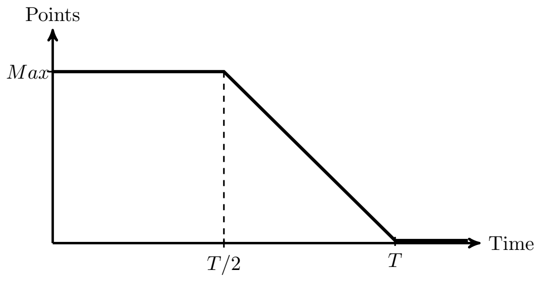

Figure 2.12: Function mapping run-time to points T represents time limit for a test and Max represents maximum number of points for the test.

- *Combined scoring.* This scoring scheme was proposed in (Kemkes et al., 2006). It combines *all-or-nothing-batch scoring* together with *a significant progress scoring scheme.* Significant progress scoring is defined as scoring which assigns full points to a test case, if at least one test run from this test case was solved correctly. The combined scoring scheme uses two pieces of information on each test case: whether the algorithm-code complex has passed at least one test run and whether it has passed all test runs. This scoring scheme was developed during the research and we have not come across its further development or application.

2.13 Overview of the Experience of Semi-Automated and Manual Evaluation

This section covers the experience of semi-automated and manual evaluation of submissions and programming assignments. The semi-automated evaluation here is associated with the manual evaluation, because conceptually semi-automated evaluation is closer to the manual evaluation than to automated, in this case. Such an evaluation involves an extended concept of the submission (not limited to the source code only) and focuses on the manual evaluation of algorithmic ideas and features of the program design. Black-box testing is not eliminated from the evaluation scheme.

Historically, a submission had to include an informal block-diagram description and argumentation of the algorithm in IOI. However, the evaluation of these descriptions turned out to be too labour intensive and was dropped (Verhoeff, 2006).

We were looking for papers that describe the experience of applying the manual and semi-automated evaluation in informatics contests and we found very few references (Pankov and Okruskulov, 2007; Pohl, 2004). The most comprehensive reference of manual evaluation was in (Pohl, 2008) which presented the experience of applying the manual evaluation in the BWINF [1] (Pohl, 2007).

The manual evaluation focuses on a qualitative assessment and has been successfully applied in BWINF for some years. A written description of solution is manually evaluated in BWINF. If needed, the source code is also analysed. That explicitly allows evaluate the problem solving part. We identified two essential differences from black-box scoring approach.

One difference is that a submission is not restricted to the source code only in order to provide more materials for evaluation. In the case of BWINF, the necessary part of a submission is a written description of the solution approach. The description should cover not only the algorithmic solution (which is the case in LitIO), but also the implementation approach of the solution described. Typically this is presented in a form of a short description of program components. It should be stressed that the success of the contestant also heavily relies on his/her ability to explain algorithmic and implementation solutions in the natural language.

The source code can also be inspected by the jury if other parts of a submission leave doubts on how to grade, or if they want to understand the reason for the mistake. This is different from the current LitIO situation where the evaluation of a verbal algorithm description is not related to the source code and the jury is not allowed to look into the source code while evaluating the verbal algorithm descriptions.

The contestants are also asked to design their own examples or test cases in order to demonstrate the functionality of their program. However, we found no data how this is graded in BWINF. A similar approach with grading suggestions was found in (Cormack et al., 2006). Evaluation criteria might include the test validity, exposition of special cases, and detection of errors. The submitted tests might be

[1] *Bundeswettbewerb Informatik* (BWINF) is a *German National Computer Science Contest* (Bun, 2010).

run either against the programs composed by the jury or against the submissions of other contestants. The suggestion was that the score should be a fraction of all incorrect algorithm-code complexes detected by a set of test cases submitted by a contestant.

Another significant difference from the black-box grading approach is that an evaluation scheme (a set of evaluation criteria) has to be developed for each task and refined after the jury examines some selected submissions. The reason for refining the evaluation scheme is that real submissions may contain unexpected solution approaches or unexpected flaws. This is very different from the black-box evaluation where the grading scheme is designed and fixed before the start of evaluation.

Refinement of the grading scheme, after black-box testing has been started, is not considered appropriate because that would tailor the grading scheme to those few submissions examined. For example, it would be tailored to catch unexpected heuristics discovered in those few submissions, but would not catch unexpected flaws in other submissions that were not examined.

The requirement to create explicit evaluation criteria forces the jury to express their reasoning about the problem in written form and allows a feedback for the contestants.

A peculiarity of the BWINF evaluation scheme is that it is negative, i.e. it is oriented towards discovering weaknesses of submissions. The evaluation is performed over a weekend when all the jury members meet and each submission is evaluated by two jury members. However the total score can't become negative.

Discussions on returning the manual evaluation to informatics contests were started by (Verhoeff, 2006). One of the proposals was to introduce a motivated and prepared evaluation scheme supported by measurements.

Most of the recent publications related with evaluation in informatics contests, discus the evaluation in international contests in particular IOI. However multilingualism of the contestants becomes an issue and has to be analysed separately. The experience of other international science contests, for example IMO, might be useful while considering this issue (Verhoeff, 2002).

We also reviewed the experience of manual and semi-automated evaluation in programming courses. However most of it does not correlate with the informatics contests. We will show that this time the concerns, relevant to evaluation in the programming courses, do not have such a relevance in the informatics contests.

Having multiple human graders in the mass programming courses is common and it is not easy to achieve a consistent grading among multiple graders. Therefore that becomes a topic for investigation, and semi-automated evaluation systems are supplemented with additional features to facilitate this issue (Ahoniemi and Reinikainen, 2006; Daniels et al., 2005; Spacco et al., 2005).

In LitIO, the consistency of multiple human graders is achieved by determining common evaluation criteria and evaluating each submission by at least two different graders. Each submission where the scores between graders differ by more than a fixed amount of points, is revised and discussed separately. Therefore this is not considered as an issue in LitIO. The difference emerges because the informatics contest is an event (LitIO is the event of the year) and therefore more human and time resources are available for evaluation. Such an extensive evaluation as in LitIO would require too much human resources if applied in evaluation of regular programming assignments in the programming courses.

Black-box grading systems do not provide enough feedback about the origin of an error. To assist this, semi-automated grading tools might be preferred to fully automated evaluation (Ahoniemi and Reinikainen, 2006). In such cases, human graders have a possibility to add their comments, remarks and other suggestions for the students. Such a feedback is less important in informatics contests. Even if some feedback of human graders is provided (Pohl, 2008), it is not considered as important in informatics contests because of a limited role of contests as an educational event.

2.14 Conclusions

Informatics contests are individual contests, where the contestants have to design and implement an algorithm in order to solve the given task during short term exam sessions. Informatics contests form the environment for research in the computing science education. The evaluation of algorithms implemented as programs develop an educational situation where many submissions have to be evaluated in a short period of time. The goals of contests imply that both problems solving and problem development skills have to be taken into account.

The (automated) evaluation of programming assignments is not a new area in computing education research. Much research on the evaluation of programming assignments during the programming courses has been conducted and published. However, the research and development of automated evaluation programming assignments are moving into different direction than the informatics contests. An exception is automated evaluation of programming style. Many tools were created for this purpose and much research has been pursued on applying such tools to the programming courses. That this experience might be transferred to informatics contests. However, it requires a separate study.

Informatics contests are not part of any successive learning process, therefore many aspects relevant in the learning process are not valid in the informatics contests, so in the evaluation we concentrate on the quality of the submission. Quality is understood either as a conformance to specifications or as the ability to satisfy the needs of the user. In the case of informatics contests both the designers of specifications and users are the same, i.e. the jury (scientific committee).

There are two major approaches to evaluating programming assignments: static and dynamic. From the point of view of automation, evaluation can be manual, semi-automated and automated. Evaluation schemes in the informatics contests can be categorised to those which are limited to black-box testing (dynamic automated evaluation) and those which foresee another type of evaluation. In LitIO, the static and dynamic, automated and semi-automated evaluations are applied. The verbal algorithm description, testing to specifications and efficiency as well as the programming style are included into the scoring scheme.

The dominant use of black-box testing attracts most criticism and attention. Note that many concerns are not related to the black-box testing itself, which is a natural part of software development, but to the conclusions presented in the form of scores. The concerns can be divided into two categories. The first category consists of these concerns that are not directly related to black-box testing, but to its dominant use, i.e. the absence of other forms of evaluation, for example, incapability to identify interesting and original problem solving approaches. The only way to solve these problems is to introduce other forms of evaluation. The second category consists of the concerns that are expected to be solved by black-box testing an a satisfactory level. For example, it is expected that black-box testing might identify incorrect solutions on a satisfactory level.

Scoring schemes for black-box evaluation differ from all-or-nothing scoring, where a submission is either accepted (or not) to partial scoring where points are assigned for each successfully passed test. There is a tendency to move towards a more qualitative evaluation.

As for other than black-box scoring, we found very little published experiences. Such scoring involves an extended concept of submission (not restricted to the source code only) and focuses on the manual evaluation of algorithmic ideas and features of program design. Black-box testing is not eliminated from the scoring scheme.

The analysis has showed that there are concerns whether black-box testing achieves the goals on a satisfactory level. The concerns were based on the presentation of separate cases except for one publication containing an extensive report. This induced the goal to investigate how much black-box testing corresponds the expectations of the evaluators in LitIO.

The ultimate goal of this dissertation is to come up with an improved scoring scheme that would have a motivated list of criteria. Each submission is evaluated against each criterion and the obtained results are aggregated to get the final score. However, the evaluation has to be performed taking into account the available human and time resources. Moreover, there are different views among the jury towards various aspects of evaluation. Therefore we suggest to attribute this problem to the category of multiple criteria decision problems and the scoring scheme can be developed by applying multiple criteria decision techniques and algorithms.

3 Overview of the MCDA Process and Methods

3.1 Concept of MCDA

The field of multiple criteria decision analysis (MCDA) is also termed as a multiple criteria decision aid or multiple criteria decision making (MCDM). Its target is to help reach a consensus and compromises between conflicting goals (i.e., multiple criteria) in complex problems.

In real life it is unusual that the problem is presented to the analyst in a form of a clearly defined set of alternatives and criteria (Belton and Stewart, 2003). Problems might be complex and confusing and they typically involve a wide range of criteria that need to be considered. They might involve conflicting criteria, the conflicts between different stakeholders about the importance of criteria in making a decision. It might even be required to define criteria as they are not clear at the initial stage of the problem. The general goal of MCDA is to assist individual or groups of decision makers to choose the best alternative. Potential problems that MCDA can be applied come from a variety of areas like business, medicine, public policies or education.

MCDA is defined as a collection of formal approaches which seek to take into account multiple criteria in order to help decision makers to explore different decision alternatives (Belton and Stewart, 2003).

Even though mathematical MCDA algorithms help to arrive at some acceptable alternative, many authors emphasize that MCDA cannot be used to arrive at the "right" answer and it cannot provide a fully objective analysis and totally eliminate subjectivity (Belton and Stewart, 2003). The process of MCDA is emphasised more than the decision it helps to arrive at (Keeney and Raiffa, 1976; Roy, 1996; Zeleny, 1982). The process involves not only the application of mathematical algorithms to come up to the final decision, but also learning about the problem, identifying the key concerns, priorities, uncertainties, values, exploring and generating different alternatives. This should lead to better explainable and justifiable decisions.

3.2 Main Concepts

We did not find a unique understanding of MCDA concepts and terminology. Therefore we presented our own definitions which were developed based on (Val, 2002; Li and Yang, 2004; Triantaphyllou, 2000). In this subsection we also recall a few other relevant concepts.

Alternatives. Different choices available to the decision maker. In the case of evaluation in the LitIO problem, the set of alternatives consists of all the submissions designed to solve a particular task in an exam session of the informatics contest. We assume that the set of alternatives is finite.

Attribute. A statement of something that is desired to be achieved. Attributes represent the different dimensions from which the alternatives can be viewed. Attribute specification does not require a measure specification. It is possible that attributes are arranged in a hierarchical manner.

Criteria. Each attribute of an MCDA problem is measured in terms of one or more criteria. The same criteria may be used for measuring different attributes. Different criteria might be associated with different units of measure.

Decision weights. Weights of importance assigned by decision makers to each criterion.

Measurement The assignment of numbers to objects or events in a systematic order.

Scale. A rule by which the measurement is performed (Stevens, 1946).

Interval scale. A measurement scale where one unit on the scale represents the same magnitude across the whole range of the scale.

Ratio scale. An interval scale in which *zero* represents the absence of a thing being measured.

3.3 Evaluation in LitIO as an MCDA Problem

In LitIO the contestants get algorithmic tasks and have to design and implement an algorithm in one of the allowed programming languages. A submission consisting of an algorithm-code complex and a verbal algorithm description is submitted for evaluation.

At present three submission attributes are identified. They are: the quality of verbal algorithm description (it includes both the quality of description and characteristics of the algorithm described), algorithm-code complex performance with correctness and efficiency tests, and the quality of programming style. The algorithm-code complex performance is evaluated using black-box testing. Other attributes are evaluated semi-automatically (manually). Partial scores for each criterion are added up to get the total score. After the contest, the ranking based on the total scores is derived.

The evaluation scheme is based on the practice of other similar contests, LitIO traditions, and the school of teaching algorithmics in Lithuania (Dagienė and Skūpienė, 2007). However, the scheme was neither analysed, nor supported by scientific methods. In the last few years there appeared concerns whether such an evaluation practice, especially assigning the scores based on the program performance, both in LitIO and in other similar contests, corresponds to the goals of informatics contests (Forišek, 2006).

Thus the issue about scientific motivation of the evaluation scheme can be identified as an MCDA problem. Submissions play the role of alternatives, evaluation criteria correspond to the MCDA concept of criteria. The criteria are conflicting. For example, how to compare a faultless algorithm-code complex which implements a correct inefficient algorithm with an algorithm-code complex which implements an efficient algorithm, but contains implementation mistakes. There are many other decision context issues that have to be investigated and taken into account. Here are some of them:

- Review the goals of the contest.
- Investigation to which extent the automated performance testing results correspond the expectations of the task designers.
- Investigation whether the metrics used for measuring various characteristics of software are applicable as the evaluation criteria (submitted algorithm-code complex is also a piece of software).
- Reconsideration of the concept of submission.
- Reconsideration of the set of evaluation attributes and criteria.
- Reconsideration of the score aggregation function.
- etc.

The answers to some of the questions raised here have already been obtained during this research and presented in the previous chapters.

It can be concluded from the considerations presented above that evaluation in the LitIO problem is a complex MCDA problem, and therefore it can be solved using MCDA techniques and algorithms.

Further we will use the term *evaluation in the LitIO problem* as a formal term. By this term we understand an MCDA problem the goal of which is to investigate the background and problems of evaluation in LitIO and other informatics contests, to construct the concept of submission, and to propose the evaluation scheme for use in LitIO.

3.4 Roles in MCDA

Three major roles can be identified in the decision analysis. They are: *decision maker*, *decision analyst* and *stakeholder* (Val, 2002). A decision maker has the power to make decisions and typically is responsible for the consequences of this decision. A decision analyst analyses the problem, generates and suggests alternatives and facilitates decision making. A stakeholder is a person or a body with an interest in the decision under consideration. The roles can overlap and there exist different relationship models among the three roles.

We will add one more role in this study, i.e., the role of *expert*. By an *expert* we assume a person who has the authority and experience in the area of the problem under consideration. This is a very general definition, while concrete requirements for someone to be considered as an expert will be made taking into account the peculiarities of evaluation in the LitIO problem. The main mission of an expert is to provide valuable insights about the problem. In terms of the three main roles, the experts are decision analysts. However, we would like to maintain the term *expert* in order to emphasise the knowledge and authority in the area versus the responsibility to perform problem analysis.

Having presented possible roles in the MCDA process, we will look at the roles of evaluation in the LitIO problem.

The scientific part of LitIO is managed by the scientific committee. The scientific committee also performs the role of jury. Therefore these two terms are used as synonyms in LitIO. The scientific committee is responsible for all the scientific decisions, i.e., approving the syllabus of the contest, designing tasks and tests, approving the evaluation procedure, performing evaluation, approving ranking and declaring winners. In 2010, the scientific committee of LitIO consisted of 13 members (Sci, 2010). The scientific committee is the only decision maker in this context.

The role of a decision analyst is played by the author of this thesis.

The most important stakeholders are interested in programming and algorithmics students at secondary education from all over Lithuania, as well as the community of informatics teachers. The community of stakeholders is affected directly by each decision or change in the evaluation scheme. The scientific committee of LitIO is also a stakeholder, because possible changes in the evaluation scheme might change their working procedures, time spent on task design and evaluation.

Other stakeholders include the Ministry of Education and Science of the Republic of Lithuania (providing financial support to LitIO), the Lithuanian Youth and Technical Creativity Palace (having the responsibilities for organising LitIO). Even universities in Lithuania are stakeholders, because they expect the scientific quality in LitIO and grant the winners of LitIO extra points when entering computer science studies in the universities.

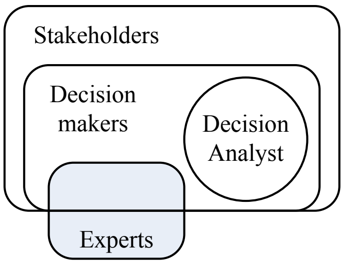

Figure 3.1: Model of relationship among different roles in decision analysis of evaluation in the LitIO problem

By an expert we define a person having the background in informatics and at least five-year experience of work in informatics contests either as a member of the scientific committee or as the jury member. More details about the experts that took part in this research will be provided in Subsection 6.1.1, where the background and the details of involvement of the experts are described. Here we only want to provide the relationship structure between the different roles. Some of the experts involved in this research belonged to the Scientific committee of LitIO. We invited those experts deliberately, because members of the Scientific committee of LitIO know the peculiarities and problems of carrying out evaluation in LitIO best. The other experts were invited from outside, i.e. they had some experience in national informatics contests of other countries as well as in the regional and international informatics contests.

A model of relationship among the different roles in the decision analysis process in evaluation in the LitIO problem, is presented in Fig. 3.1.

3.5 Classification of MCDA Problems

Four broad categories of MCDA problems have been proposed (Roy, 1996):

- *The choice problematique.* Problems fall into this category if there is a need to make a choice from a set of alternatives. However the set of alternatives might be either finite or infinite.

- *The sorting problematique.* In this case the given alternatives have to be sorted into several categories, such as “definitely acceptable”, “possibly acceptable”, “definitely unacceptable”.

- *The ranking problematique.* The alternatives have to be ranked in some order of preference.

- *The description problematique.* Possible alternatives and their consequences have to be described formally in a systematic way so that the decision makers could evaluate the alternatives.

Variations or amendments to this classification are also possible (Belton and Stewart, 2003).

Another classification of MCDA problems is *one-off* versus *repeated* problems. In some cases, a decision has to be made only once as the problem is unique. This is a one-off problem and the process is oriented towards arriving at a specific decision. In the case of repeated problems the same problem is recurring a few times or periodically. Then MCDA is oriented towards creating a procedure to be used in decision making.

An MCDA problem can also be classified either as *a single decision making* or *group decision making* problem. In the case of a group decision making problem, several decision makers are involved and they can have different values and opinions how to address the problem. In order to approve the decision, the consensus and compromise among different decision makers has to be reached.

According to the classifications presented above, evaluation in the LitIO problem is *the ranking problematique* as the final outcome of evaluation procedure is a ranked list of contestants based on which the awards will be distributed. Based on the second type of categorisation, the evaluation problem is a *repeated* problem, therefore the focus of the research is on refining the evaluation scheme which could be applied annually in LitIO. It is a *group decision making* problem, because the role of a decision maker is played by the members of the LitIO Scientific committee and in order to approve the proposed evaluation scheme, the consensus among the decision makers is necessary.

3.6 Stages of MCDA

Different authors suggest different stages of the MCDA process. (Val, 2002) proposes a scheme consisting of four stages in particular, problem structuring (decomposed into five sub-stages), preference elicitation, recommended decision, and sensitivity analysis. (Oberti, 2004) suggests four stages of the MCDA process, i.e., beginning of the study, evaluation of actions, multiple criteria modelling, multiple criteria processing, and recommendations.

Each stage consists of two or three sub-stages. (Belton and Stewart, 2003) offer three stages: problem identification and structuring, model building, and using a model to inform and challenge thinking. The scheme based on (Belton and Stewart, 2003) is presented in Fig. 3.2.

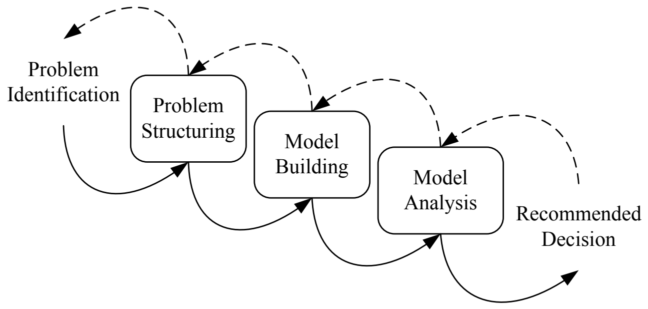

Figure 3.2: Basic stages of the MCDA process

These stages reflect a variety of approaches to MCDA, however, they confirm that an extensive problem analysis and structuring are vital before mathematical algorithms can be applied. In all those approaches the stages are iterative and interactive, i.e., they foresee a return to previous stage, review and update its outcome.

Next we would like to comment on a further structure of the thesis. In the subsequent sections we will overview the theoretical background of problem structuring, model building, and sensitivity analysis.

Without actually performing problem structuring (unambiguously describing criteria and alternatives) it is not reasonable to search for the most suitable MCDA algorithm. Their suitability to solve the problem depends upon the type, structure and interrelationship of criteria, and other requirements. Therefore, in this study we have performed problem structuring (presented in Chapter 6) before analysing MCDA approaches of model building (presented in Section 3.8). We assume that the reader will become familiar with the outcome of problem structuring before reading about model building.

Note that different sources use rather different MCDA terms, and the terms we are using in the thesis might have different meanings elsewhere.

3.7 Problem Structuring

3.7.1 General Overview

As indicated above, different authors suggest different stages of MCDA, however, in all the approaches the first big step is a deep analysis of the problem. Its goal is to initiate thinking about the problem, to capture its complexity and identify the key issues, external environment, constraints, goals, values, uncertainties, alternatives, and stakeholders. We will use the term *problem structuring* to refer to this stage and assume that, at the end of this stage, not only a complex analysis has been made, but, in particular, the problem category, the roles, objectives, alternatives, criteria, i.e., the parts required in order to proceed with the next step of decision analysis process have been identified.

It is important to understand that problem structuring does not only seek to modell the existing reality, but also is a constructive one, i.e., tries to abstract the reality. There is a difference between modelling real life objects and modelling MCDA problems. Once the real life objects are modelled, it is possible to try to test how close the model approximates to the reality. However, there is no easy way to check the validity of a model which encompasses values (Belton and Stewart, 2003)

There are *problem oriented* and *value oriented* approaches to problem structuring. In the problem oriented approach, the problem is identified at first and the goals are figured out afterwards. In the value oriented approach, values and goals are discussed first, and only then one should start looking for decision opportunities (Val, 2002).

Many sources stress the importance of this process, but a huge variety of problem areas and complexities make it difficult to construct a good for all cases method how to perform problem structuring. Apparently, a decomposition of the problem structuring process becomes easier when we consider the structuring process of a more specific problem. An example of the decomposition can be found in Fig. 3.3 (Val, 2002).

If we look at the decomposition from the point of view of evaluation in LitIO, part of the job of defining the decision context already has been performed by presenting an extensive analysis of the context of evaluation of algorithm-code complexes. The remaining part, where the current LitIO evaluation scheme will be analysed from the point of view of existing software quality standards, will be presented in the subsequent chapter.

In general, generating alternatives might be quite a complicated issue. However, in our case, the set of alternatives is clear, i.e., submissions that have to be ranked based on the evaluation results. However, the concept of submission needs to be reviewed. The main task of problem structuring still to be performed is identifying the attributes, creating a hierarchical model of attributes, and specifying the criteria.

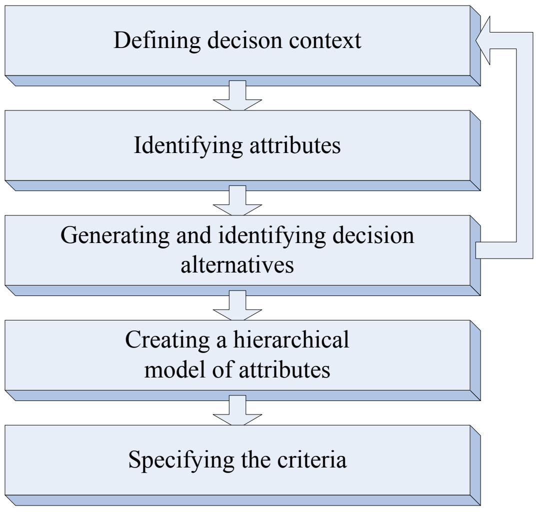
Defining decison context
Identifying attributes
Generating and identifying decision alternatives
Creating a hierarchical model of attributes
Specifying the criteria

Figure 3.3: Decomposition of the problem structuring process (based on (Val, 2002))

Problem structuring can be performed either in an informal or in a formal way. There exist many general managerial tools, including software, which can support performing problem structuring in a more formal and systematic way. Examples of such tools can be SWOT (Strengths, Weaknesses, Opportunities, Threats) (Swo, 2007, 2010; Hill and Westbrook, 1998), SODA (Strategic Options Development and Analysis) (Eden and Simpson, 1989), JOURNEY (Eden and Ackermann, 1998), CATWOE (Customers, Actors, Transformation, Worldview, Owners, Environment), CAUSE (Criteria, Alternatives, Uncertainties, Stakeholders, Environmental factors), and Cognitive mapping (Eden, 1988). Many tools of this kind are described in (Keeney and Raiffa, 1976; Keeney, 1992). A lot of those methods are suitable for or suggest working with experts that need to be interviewed or involved into the discussion as part of idea generation.

The choice of a particular tool depends upon the type of the problem and other various aspects of the problem and the environment. Some methods help in defining a decision context. Some are intended to assist in structuring already generated ideas. Some methods are more suitable when the experts are discussing the problem together, others are more suitable when the experts are interviewed separately or when the experts are located remotely, but need to reach a consensus, and so on.

The methods discussed above are very general, managerial. We went through many such methods (including those mentioned above) and we did not feel that they would completely suit the context of the problem. Different methods are different ways helping to arrive at the same goal. Therefore we decided that the decision maker makes the decision as to which method should be chosen.

By performing the search for the best approach, we came across the GQM (Goal/ Question/ Metric) approach, the basic idea of which is defining and interpreting measurable software.

3.7.2 GQM (Goal/Question/Metric) Approach

GQM framework was the outcome of works of V. Basili and D. Weis, and was originally designed for evaluating defects of several projects (Basili and Weiss, 1984). One of the practical applications that GQM is associated with is working towards the improvement of software.

Formally GQM can be defined in the following way: *"GQM presents a systematic approach for integrating goals to models of the software processes, products and quality perspectives of interest based upon the specific needs of the project and the organization"* (Basili et al., 1994). The main idea of the GQM framework is that measurement should be goal-oriented. It starts from defining a goal, then transforms the goal into a set of questions and finally defines metrics that present answers to the questions (Fig. 3.4), i.e., it derives software measures from measurement questions and goals (van Solingen and Berghout, 1999). This is especially useful in cases where it is difficult to decide what to measure in order to achieve the goals.

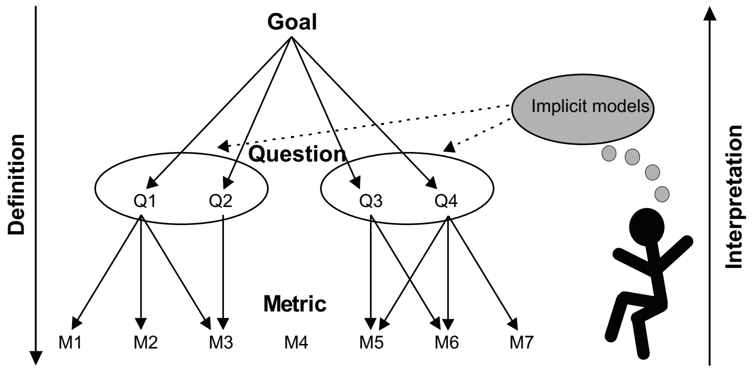

Figure 3.4: The GQM paradigm Metrics are defined in a top-bottom way and they are interpreted in a bottom-up way (van Solingen and Berghout, 1999).

The obtained measurement model consists of three levels (Basili et al., 1994):

- *Conceptual level (Goal).* The goal is defined for an object from different points of view taking into account different quality models. It specifies the purpose of measurement. Products, processes or resources can be the objects of measurement.

- *Operational level (Question).* The achievement of each goal is characterised by a number of questions with respect to a selected quality issue.

- *Quantitative level (Metric).* Each question is associated with one or more metrics in order to provide the answer to the question in a quantitative way. The metrics can be either objective or subjective. Objective metrics depend upon the object being measured, but not upon the viewpoint. Subjective metrics depend both upon the object being measured and upon the viewpoint.

The same metrics can be associated with different questions under the same goal. The same metric might have different values if taken from different viewpoints. The final step of the GQM process is to define the data collection, validation and analysis mechanism. These final steps can be associated with the final steps of MCDA performed after problem structuring has been completed.

We suggest that the GQM framework is suitable to perform the required part of problem structuring, defined in the previous subsection. It foresees a hierarchical structure and a systematic way of eliciting metrics (criteria). The advantage of this framework is that it was originally created for deriving software metrics, and a submission can be treated as software.

3.8 Model Building

3.8.1 Requirements for the Model

We have already introduced *evaluation in the LitIO problem* as an MCDA problem in Section 3.3 and indicated that the goal is to come up with *an evaluation scheme* for use in LitIO. After problem structuring, the next phase is model building, i.e., the choice of an MCDA model. Before analysing various possibilities, we have to define the requirements for the model.

The decision context of our problem is rather specific. The problem belongs to the ranking problematique category and is a group decision making problem. Moreover, the chosen method will be applied in an educational informatics contest situation. Therefore it is highly important for the approach to be accepted by the community of informatics contests. (Belton and Stewart, 2003) emphasize that the ability to explain the chosen approach to a variety of backgrounds is an important factor in choosing the MCDA approach. It should be noted that our problem is a repeated problem. It means that the process of ranking submissions will have to be repeated each time the informatics contest takes place. This strengthens the importance of method acceptability and understandability by the stakeholders.

The evaluation scheme should be motivating to the contestants and one of important motivating factors is its understandability. Therefore in search of the best model we will give the priority, to the models which foresee simpler scoring functions.

Note that the evaluation scheme consists of parts which are revealed to the contestants, but it also contains the parts hidden from the contestants. For example, the scores assigned by individual jury members during manual evaluation are not revealed to the contestants, only the aggregated score is. We emphasise that the parts of the evaluation scheme which are revealed to the contestants must be easily understandable and transparent.

Even though the problem is described as the ranking problematique, it should be noted that it is not enough to present ranking to the contestants. The values, based on which the ranking was made, also are to be presented. After the contest, the contestants are interested not only in their position in the ranking table, but also in the closeness of their scores to that of their rivals, and to the scores of the winning positions.

It is commonly accepted in informatics contests that a score aggregation function mapping the performances for separate criteria into real numbers is defined and announced to the contestants in advance. From what we managed to find in the publications, the ACM-ICPC type contest is the only contest where the value function is mapped to two-variable output (Subsection 2.12.3). However it is still a sore aggregation function that unambiguously induces ranking. Therefore we will focus on the MCDA approaches which foresee defining the score aggregation function, inducing the ratio scale, and the ranking is made after the values of the function for each alternative have been calculated.

3.8.2 Single Decision Maker Problem

Let $A = \{A_1, A_2, \cdots A_n\}$ be a finite set of alternatives and $C = \{C_1, C_2, \cdots C_m\}$ be a finite set of criteria. Let x_{ij} be the performance of alternative A_i $(i = 1, 2, \cdots, n)$ in terms of criteria C_j $(j = 1, 2, \cdots, m)$. Suppose $w_j \geq 0$ is a relative weight of the criterion C_j $(j = 1, 2, \cdots, m)$ and $\sum_{j=1}^{m} w_j = 1$.

Then a single decision maker MCDA problem can be expressed by the following decision matrix:

$$\begin{array}{ccccc} & C_1 & C_2 & \cdots & C_m \\ & w_1 & w_2 & \cdots & w_m \\ A_1 & x_{11} & x_{12} & \cdots & x_{1m} \\ A_2 & x_{21} & x_{22} & \cdots & x_{2m} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ A_n & x_{n1} & x_{n2} & \cdots & x_{nm} \end{array} \tag{3.1}$$

Note that the decision matrix does not indicate the category of the MCDA problem. The same decision matrix can be used to describe both the choice problematique and the ranking problematique MCDA problems.

We presented the definition of a single decision making problem, because classical MCDA methods assume a single decision maker problem, and the existing single decision making techniques are extended to solve group decision problems.

3.8.3 Choice of the MCDA Approach

From the requirements to the LitIO evaluation scheme, presented at in the beginning of this section, it follows that we have to focus on MCDA approaches which foresee defining the score aggregation function and the ranking is made after the values of the function for each alternative have been calculated.

After the problem structuring stage has been completed, a model representing preferences of a decision maker has to be constructed. Such a model should contain two primary components (Belton and Stewart, 2003):

- *Preferences in terms of each individual criterion.*
- *An aggregation model, i.e., s model that would combine preferences across criteria.*

Once unambiguous quantitative metrics at the lowest level have been defined, the first model of the component is not necessarily trivial. In general, the strength of preference of one alternative over another should be also taken into account. Some thresholds might be introduced.

If the attributes and metrics are arranged in a hierarchical way, then it is important that the lowest level of the tree (metrics) induce unambiguous ordering in terms of each criterion. Aggregation can be applied either in a single operation to all the criteria or at each tree level separately. Theoretically the formulas describing both approaches are algebraically equivalent (Belton and Stewart, 2003). This is one of the reasons why many MCDA methods assume the criteria aggregation in a single step.

Many different MCDA approaches are presented and categorised in (Belton and Stewart, 2003; Carlsson and Fullér, 1996; Chen et al., 1992; Kahraman, 2008; Triantaphyllou, 2000). Instead of focusing on separate MCDA methods, we will first look at the major families of MCDA methods. (Belton and Stewart, 2003) distinguish three major families of MCDA approaches:

- *Value measurement theory* (Keeney and Raiffa, 1976). The main idea of this approach is to construct a value function that would associate each alternative with a real number in order to produce ranking of alternatives. The main idea of this theory correspond to the intentions and reasoning presented at the beginning of this subsection. Therefore we will include it for a further consideration.
- *Satisficing (or Goal programming)* (Simon, 1976). This approach instead of creating one value function operates on partial value functions. By a partial value function we understand a value function that maps the performance of alternatives in terms of a certain criterion to a real number. The main idea of the approach is that the most important criterion is identified and the acceptable level of it is determined. Then the alternatives are eliminated until all the remaining alternatives achieve the acceptable level. At this point, the second most important alternative together with its satisfactory level is identified. The alternatives which do not reach the satisfactory level of the second criterion are eliminated again.

  This approach is not suitable for our problem as it does not assume the score aggregation at all.
- *Outranking* (Roy, 1996). Outranking methods also operate with partial value functions and involve pairwise comparisons of alternatives. An alternative is dominated by another alternative if another alternative performs better in

terms of one or more criteria and equals in the remaining criteria. The concept of *outranking* is introduced. The outranking relationship of two alternatives describes that even though the two alternatives do not dominate each other mathematically, the decision maker accepts the risk of regarding one alternative almost surely better than the other.

We consider this approach also unacceptable in our situation because it again deals with preferences in terms of separate criteria and does not foresee score aggregation using a single value function. The concept of outranking, i.e., allowing the decision maker to take the risk of considering one alternative better than the other is not acceptable in a contest community where scoring is an extremely sensitive issue.

Besides the main families of MCDA approaches, a fuzzy logic is often considered to be applied to MCDA problems. The fuzzy logic is used in group decision making, however, the fuzzy logic is not a separate methodology, but a tool that can be applied within other MCDA approaches including the ones described above. Therefore we assume that the fuzzy logic might be applicable in the case of this problem and we will look at the concepts of fuzzy logic as well.

At the same time we must be aware about the MCDA paradox which asks *what decision-making method should be used to choose the best decision-making method* (Triantaphyllou, 2000). This paradox reveals the roots of difficulty of comparing different MCDA methods in search of the best one.

3.8.4 Value Measurement Theory

This theory was mainly started by Keeney and Raiffa (Keeney and Raiffa, 1976). More on it can be found in (French, 1988; Roberts, 1979).

The main idea of this theory is that a real number ("value") is associated with each alternative in order to produce ranking of alternatives. *The value function* is defined as a function that assigns a non-negative number to each alternative, indicating the desirability (or preference) of the alternative.

The value function has to satisfy the following requirements: an alternative A_{i1} is preferred to the alternative A_{i2} ($A_{i1} \succ A_{i2}$) if and only if $V(A_{i1}) > V(A_{i2})$; the alternatives are indifferent if and only if $V(A_{i1}) = V(A_{i2})$.

Note that the value function must induce complete order. It means that, for any pair of alternatives A_{i1} and A_{i2}, either one is preferred to the other or there is indifference between them, i.e. either $A_{i1} \succ A_{i2}$ or $A_{i2} \succ A_{i1}$ or $A_{i1} \sim A_{i2}$. This fact also means that preferences and indifferences are transitive, i.e., if $A_{i1} \succ A_{i2}$ and $A_{i2} \succ A_{i3}$, then $A_{i1} \succ A_{i3}$. The same holds for indifference.

The value measurement approach introduces partial value functions $v_j(A_i)$. They are constructed for each criterion and partial value functions hold the essential features (i.e., induce complete order) of a value function in terms of separate criteria.

We would like to emphasise the difference between x_{ij} and $v_j(A_i)$, i.e., between the quantitative performance of alternative A_i in terms of criterion C_j and the value of a partial value function for criterion C_j in terms of alternative A_i. By x_{ij} we mean a direct measure (performance) of a certain alternative in terms of certain criteria.

For example the metrics (criteria) of evaluating the algorithm-code complex efficiency (attribute) might be the algorithm-code complex performance in seconds. While the partial value function might be simply a function of x_{ij} or it can even be standardized so that the worst outcome results in zero value and the best outcome results in a convenient standard value (e.g. 100). The value function does not have to be linear.

Several value measurement theory algorithms were developed and the most popular ones are *Weighted Sum Model*, *Weighted Product Model.* We would also assign the Topsis algorithm to the same category of algorithms.

Weighted Sum Model (WSM) is the most commonly used method for single decision making problems (Triantaphyllou, 2000). It can be described using the following formula:

$$V(A_i) = \sum_{j=1}^{m} w_j v_j(A_i) \tag{3.2}$$

where $i = 1, 2, \cdots n$ and $j = 1, 2, \cdots m$.

One of the reasons for a wide acceptance of this model is its simplicity, i.e., it can be easily explained by decision makers to a variety of backgrounds (Belton and Stewart, 2003).

Note that the requirement *preferential independence* must be satisfied so that the WSM model could be applied (Belton and Stewart, 2003). Suppose that two alternatives A_{i1} and A_{i2} differ only in a set of criteria $R \subset C$ (R is a proper subset of C) and the values of partial functions are equal in all the other criteria. Then it is possible to decide the relationship of A_{i1} and A_{i2} (i.e. $A_{i2} \succ A_{i1}$ or $A_{i1} \succ A_{i2}$ or $A_{i1} \sim A_{i2}$) knowing their performances on criteria from R only, i.e., irrespective of the values of their performances in all the other criteria.

The criteria obtained after performing the problem structuring phase are presented in Chapter 6. However, among the criteria there are several dependent criteria, e.g., the quality of programming style is related either to the performance of an algorithm-code complex or to its efforts to solve the task. Thus a partial independence of criteria is violated.

We suppose that this does not eliminate WSM from applying it to the score aggregation in LitIO. WSM can still be applied in aggregating those criteria that are preferentially independent. The aggregation of dependent criteria will have to be calculated separately. WSM can be potentially applied to the score aggregation in LitIO, though the above mentioned condition must be observed.

Weighted Product Model (WPM). WPM can be described using the following formula:

$$V(A_i) = \prod_{j=1}^{m} [v_j(A_i)]^{w_j} \tag{3.3}$$

where $i = 1, 2, \cdots, n$ and $j = 1, 2, \cdots, m$.

Arguments have been presented that preferences are often perceived in the ratio scale terms, therefore product is more natural than sum (Lootsma, 1997; Triantaphyllou, 2000). The consequence of tradeof an additive approach to a multiplicative approach is that partial value functions have to satisfy the ratio scale properties instead of interval scale properties. WPM is also proposed in the cases where a different than described here version of WSM is used. Then WPM should solve a problem when different criteria are measured in terms of different measurement units and there is no way how the values of performance in terms of different criteria can be added directly (Triantaphyllou, 2000). The WSM presented above uses partial value functions which can solve this issue.

Simplicity of the approach is a high priority in the choice of the score aggregation algorithm. We conclude that the WSM algorithm would be more suitable than WPM as it is simpler and better understandable to a wide audience and, otherwise, they seem to be identical in terms of the problem under consideration.

Topsis (Technique for Order Preference by Similarity to Ideal Solution) (Saghafian and Hejazi, 2005; Triantaphyllou, 2000). We did not find it explicitly stating that Topsis belongs either to Value measurement theory approaches, or to another specific family of MCDA approaches. However, as it involves calculating the value of the closeness coefficient and ranking, based on the values of the coefficient, we assume that it is appropriate to consider it here.

Topsis introduces concepts of hypothetical solutions, i.e. the positive ideal solution and the negative ideal solution. The positive ideal solution is calculated as a function from the best performance values of the concrete decision matrix in terms of each criterion. The negative ideal solution is calculated as a function from the worst performance values in terms of each criterion. For each alternative the Euclidean distance from the ideal positive solution and the ideal negative solution is calculated. Finally, a relative closeness coefficient to the ideal positive solution is calculated and the alternatives are ranked, based on the value of the relative closeness coefficient to the ideal solution of each alternative. This method from the mathematical point of view is interesting and appealing, however, it concedes to WSM due to the simplicity of the latter.

After looking at several value measurement theory associated methods, we arrived at the conclusion, that simplicity and the ability of a wide audience to accept the evaluation scheme plays a significant role in the choice of approaches, therefore the WSM approach suits best for solving evaluation in the LitIO problem. Though certain requirements have to be observed. We did not find any evidence that other methods would be more suitable than WSM.

3.8.5 Fuzzy Set Theory and Its Applications in MCDA

We are all well aware that it is very difficult to present precise descriptions of real life physical situations. The main problem is that the transition from one situation to another is sharp in descriptions, but not in the real life. For example, submissions have to be sorted into two categories: *passed* and *failed*. The jury can agree that the lowest score for *passed* is 40 out of 100, i.e., all the submissions that scored 40 and more are considered as *passed*.

However, at the same time we understand that in reality the boundary between *passed* and *failed* is not so sharp. The difference between a submission which scored 39 points and failed and a submission which scored 40 points and passed is not so significant for us that we could claim that they definitely belong to different categories. Fuzzy sets contrary to the classical crisp sets take into account the fuzziness of real life situations.

Fuzzy sets were proposed in 1965 (Zadeh, 1965, 1968) in order to quantify fuzziness that is encountered in real life situations. By fuzziness we mean a situation associated with sets when there is no sharp transition from the membership to a non-membership situation, i.e., intermediate values between conventional values like true/false are allowed. Currently the fuzzy logic is a very powerful tool for using it in expert systems, complex industrial processes as well as MCDA (Hellmann, 2001; Triantaphyllou, 2000).

3.8.5.1 Main Crisp and Fuzzy Set Related Concepts

The fuzzy logic concepts presented in this thesis are based on (Lee, 2005; Triantaphyllou, 2000).

Crisp set. Any collection of objects from the given universe regardless of their order. For any object from the given universe its membership in the crisp set must be unambiguously defined.

Membership function. Crisp sets can be mapped to functions. Suppose X is a universe. For any crisp set A we define its membership function $\mu_A : X \to \{0, 1\}$ in the following way:

$$\mu_A : (x) = \begin{cases} 1, & \text{iff } x \in A \\ 0, & \text{iff } x \notin A \end{cases}, \text{ for each } x \in X \tag{3.4}$$

Fuzzy set. A fuzzy set is any set that allows its members to have different grades of membership (membership function) in the interval $[0, 1]$, i.e., for any subset $\widetilde{A}$ of the universe X it is possible to define a membership function of a fuzzy set: $\mu_{\widetilde{A}} : X \to [0, 1]$.

$\widetilde{A}$ is completely defined by a set of tuples $\widetilde{A} = \{(x, \mu_{\widetilde{A}}(x) \mid x \in X\}$.

A crisp set is a separate case of fuzzy set, and to distinguish between the crisp and fuzzy sets we will use $\widetilde{A}$ notation for fuzzy sets.

Convex fuzzy Set. Suppose $a \in \mathbb{R}$, $b \in \mathbb{R}$, X is defined in $\mathbb{R}$ and $t = \lambda a + b(1 - \lambda)$ where $\lambda \in [0,\ 1]$ and is freely chosen. The fuzzy set $\widetilde{A}$ is convex if $\mu_{\widetilde{A}}(t) \geq \min\ [\mu_{\widetilde{A}}(a),\ \mu_{\widetilde{A}}(b)]$.

Normalised fuzzy set. A fuzzy set $\widetilde{A}$ is normalised if $\exists a \in \widetilde{A} \mid \mu_{\widetilde{A}}(a) = 1$.

Operations on Fuzzy sets. Similarly to the operations on crisps sets, operations can be introduced on fuzzy sets. Fuzzy operations were proposed by (Zadeh, 1965) and a more detailed reference on fuzzy set operations can be found in (Lee, 2005; Triantaphyllou, 2000). We present definitions of the main operations: union, intersection, negation.

- *Negation:* $\mu_{\bar{\widetilde{A}}}(x) = 1 - \mu_{\widetilde{A}}(x), \forall x \in X$,
- *Union:* $\mu_{\widetilde{A} \cup \widetilde{B}}(x) = Max\ [\mu_{\widetilde{A}}(x),\ \mu_{\widetilde{B}}(x)], \forall x \in X$,
- *Intersection:* $\mu_{\widetilde{A} \cap \widetilde{B}}(x) = Min\ [\mu_{\widetilde{A}}(x),\ \mu_{\widetilde{B}}(x)], \forall x \in X$, (Fig. 3.5)

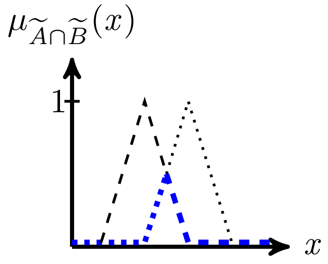

Figure 3.5: Operation of the intersection of two fuzzy sets

Fuzzy number. A fuzzy set is called a fuzzy number, if the fuzzy set is convex, normalised, its membership function is defined in $\mathbb{R}$ and is piecewise continuous. Examples of fuzzy numbers are presented in Figs. 3.6, 3.7.

Triangular fuzzy number. A triangular fuzzy number is a fuzzy number represented by three points as follows: $\widetilde{A} = (a_1, a_2, a_3)$ and this representation is interpreted in the following way:

$$\mu_{\widetilde{A}}(x) = \begin{cases} 0, & x < a_1, \\ \frac{x-a_1}{a_2-a_1}, & a_1 \leq x \leq a_2, \\ \frac{a_3-x}{a_3-a_2}, & a_2 \leq x \leq a_3, \\ 0, & x > a_3. \end{cases} \tag{3.5}$$

An example of the triangular fuzzy number is given in Fig. 3.6.

Trapezoidal fuzzy number. A trapezoidal fuzzy number is a fuzzy number represented byh four points as follows: $\widetilde{A} = (a_1, a_2, a_3, a_4)$ and this representation

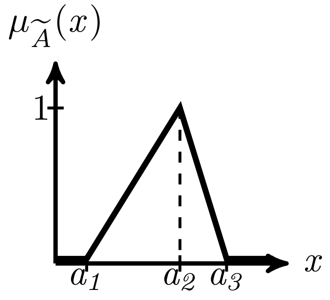

Figure 3.6: Triangular fuzzy number $\widetilde{A} = (a_1, a_2, a_3)$.

is interpreted in the following way:

$$\mu_{\widetilde{A}}(x) = \begin{cases} 0, & x < a_1 \\ \frac{x-a_1}{a_2-a_1}, & a_1 \leq x \leq a_2 \\ 1, & a_2 \leq x \leq a_3 \\ \frac{a_4-x}{a_4-a_3}, & a_3 \leq x \leq a_4 \\ 0, & x > a_4 \end{cases} \tag{3.6}$$

Ifn $a_2 = a_3$, the trapezoidal number is coincident with the triangular fuzzy number. An example of the trapezoidal fuzzy number is given in Fig. 3.7.

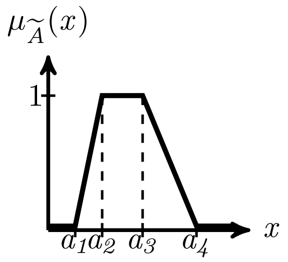

Figure 3.7: Trapezoidal fuzzy number

Addition and multiplication of triangular fuzzy numbers. These operations were developed by (Laarhoven and Pedrycz, 1983).

- *Addition:* $\widetilde{A} \oplus \widetilde{B} = (a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- *Multiplication:* $\widetilde{A} \otimes \widetilde{B} = (a_1 \times b_1, a_2 \times b_2, a_3 \times b_3)$

Ranking triangular fuzzy numbers. There exist many different algorithms for ranking fuzzy numbers. (Triantaphyllou, 2000; Zhu and Lee, 1991) proposes the following one. Let $\widetilde{A}$ and $\widetilde{B}$ bee triangular fuzzy numbers. We define:

$$e(\widetilde{A}, \widetilde{B}) = \underset{x \geq y}{max}[min[\mu_{\widetilde{A}}(x), \mu_{\widetilde{B}}(y)]] \tag{3.7}$$

The triangular fuzzy number $\widetilde{A}$ dominates (or outranks) $\widetilde{B}$ if and only if $e(\widetilde{A}, \widetilde{B}) = 1$ and $e(\widetilde{B}, \widetilde{A}) < Q$, where Q is a fixed positive fraction smaller than 1 (e.g. 0.9). Value Q is set by the decision analyst and later is possibly checked for sensitivity.

3.8.5.2 Application of Fuzzy Numbers in Quantifying Linguistic Variables

In the evaluation in LitIO we deal with both crisp and linguistic data. Linguistic variables are variables whose values are linguistic terms, not numbers. They are used to express the results of subjective qualitative evaluation. Linguistic variables were introduced and described by (Zadeh, 1975a,b,c). Triangular and trapezoidal fuzzy numbers are used for quantifying linguistic variables.

Item of linguistic scale	Numerical weights
Very poor (VP)	$(0, 0, 0, 0.2)$
Between poor and very poor (BPV)	$(0, 0.2, 0.2, 0.4)$
Poor (P)	$(0, 0.2, 0.2, 0.4)$
Between poor and fair (BPF)	$(0, 0.2, 0.5, 0.7)$
Fair (F)	$(0.3, 0.5, 0.5, 0.7)$
Between fair and good (BFG)	$(0.3, 0.5, 0.8, 1)$
Good (G)	$(0.6, 0.8, 0.8, 1)$
Between good and very good (BGV)	$(0.6, 0.8, 0.8, 1)$
Very good (VG)	$(0.8, 1, 1, 1)$

Table 3.1: Weights of a trapezoidal distribution of a linguistic scale (Sule, 2001).

Many conversion scales were created for transforming linguistic terms into fuzzy numbers. (Chen et al., 1992) proposed eight conversion scales with different numbers of linguistic terms which are commonly used. An example of pretty standard in fuzzy set theory a nine-item scale is presented in Table 3.1 and Fig. 3.8 (Sule, 2001). The choice of a concrete scale from the available ones is intuitive and left for the responsibility of the decision maker.

3.8.5.3 Application of Fuzzy Logic in Solving MCDA Problems

In MCDA the data can be categorised into three groups: all data are crisp, all data are fuzzy, data are either crisp or fuzzy (Zhang, 2004). Many classical MCDA

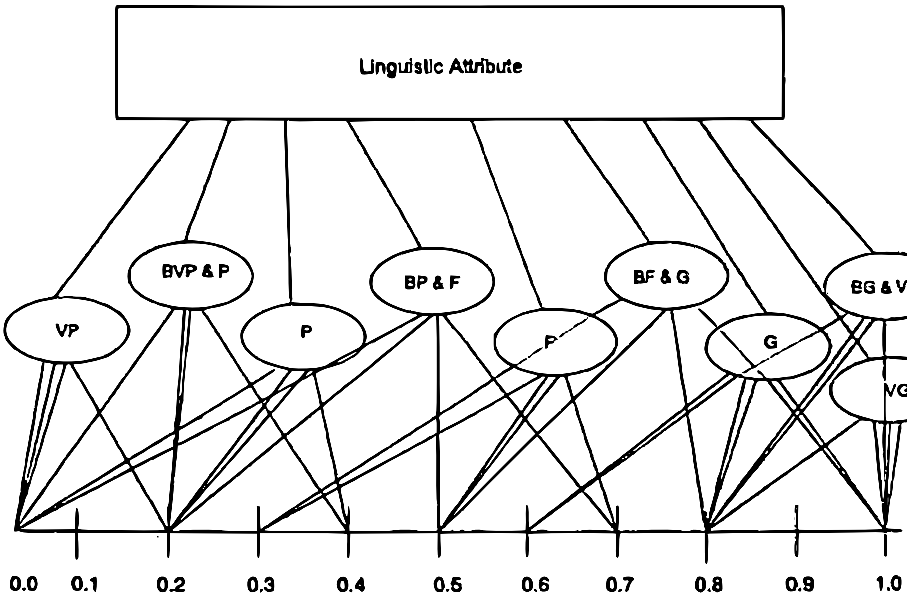

Figure 3.8: Trapezoidal fuzzy numbers are used to quantify the nine-item linguistic scale (Sule, 2001)

algorithms are modified and adapted to apply them to fuzzy data. Among the adapted ones are WSM, WPM, and Topsis.

A systematic and critical study of the existing fuzzy MCDA methods was performed. A conclusion was drawn, that the majority of currently existing fuzzy MCDA approaches involve complicated calculations, require all the elements of decision matrix to be presented in a fuzzy format (though some of them might be crisp), and are not suitable for solving problems with more than ten alternatives associated with more than ten criteria (Chen et al., 1992; Rao, 2007).

The method presented by (Chen et al., 1992) is considered to be the one which avoids the above mentioned problems (Rao, 2007; Zhang, 2004). It consists of the following phases:

- Linguistic terms (if such are used) are converted to fuzzy numbers.
- Fuzzy numbers are converted into crisp scores.
- Classical MCDA approaches, which assume crisp values, are applied.

The crisp score of the fuzzy number $\widetilde{A}$ is calculated in the following way. First, two functions $\mu_{max}(x)$ and $\mu_{min}(x)$ are defined:

$$\mu_{max}(x) = \begin{cases} x, & 0 \leq x \leq 1 \\ 0, & otherwise \end{cases} \tag{3.8}$$

$$\mu_{min}(x) = \begin{cases} 1 - x, & 0 \leq x \leq 1 \\ 0, & otherwise \end{cases} \tag{3.9}$$

Then the left and the right scores of $\widetilde{A}$ are defined as:

$$\mu_L(\widetilde{A}) = \underset{x}{Sup}[\mu_{\widetilde{A}}(x) \cap \mu_{min}(x)] \tag{3.10}$$

$$\mu_R(\widetilde{A}) = \underset{x}{Sup}[\mu_{\widetilde{A}}(x) \cap \mu_{max}(x)] \tag{3.11}$$

Here *Sup* stands for the least upper bound. The total crisp score of the fuzzy number $\widetilde{A}$ is defined as:

$$\mu_T(\widetilde{A}) = (\mu_R(\widetilde{A}) + 1 - \mu_L(\widetilde{A}))/2 \tag{3.12}$$

The conversion of a fuzzy number to a crisp value is illustrated in Fig. 3.9.

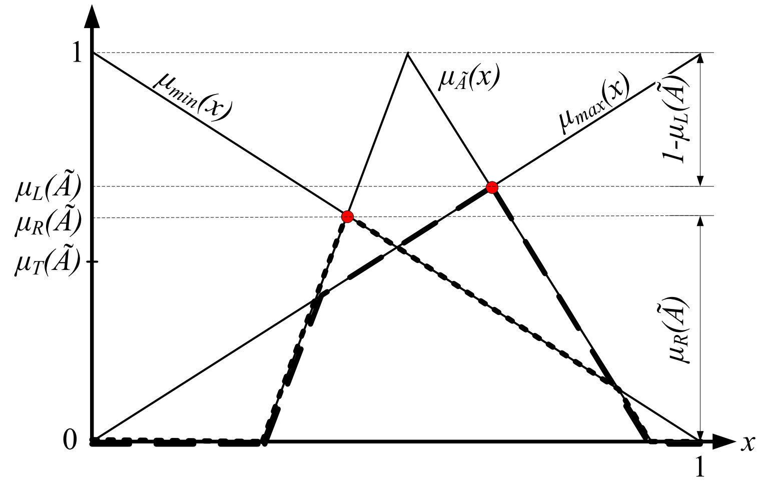

Figure 3.9: Conversion of a triangular fuzzy number to a crisp value

The values of the nine-item linguistic scale, presented in Table 3.1, are converted to crisp values and presented in Table 3.2. Note that the same linguistic term in different conversion scales can have different crisp values.

Just for comparison, as an alternative to the approach of (Chen et al., 1992), we present the fuzzy WSM method (Triantaphyllou, 2000).

Let $\widetilde{w}_1, \widetilde{w}_2, \cdots, \widetilde{w}_m$ be the fuzzy weights of the criteria where each weight $\widetilde{w}_j$ is a triangular fuzzy number[1]: $\widetilde{w}_j = (w_{ja_1}, w_{ja_2}, w_{ja_3})$, $j = 1, 2, \cdots, m$. Similarly, $\widetilde{x_{ij}}$ are

[1]It is also possible to use trapezoidal numbers. This choice of triangular fuzzy numbers was made for the sake of simplicity.

Item of linguistic scale	**Fuzzy number** $\widetilde{A}$	$\mu_R(\widetilde{A})$	$\mu_L(\widetilde{A})$	$\mu_T(\widetilde{A})$
Very poor	$(0, 0, 0, 0.2)$	1	0.17	0.08
Between poor and very poor	$(0, 0.2, 0.2, 0.4)$	0.83	0.33	0.25
Poor	$(0, 0.2, 0.2, 0.4)$	0.83	0.33	0.25
Between poor and fair	$(0, 0.2, 0.5, 0.7)$	0.83	0.58	0.38
Fair	$(0.3, 0.5, 0.5, 0.7)$	0.58	0.58	0.50
Between fair and good	$(0.3, 0.5, 0.8, 1)$	0.58	0.83	0.63
Good	$(0.6, 0.8, 0.8, 1)$	0.33	0.83	0.75
Between good and very good	$(0.6, 0.8, 0.8, 1)$	0.33	0.83	0.75
Very good	$(0.8, 1, 1, 1)$	0.17	1.00	0.92

Table 3.2: Calculation of crisp values of the nine-item linguistic scale, given in Table 3.1

performances of each alternative in terms of each criterion expressed as triangular fuzzy numbers: $\widetilde{x_{ij}} = (x_{ij_{a_1}}, x_{ij_{a_2}}, x_{ij_{a_3}})$, $i = 1, 2, \cdots, n$ and $j = 1, 2, \cdots, m$.

Then the fuzzy WSM method can be described by the following value function formula:

$$V(\widetilde{A_i}) = \sum_{j=1}^{m} \widetilde{w_j} \otimes v_j(\widetilde{A_i}) \tag{3.13}$$

The calculated value function $V(\widetilde{A_i})$ values are fuzzy numbers. Afterwards, the ranking procedure of fuzzy numbers is chosen and the alternatives are ranked. (Triantaphyllou, 2000) suggests to use the ranking procedure described in 3.8.5.1 for the fuzzy WSM algorithm.

The fuzzy WSM method gives in to the approach of (Chen et al., 1992) in the context of our research. Fuzzy WSM requires fuzzification of all the crisp values of the decision matrix, the value function is a triangular fuzzy number, i.e., it consists of three real numbers, the true meaning of which might be difficult to explain to the wide audience interested in programming contests. Ranking is performed by applying the fuzzy numbers ranking procedure which is also difficult to explain to the wide audience.

The algorithm for converting fuzzy numbers to crisp values might also be hardly understandable to the wide audience, however, its application, will remain invisible for the contestants and their coaches. It will only be applied in dealing with group decisions and linguistic evaluation. If a criterion requires manual evaluation, the linguistic scores and the scores of individual jury members are never revealed to the contestants, just the aggregate score for the criterion is announced. Thus, if fuzzy techniques are used to aggregate the scores of several jury members, they remain behind the curtains and do not become the source of discussions and doubts for the contestants.

3.8.5.4 Group Decision Making

Group decision making (GDM) can be defined as a decision making process, based on the opinions of several individuals. The goal of GDM is to arrive at the satisfactory for the group solution, rather than at the best solution which almost does not exist (Lu et al., 2007). Various methods are available for group decision making: from mathematical to psychological and social ones.

Among MCDA approaches explicitly meant for solving group decision making problems there are techniques that foresee negotiation theory, working with group dynamics, etc. References to that can be found in (Carlsson and Fullér, 1996; Lu et al., 2007). Those approaches have been experienced in LitIO many times. Investigation of their suitability in the LitIO evaluation problem would require much investigations from other sciences, in particular, management and psychology. For example, most meetings are conducted on-line (as members of the scientific committee are associated with different universities in different cities and even countries), some members are reluctant to discuss the issues on-line, less experienced tend to vote as more experienced members, etc. These aspects should have been investigated if the above mentioned direction is taken.

Our choice is to focus on mathematical group decision making methods which assume eliciting concrete information from decision makers and using it in a mathematical algorithm, but do not require interaction and negotiation between decision makers.

There are different ways how to implement group decision making. Much references can be found int (Lu et al., 2007; Rao, 2007). Many common GDM methods (e.g. *authority rule, majority rule, negative minority rule*) are not suitable because they are intended for *the choice problematique*, but not for *the ranking problematique* problems.

(Lu et al., 2007) distinguishes three factors that influence GDM:

The weights of decision makers. Among the decision makers there might be those who play more important roles in decision making. In this case, the decision makers should be assigned different weights and that should be reflected in the group decision making process.

Weights of criteria. Decision makers may have different views, attitudes, experience and therefore propose different weights of criteria.

Preferences of decision makers to alternatives. If the performance of an alternative is evaluated subjectively, then different decision makers can have a different understanding, different experiences and evaluate performance of the same alternative in a different way.

It is common in GDM that the weight of a decision maker, the proposed weights for evaluation criteria, and the performances of alternatives suggested by decision makers are expressed by linguistic terms, since the linguistic terms reflect uncertainty, inaccuracy, and fuzziness of the decision makers (Lu et al., 2007). We also assume that the information, provided by each decision maker is consistent and non-conflicting.

3.8.5.5 Group Decision Support Algorithm

We have already concluded that the WSM approach suits best for evaluation in the LitIO problem. We were looking for an extension of WSM to GDM, such that the extension would use a crisp decision matrix for the final ranking, i.e., it would be acceptable by the community of LitIO. Many fuzzy GDM algorithms (e.g. *an intelligent FMCGDM method* (Lu et al., 2007) or the one described in (Sule, 2001)) assume first aggregating fuzzy numbers and only then deriving the final ranking.

The group decision support algorithm (Csáki et al., 1995) uses crisp values. Therefore after combining it with the approach of (Chen et al., 1992) we obtain a GDM algorithm suitable to apply in the LitIO evaluation problem:

Let $A = \{A_1, A_2, \cdots A_n\}$ be a finite set of alternatives and $C = \{C_1, C_2, \cdots C_m\}$ be a finite set of criteria. Let $D = \{D_1, D_2, \cdots, D_t\}$, $t \geq 2$ be a finite set of decision makers.

Each decision maker is assigned a linguistic weight of his/her importance $\widetilde{p} = \{\widetilde{p_1}, \widetilde{p_2}, \cdots, \widetilde{p_t}\}$.

Each criterion is assigned a linguistic weight of its importance by each decision maker $\widetilde{w}^k = \{\widetilde{w}_1^k, \widetilde{w}_2^k, \cdots, \widetilde{w}_m^k\}$, $k = 1, 2, \cdots, t$.

Let $\widetilde{v}_j^k(A_i)$ be the values of partial value functions of the performance of alternative A_i in terms of each criterion C_j by the decision maker D_k, where $i = 1, 2, \cdots, n$, $j = 1, 2, \cdots, m$, and $k = 1, 2, \cdots, t$. If the evaluation of performance of alternatives of some criteria is subjective, then the values of the partial value functions are linguistic. Otherwise, they are either crisp or fuzzy.

Following the approach of (Chen et al., 1992) described in 3.8.5.3, first all the linguistic terms are converted to fuzzy numbers and afterwards all the fuzzy numbers are converted to crisp values.

Next we apply the *group decision support* algorithm (Csáki et al., 1995). Note that this algorithm does *not* require $\sum_{k=1}^{t} p_k = 1$.

First the aggregated group weights for each criterion are calculated:

$$w_j = \frac{\sum_{k=1}^{t} w_j^k p_k}{\sum_{k=1}^{t} p_k}, j = 1, 2, \cdots, m \tag{3.14}$$

The values of partial value functions of the performance of each alternative in terms of each criterion are calculated in a similar way:

$$v_j(A_i) = \frac{\sum_{k=1}^{t} v_j^k(A_i) p_k}{\sum_{k=1}^{t} p_k} \tag{3.15}$$

The aggregated values for each alternative are calculated in the following way:

$$v(A_i) = \frac{\sum_{j=1}^{m} v_j(A_i) w_j}{\sum_{j=1}^{m} w_j} \tag{3.16}$$

Based on the calculated values, the ranking of alternatives is performed.

3.9 Sensitivity Analysis

By sensitivity analysis we understand checking for ranking reversals by changing relative weights (Rao, 2007).

An extensive overview of research in sensitivity analysis can be found in (Triantaphyllou, 2000). It indicates that there is a considerable sensitivity research for some types of MCDA problems, such as inventory models or investment analysis, cases with partial and doubtful data, however the research for deterministic MCDA problems is very limited.

In decision making it is intuitively supposed that the criterion with the highest weight is most critical (Winston, 1991). However, this is not always true, especially when the criteria are qualitative. Being aware how critical each criterion is (i.e., how sensitive the ranking of alternatives is to the changes in the current weights of the criteria), decision makers can make better decisions.

Two major sensitivity analysis problems are defined (Triantaphyllou, 2000). The first problem is *determining the most critical criterion.* The second sensitivity problem is *determining the most critical measure of performance.*

3.9.1 The Most Critical Criterion

There exist two alternative definitions of the most critical criterion. The first definition associates the most critical criterion with the answer to the question of whether the best (top) alternatives reverse. The second definition is associated with the answer to the question whether there are changes in the ranking of any alternatives.

We assume that the decision matrix of an MCDA problem was defined 3.1 and the solution to the decision problem is already calculated: $F = \{F_1, F_2, \cdots, F_n\}$ where $F_i = V(A_i)$ represent the final preference. We also assume without loss of generality that $F_1 \geq F_2 \geq \cdots \geq F_n$, i.e., that the first alternative is the most preferred one (this can be achieved by simply rearranging the indices).

Next we present several definitions from (Triantaphyllou, 2000).

Minimum change for reversing two alternatives. Let $\delta_{ji_1i_2}$ for $1 \leq i_1 < i_2 \leq n$, and $1 \leq j \leq m$, denote the minimum change in the current weight w_j of criterion C_j such that the ranking of two alternatives A_{i_1} and A_{i_2} is reversed.

Note that A_{i_1} is a more preferred alternative than A_{i_2} due to the assumption above.

Minimum relative change for reversing two alternatives.

$$\delta'_{ji_1i_2} = \frac{\delta_{ji_1i_2} \times 100}{w_j}, 1 \leq i_1 < i_2 \leq n, \text{ and } 1 \leq j \leq m. \tag{3.17}$$

The Percent-Top (or PT) critical criterion is the criterion which corresponds to the smallest value of $|\delta'_{j1i_2}|$ where $1 \leq j \leq m$ and $1 \leq i_2 \leq n$.

The Percent-Any (or PA) critical criterion is the criterion which corresponds to the smallest value of $|\delta'_{ji_1i_2}|$ where $1 \leq j \leq m$ and $1 \leq i_1 < i_2 \leq n$.

The criticality degree of criterion C_j denoted as D'_j, is the smallest percent amount by which the current value of w_j must change so that the existing ranking of the alternative will change:

$$D'_j = \min_{1 \le i_1 < i_2 \le n} \{|\delta'_{ji_1i_2}|\}, \text{ for all } 1 \le j \le m. \tag{3.18}$$

The sensitivity coefficient of criterion C_j denoted as $sens(C_j)$, is the reciprocal of its criticality degree:

$$sens(C_j) = \frac{1}{D'_j}, \text{ for all } 1 \le j \le m. \tag{3.19}$$

If the criticality degree is infeasible, i.e., it is not possible to change the ranking of alternatives by changing the weight, then the sensitivity coefficient is equal to zero.

It has been proved theoretically (Triantaphyllou, 2000) that if WSM is used, then the minimum relative change for reversing two alternatives is given as follows:

$$\begin{cases} \delta'_{ji_1i_2} < \frac{F_{i_2}-F_{i_1}}{v_j(A_{i_1})-v_j(A_{i_2})} \times \frac{100}{w_j}, & \text{if } v_j(A_{i_1}) > v_j(A_{i_2}), \text{ or} \\ \delta'_{ji_1i_2} > \frac{F_{i_2}-F_{i_1}}{v_j(A_{i_1})-v_j(A_{i_2})} \times \frac{100}{w_j}, & \text{if } v_j(A_{i_1}) < v_j(A_{i_2}) \end{cases} \tag{3.20}$$

for each $1 \le i_1 < i_2 \le n$ and $1 \le j \le m$.
The following condition should be satisfied for the value to be feasible:

$$\frac{F_{i_2} - F_{i_1}}{v_j(A_{i_1}) - v_j(A_{i_2})} \le w_j. \tag{3.21}$$

3.9.2 The Most Critical Measure of Performance

The sensitivity analysis problem examined in this subsection is determination of the most critical value of performance $v_j(A_i)$.

The threshold value of $v_j(A_i)$, denoted as τ_{ijk} for $1 \le i < k \le n$, $1 \le j \le m$, is defined as the minimum change which has to occur in the current value of $v_j(A_i)$ so that the current ranking between A_i and A_k will change.

The relative threshold value of $v_j(A_i)$ is denoted as τ'_{ijk} and defined in the following way:

$$\tau'_{ijk} = \frac{\tau_{ijk} \times 100}{v_j(A_i)}, \text{ for any } 1 \le i < k \le n, \text{ and } 1 \le j \le m. \tag{3.22}$$

The most sensitive alternative is the one which is associated with the smallest threshold value.

The criticality degree of alternative A_i in terms of criterion C_j is defined as the smallest amount (%) by which the current value of $v_j(A_i)$ must change so that the existing ranking of alternative A_i will change:

$$\Delta'_{ij} = \underset{k \neq i}{min}\{|\tau'_{ijk}|\}, \text{ for any } 1 \leq i \leq n, \text{ and } 1 \leq j \leq m. \tag{3.23}$$

The most critical alternative A_L is the one which is associated with the smallest criticality degree:

$$\Delta'_{Lk} = \underset{1 \leq i \leq m}{min}\{\underset{1 \leq j \leq n}{min}\{\Delta_{ij}\}\}, \text{ for some } 1 \leq k \leq n. \tag{3.24}$$

The sensitivity coefficient of alternative A_i **in terms of criterion** C_j is defined as the reciprocal of the criticality degree:

$$sens(v_j(A_i)) = \frac{1}{\Delta'_{ij}}, \text{ for any } 1 \leq i \leq n, \text{ and } 1 \leq j \leq m. \tag{3.25}$$

If the criticality degree is infeasible, then the sensitivity coefficient is set to be equal to zero.

From the definitions above it follows that the most sensitive alternative is the one with the highest sensitivity coefficient.

It has been proved (Triantaphyllou, 2000) that, if WSM is used, then the relative threshold value is calculated as follows:

$$\begin{cases} \tau'_{ijk} < \frac{F_i - F_k}{w_j} \times \frac{100}{v_j(A_i)}, & \text{when } i < k, \quad \text{or} \\ \tau'_{ijk} > \frac{F_i - F_k}{w_j} \times \frac{100}{v_j(A_i)}, & \text{when } i > k. \end{cases} \tag{3.26}$$

The threshold value is feasible, if the following condition is satisfied: $\tau'_{ijk} \leq 100$.

3.10 Conclusions

The evaluation in LitIO can be defined as a group decision making repeated problem, belonging to the category of ranking problematigue. Submissions play the role of alternatives and evaluation criteria correspond to the concept of criteria in MCDA terminology. Therefore evaluation in the LitIO problem can be addressed using MCDA approaches.

There are different options how to decompose an MCDA process, however, we have chosen a decomposition consisting of three stages: problem structuring, problem modelling, and problem analysis.

From many methods how to perform problem structuring we selected the Goal/ Question/Metric approach, because it foresees a systematic way of deriving software metrics for measuring its quality. I.e., differently than other problem structuring methods, it is intended to deal with software metrics.

Problem modelling requires to define preferences in terms of each individual criterion and a score aggregation model. However, in informatics contests we deal with a very sensitive audience, the contestants. The choice of the MCDA algorithm should be easily understandable by the wide audience and should provide functions for expressing partial and total scores mapped to real numbers. These requirements determined our decision that *value measurement theory*, and, in particular, the *WSM algorithm* are the best to apply in the score aggregation.

Nevertheless, we had to look for an extension of *WSM* that could encompass group decision making and dealing with linguistic variables. We have found out that by combining *the group decision support algorithm* (which assumes group decision making) and *the algorithm proposed by Chen* (which uses the fuzzy logic and deals with linguistic variables), we get an algorithm suitable for modeling our problem.

Note that the use of this algorithm ensures an understandable for the wide audience scoring function, and more complicated calculations are used only in such parts of score aggregation which are not revealed to the contestants (e.g., decision on the weights of evaluation criteria).

For the third part of problem structuring, the model analysis, we identified the algorithms for calculating the most critical criterion and the most critical measure of performance. Those two measures can be used for calculating the model sensitivity.

4 Semi-Automated Visualisation: Aid for Tasks Involving Graphs

4.1 Introduction

Automated evaluation dominates in informatics contests, like IOI or LitIO. One of the reasons is that manual or semi-automated evaluation, takes too much time, especially if the jury has to analyse the source code and understand how it works. The programs contain cultural elements (e.g. the contestants name the variables in their native languages), and the programming style is different or even poor.

A tool that simplifies understanding and analysis of algorithm-code complexes would be valuable. However, it is hard to create one tool suitable for all cases. Therefore, we decided to focus on tasks with graphs.

The tasks, that include graphs as a problem and/or solution feature, are very common in informatics contests. In the final round of LitIO such tasks made over 20% of all the tasks (Dagienė and Skūpienė, 2003). The tasks with graphs made 22% of all the tasks in the Baltic Olympiad in Informatics in the years 1995-2008 (Poranen et al., 2009). Such tasks are common in the IOI and other informatics contests (Manev, 2008; Verhoeff, 2009).

We analysed graph implementations in submissions of three selected LitIO tasks, identified most common graph representations, and created an experimental tool for visualisation of Pascal programs that contain graph implementations.

Note, that this work was done several years ago, when Pascal was popular both in international contests and in LitIO. The survey of IOI'2004 reveals that 46.58% of the contestants used Pascal during the contest (IOI, 2004). In BOI'2005 (Baltic Olympiad in Informatics), 49.1% of contestants used Pascal. In LitIO'2005, 89.3% of the contestants submitted solutions in Pascal. The available sample of C++ LitIO submissions in 2005 was not enough for the research. Since 2005 there was a shift from Pascal to C++ in informatics contests. In 2010 only 27% of LitIO finalists used Pascal.

By observing the tendencies, we might predict that Pascal as a programming language will be withdrawn from informatics contests by the contestants themselves. However, the main idea of semi-automatically visualising the graphs, implemented in the algorithm-code complexes, remains valid, and is presented as a theoretical result of this research.

4.2 Choice of Visualisation Approach

Visualization of the graphs can help to analyze the algorithm-code complexes and their behaviour during the program execution.

It is important to specify how the visualization is connected or applied to the algorithm. There are two main approaches to algorithm visualization (Demetrescu et al., 2002; Kerren and Stasko, 2002). One of them is based on an *interesting event* paradigm. The important or interesting events have to be identified in the program source, and calls to visualization procedures have to be inserted into the source.

The second approach, called *state mapping*, creates visualization automatically depending upon the values of the program variables. State mapping approach allows creating visualization without code modification, but it have some drawbacks. Visualisations can not be easily customized, lack smooth transitions. It is more difficult to represent abstractions (Sumner et al., 2003).

The choice of approach depends upon the conditions under which the algorithm-code complexes are analyzed. On the one hand, the jury have no prior knowledge of how the algorithm-code complex is designed. Moreover, the program source can have a poor programming style. On the other hand, the jury know the task, its background, as well as some model solutions. There is also a wide-spread tradition in informatics contests – the jury do not modify the contestant's source code.

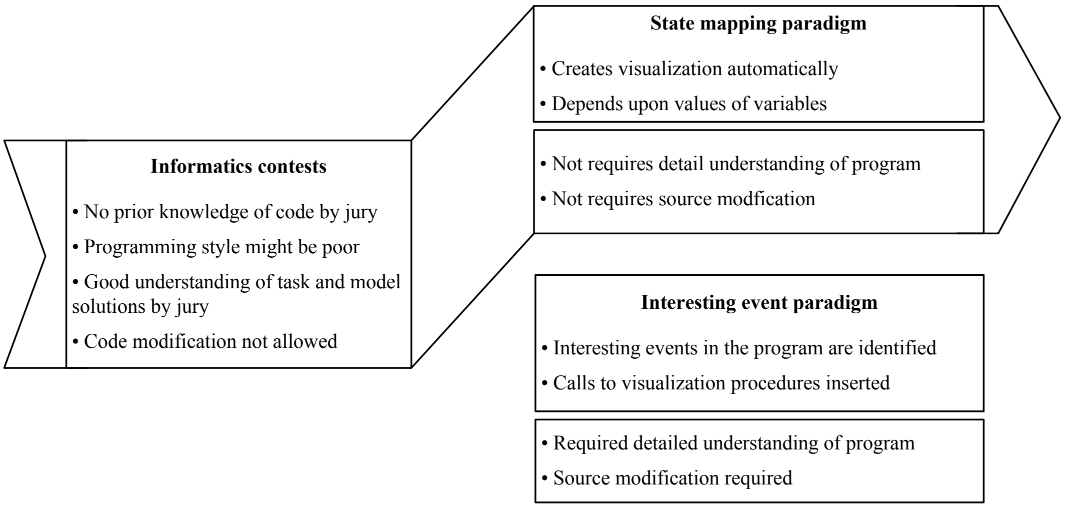
State mapping paradigm
• Creates visualization automatically
• Depends upon values of variables
• Not requires detail understanding of program
• Not requires source modfication
Informatics contests
• No prior knowledge of code by jury
• Programming style might be poor
• Good understanding of task and model solutions by jury
• Code modification not allowed
Interesting event paradigm
• Interesting events in the program are identified
• Calls to visualization procedures inserted
• Required detailed understanding of program
• Source modification required

Figure 4.1: Choice of the visualisation approach in informatics contests

The interesting event approach requires good understanding of the source code in order to identify the interesting points, and source modification or augmentation. This implies that the state mapping approach should be applied to visualise algorithm-code complexes (Fig. 4.1).

Conventional debuggers also have some features of the state mapping approach: they provide changing values of variables during the program execution (Demetrescu et al., 2002). Due to this similarity, we decided that the experimental graph visualization tool for informatics contests has to be designed as a debugger with visualization possibilities. The user of the tool, willing to get a graph animated, has to interpret the meaning of variables, identify those which represent graph data structures, and choose the method of graph implementation from the list of the available methods.

Note that, the experimental tool is not intended to display large graphs. In order to understand how an algorithm-code complex works, it is enough to analyze the program execution with small data. Large data sets help to determine how effective the solutions are. Large data are important in determining the efficiency, but not in a step by step program execution analysis.

Many other graph visualization environments are created. An example of such an environment can be EVEGA (*An Educational Visualization Environment for Graph Algorithms*) (Khuri and Holapfel, 2001). However, they are meant for the teaching purposes, and they require intervention into the source code.

4.3 Overview of Graph Implementation in Algorithm-Code Complexes Designed During the Contests

There are two common computational representations of graphs: adjacency lists and adjacency matrices. These representations can be implemented in different ways, e.g. an adjacency list can be encoded using pointers, an array of records, and a two-dimensional array. The contestants might think of many other (not necessarily reasonable) implementations. The graphs in the tasks or the solutions can be different, and have various features or attributes. Moreover, there are some tasks, where a good solution should use the memory in an efficient way. Therefore, typical implementations graphs and other data structures do not work because of predefined memory constraints.

We analyzed three final LitIO round graph tasks graphs together with their submissions, containing graph implementations. Below we present brief formulations of the tasks in graph terms. In the original task formulations, neither the word *graph*, nor other related terms (e.g. *graph vertex*, *edge*, *tree*, etc.) were used directly. Typically, in informatics contests they are described indirectly, using some kind of metaphors (Verhoeff, 2004).

Task 1. ***Acquaintance*** (LitIO'2001, Final round). N persons are expected to attend a party. It is known that if any two persons have a common acquaintance (or make one during the party), they will get acquainted during the party. However, one person did not come to the party. As a consequence, the people split into groups with no common friends, i.e., it became impossible for all the party attendants to become acquainted. Write a program to find a person whose absence might have caused such a situation.

Comment. Friendship relations represent a connected undirected graph. The task is to find an articulation vertex, i.e. a vertex removal of which disconnects the graph. It is known that the graph contains at least one articulation vertex (Fig.4.2).

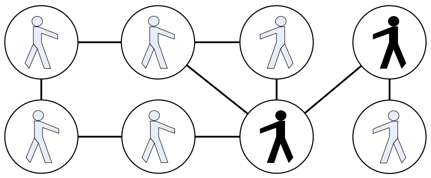

Figure 4.2: Graph in the task *Acquaintance* The graph contains two articulation vertexes marked in black.

Task 2. ***Virus*** (LitIO'2003, Final round). The computer network makes an undirected (not necessarily connected) graph. Each computer either has an anti-virus protection or not. The virus starts from computer A and passes in parallel all the edges leaving A. It travels through the network and destroys each edge it passes through. If the virus reaches the computer with an anti-virus protection, then both the virus and the anti-virus protection are destroyed. Write a program to find if and when the virus reaches computer B.

Task 3.***Computer Network*** (LitIO'2005, Final round). A set of computers and switches are to be connected into one connected network, i.e., to form a tree. Each computer has to be connected to exactly one switch. Many devices (either computers or switches) can be connected to the same switch. Given the expenses of connecting each possible pair of devices. Write a program to find the cheapest network connection.

We analysed all the submissions to the three tasks. Table 4.1 shows different options of graph implementations in the analysed algorithm-code complexes.

In the experimental tool we implemented three graph representations: adjacency lists implemented as an array of records, adjacency lists implemented as a two-dimensional array, and an adjacency matrix implemented as a two-dimensional array.

Let us comment on other, not implemented, graph representation cases. Set-based graph implementations use the Pascal set data type. This implementation is rare, because the number of elements in a Pascal set is limited. Therefore this type is not suitable to implement graphs that contain more vertexes. Pointers in graph implementations were used only for memory saving reasons, e.g., instead of a two-dimensional array, there was a pointer to the two-dimensional array. Graph representation as a list of edges, in some cases was used to represent a tree. In other cases, the contestants were presumably affected by the input data format, where the graph was presented as a list of edges. Complicated or unusual (e.g. array of strings) graph representations make a very small percentage of the analysed algorithm-code complexes.

		Task		
		Acquaintance	**Virus**	**Network**
Total number of programs		42	32	117
Adjacency lists	Array of records	6	12	–
	Two-dimensional array	12	–	–
	Two-dimensional array using pointers	–	–	1
	Two-dimensional bit array	1	–	–
	Array of records using a pointer	–	–	–
Adjacency matrix	Two-dimensional array	10	20	47
	Two-dimensional array using pointers	5	–	2
List of edges	Two-dimensional array	–	–	27
	Several arrays	–	–	16
	Array of records	–	–	29
	Array of strings	–	–	6
	Stored in a text file	–	–	4
	Other complicated structure	–	–	9
Set-based implementation	Array of sets	4	1	–
	Array of records, neighbouring vertexes stored in a set	1	2	–
Bipartite graph	Array	–	–	21
	Array of records	–	–	1
Graph implemented in more than one way		2	3	48
Graph not implemented		6	–	16

Table 4.1: Graph implementation in the analyzed algorithm-code complexes

4.4 Graphs' Visualisation in the Experimental Tool

In the experimental tool graphs are visualized so, that it were as easy as possible for the user to use it. The tool has only the main commands of a debugger i.e., only the features that are important when the program is traced in order to understand how it works.

Data structures (variables) that represent graphs, are different type of watches. The user has to indicate the variable(s), which represents the graph(s) and the implementation type. The latter is chosen from the list of available implementations. All the visualization settings can be changed or updated during debugging (tracing).

In general, it is difficult to predict which layout of the graph is most suitable for a particular task (data). Moreover, animations which change graph layouts automatically are confusing, because it is complicated for the user to follow all the changes, happening on the screen (Diehl et al., 2002). The experimental tool does not change the graph layout automatically. However, the graph might lose its good layout when many new vertexes are added during the algorithm execution. The choice is left for the user, who decides when and if to rearrange the graph automatically or manually. The user can drag vertexes and edges and modify the layout.

The analysis of the algorithm-code complexes, shows that the contestants avoid using more complicated data structures to represent complicated graphs. Instead, they create several separate data structures (components), and the graph is assembled from the components.

For example in the *Network* task it is required to find one graph which connects all the computers and switches into one network. However, in many contestants' programs two different graphs were created. One graph – to represent computer–switch connections, another – switch interconnections.

Another example comes from the *Virus* task. The contestants who implemented the graph as an array of records, used the same data structure (one field of the record) to store the information whether the computer (graph vertex) has anti-virus protection or not. The contestants who implemented the graph as a two-dimensional array, created a separate one-dimensional array to store the existence of the anti-virus protection.

The components encountered in the analysed algorithm-code complexes were classified into the following categories. *Graph component* is a graph that can be either added as a component to other graph or it can be treated as a separate graph. *Vertex component* is a list of all graph vertexes with the values assigned to each vertex (see Figs. 4.3 and 4.4). It can be added as a component to other graphs, though it cannot be visualized separately in the tool. *Vertex list component* is a list of vertexes without the assigned values, i.e. inclusion into the list already means possession of a certain attribute. A vertex list component can also be added to graphs as a component and cannot be visualized separately (see Fig. 4.5). Colour is used to depict this type of components. The tool allows connecting to the graph up to two such components.

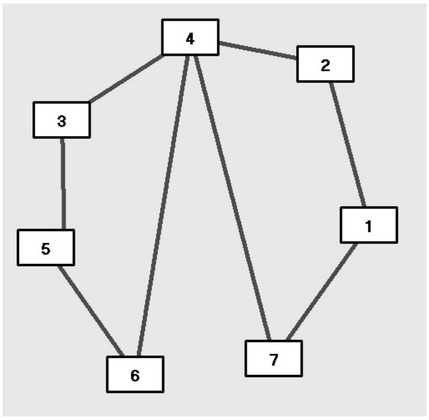

Figure 4.3: Visualised graph after reading input data Task *Acquaintances.* At the moment the graph consists of one *graph component.*

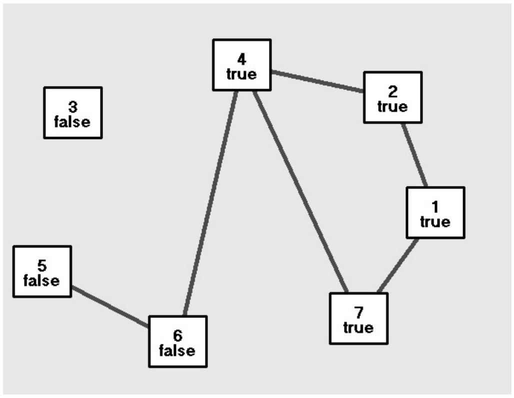

Figure 4.4: Visualised graph in task *Acquaintances* The same graph as in 4.3. Edges, leaving from vertex 3, were removed and the connectivity of the remaining graph is checked by recursively traversing it. *The vertex component* indicates whether the vertex has already been reached (*true*) or not (*false*).

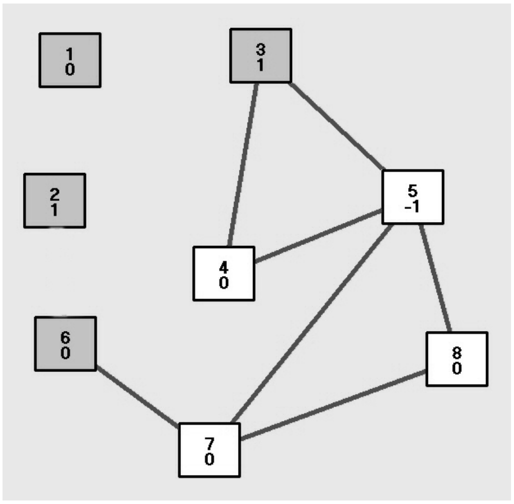

Figure 4.5: Visualised graph in task *Virus* Graph vertexes have two additional components:
1) *Vertex component*: one dimensional array where each vertex is assigned either ˘1 (computer has anti-virus protection) or 0 (no anti-virus protection) or the time when the computer was infected.
2) *Vertex list component*: set of infected vertexes (they are coloured in gray). Computers 1 and 2 were disconnected by the spreading virus.

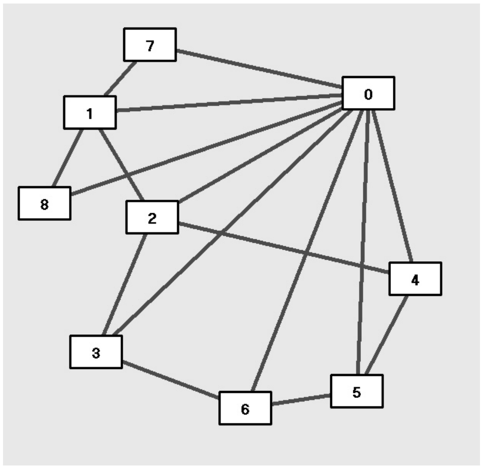

Figure 4.6: Misrepresented graph in task *Acquaintances* View of the graph after reading input data. The graph was implemented as a two-dimensional array, however, the *zero* column was introduced to store the degree of a vertex. Therefore the graph is misrepresented.

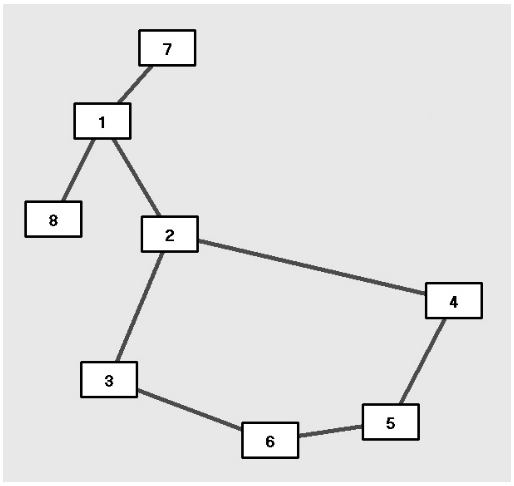

Figure 4.7: Fixed graph representation in task *Acquaintances* *Zero* column is marked as non displayable. Now the graph representation is correct.

		Task		
		Acquaintance	**Virus**	**Network**
No of programs with graphs implemented		36	32	101
No of programs without extra components		5	7	8
No of programs with	1 component	17	7	29
	2 components	10	7	30
	3 components	4	6	18
	≥ 4 components	–	5	17
Vertex components	Set	8	3	1
	One dimensional array	7	11	4
	Dynamical list	1	–	–
	Array of enumerations	–	–	1
Vertex list components	One dimensional array	19	16	26
	Array of records	1	5	–
	Array of records using a pointer	–	1	-
Graphs as components		2	3	81

Table 4.2: Statistics of the graph components implemented separately from the main graph

Statistics of graph components implemented separately from the main graph is presented in Table 4.2.

The tool allows visualizing several different graphs at the same time, and they can be assembled from the described components. The same components can be added to different graphs at the same time.

We also encountered the cases, where the contestants implemented the graph as a two-dimensional array, and used a column of the same array to store the vertex components. This cannot be identified automatically by the tool. Therefore the tool foresees a possibility to mark one column or a row as an non displayable one, or as a vertex component (see Figs. 4.6 and 4.7).

4.5 Conclusions

The analysis of submissions to the three graph tasks shows that a limited number of different graph implementations is used in the majority of analysed algorithm-code complexes. This makes it possible to implement the semi-automated visualisation of graphs encoded in the algorithm-code complexes. We also discovered the tendency among the contestants to implement more complicated graphs by decomposing the graphs into separate structures.

We created an experimental tool for semi-automated visualisation of graphs implemented in Pascal in algorithm-code complexes. The tool is based on the state-mapping paradigm, which does not require good understanding of the program being analyzed. This is the main difference from other graph visualisation tools created for educational purposes. Here we refer to the theoretical result, because the tool should to be updated with C/C++ languages in order to be applicable in LitIO.

We experimented using the tool for the analysis of algorithm-code complexes and determined that the tool fastens the process of analysis of algorithm-code complex thus allowing to introduce additional evaluation criteria which require deeper analysis of the algorithm-code complex.

5 Evaluation in Terms of the Existing Quality Standards

In Section 2.9 we showed that program development skills of the contestants play an important role in informatics contests. An algorithm-code complex is a program, i.e. software. We can argue that the whole submission can be treated as software, if verbal idea description corresponds the implementation.

We already decided, that we evaluate program development skills by evaluating the quality of the submission. There have been created various software quality standards, and the evaluation of the quality of the submission can be based on that.

However, the submission differs from the typical understanding of software. Algorithm-code complexes are designed and used only during the contests. The only customers and evaluators are jury members, and the complexes are not intended to be used outside the contests. Therefore it is not possible to apply software quality standards for evaluation of submission quality without taking into account the differences.

In this chapter we compare life cycles of a submission and software, identify the differences, look at the existing software quality standards, and analyse how the standards can be applied for evaluation in the informatics contests.

5.1 Comparing Life Cycles of a Submission and Software

A software life cycle is the sequence in which a project specifies, prototypes, designs, implements, tests, and maintains a piece of software (Kauffman et al., 2001). The standards for software life cycle processes are defined in ISO/IEC 12207 (ISO, 2008).

For comparison we use the waterfall life cycle model. Even though this model is not considered flexible, it works well with the projects that have well defined user interface and performance requirements (Wat, 2009; Royce, 1987). This is the case in informatics contests.

The comparison of life cycle of a submission and a software project are presented in Table 5.1, Figs. 5.1 and 5.2. The expected life cycle of a submission is rather similar to that of software projects. The main differences are: there is no return cycle from the *analysis* to *system requirements*, and there is no return cycle from *operation* to *testing*. Another important difference is that the operation period of a submission is very short, and no maintenance and support is required. However, the contestants do not necessarily follow good program developing practices, and there is no control over the process of program development in informatics contests. Therefore, in reality the life cycle of a submission and a software project may have more differences

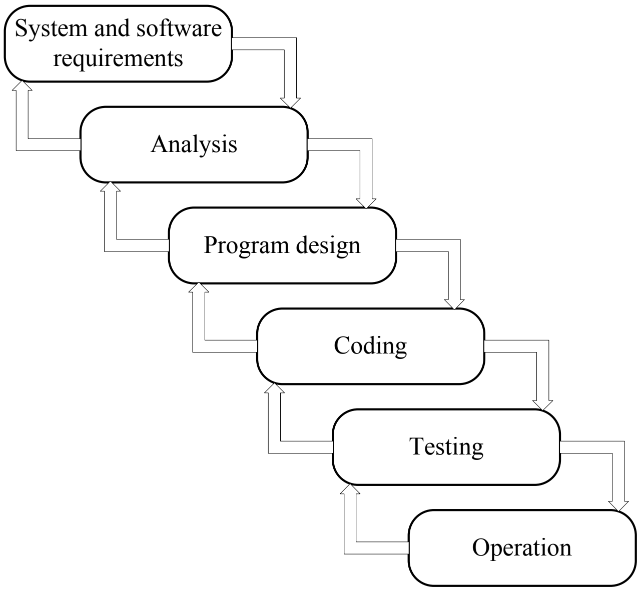

Figure 5.1: Waterfall life cycle model Based on (Royce, 1987).

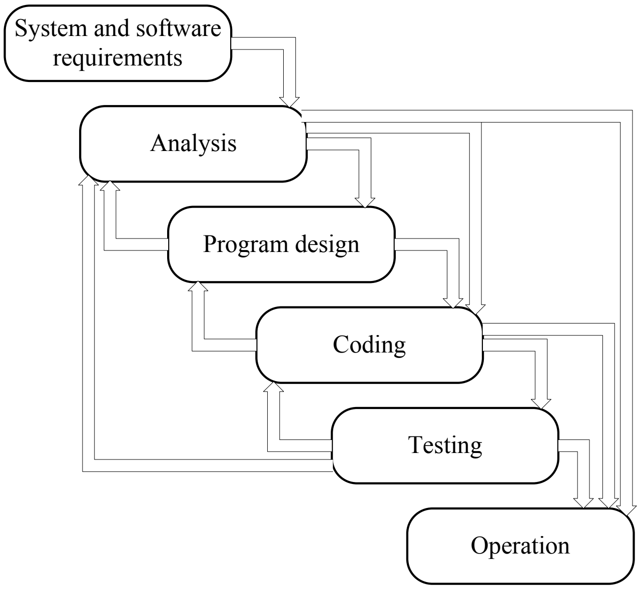

Figure 5.2: Waterfall life cycle model of a submission It is based on observations how it works in reality.

Software life cycle phase	**Description**	**Submission**
System requirements, Software requirements	Identify and document functional and scheduling requirements, required software features	Only customers (scientific committee) decide on system requirements. It is impossible to return to this stage once the next stage is started. The final outcome of this phase is a precise task specification together with all the necessary constraints.
Analysis	Methodically work through the details of each requirement, document algorithm	Verbal algorithm description should be the final outcome of this stage. However, in practice it happens that the algorithm analysis made by contestants is not exhaustive enough. Some contestants design an algorithm in their minds and postpone documenting it until the implementation and testing phases have been completed.
Program design	Use programming techniques to design software. Design specification is delivered at the end of this stage	Design specifications are not required as part of a submission. Note that some contestants perform this phase by presenting a program design in the form of creating required data structures, procedure headlines.
Coding	Implement the program as designed in earlier stages	Obligatory phase as the source code that compiles has to be submitted for evaluation.
Testing	Test the software	We observed in LitIO that often contestants (and even their teachers) do not know how to perform this phase. They either skip it or limit to testing with sample tests.
Operation	Provide maintenance and support of software	The operation phase is very short (big difference from the real life software projects). Just a very short period of time (few seconds or minutes) when the evaluation is performed. The operation phase is performed by the customers (i.e. scientific committee) without involving the developers (i.e. contestants). No real maintenance is expected to be performed during this period.

Table 5.1: Life cycle of a submission using the Waterfall model

(Fig. 5.2). We believe that this deviation does not help the contestants to achieve better results.

5.2 Evaluation of Quality of an Algorithm-Code Complex in Terms of the ISO-9126-1 Quality Standard

Many different software quality models were created. One of the first wide known models was created by McCall (McCall et al., 1977). The model foresaw three directions of the software quality: a possibility to change software, a possibility to adapt it to the new environments, and the performance characteristics. There was created a hierarchical tree that included factors (i.e. external views to software as it is seen by the user), quality criteria (as it is seen by the developer), and metrics for measuring and evaluating various aspects of criteria.

The most popular standard currently is ISO-9126 software quality standard (ISO, 2001). It was based on earlier works, among them, that of McCall. When analysing the evaluation of algorithm-code complexes, we refer to the ISO-9126 software quality standard. One of the reasons is, that there exists a huge variety of opinions among LitIO contestants, their coaches, and even among the jury members. Therefore, a commonly accepted and wide used standard adds some authority when motivating the evaluation scheme. This standard can be used as guidelines, while the quality model has to be adapted to the peculiarities of the informatics contest.

The ISO-9126 standard consists of four parts: quality model, external metrics, internal metrics, and quality in use metrics (ISO, 2010). The quality model is the first part of the standard (ISO-9126-1) and it is used to evaluate the quality of a program based on its source code. The rest three parts are more important for ensuring the software quality and for improving the process of developing software (Sof, 2007; ISO, 2010). Since contest organizers have no control of these parts and only focus only on the quality of the already submitted program, therefore we will refer to the ISO-9126-1 standard.

The software quality model ISO-9126-1 provides six software quality characteristics: functionality, reliability, usability, efficiency, maintainability, portability. Table 5.2 gives an overview of the current LitIO evaluation scheme in terms of these characteristics. The comments below about each characteristic complement the table.

Functionality. The main function of an algorithm-code complex is to solve the given task, i.e. to perform calculations and to output the required solution. It is measured by black-box testing. However, the question is raised in the scientific papers, whether this way of measuring functionality is reliable enough. It is common to name this characteristic *correctness* in the community of informatics contests.

Reliability. Input/output format is clearly defined in the task description, and the contestants are recommended not to check any unexpected situations. After performing calculations the program should stop its execution. The regulations

Characteristic	**Description**	**Is it included into the LitIO evaluation scheme?**
Functionality	It is concerned with evaluation how software performs its main task, i.e. whether all required functions are implemented	40%–70% of points.
Reliability	Defines the capability of the system to operate reliably under the defined conditions.	Some statistics is collected however it is not included into the evaluation scheme.
Usability	It is related with the ease of using the system (for the user)	Points are not assigned for this characteristic, however only the programs that satisfy interface requirements are accepted for evaluation.
Efficiency	It is related with the efficiency of software in terms of use of other resources.	30%-60% of points.
Maintainability	It is related with the ability to identify and fix a fault or modify software.	Up to 10% of points.
Portability	This characteristic refers to how well the software can adapt to the changes in its environment	Not included into evaluation scheme.

Table 5.2: Overview of the current LitIO evaluation scheme in terms of six software quality model ISO-9126-1 characteristics

of the contests do not indicate how the program should operate, if it does not find a solution. Typically, in such situations, the program either loops, or outputs an incorrect solution or an exception occurs. Even though CMS provides data about the success of program execution with each test, this is not measured separately and is equated to the incorrect result (i.e., absence of functional correctness). This characteristic is not emphasised in informatics contests, and not included into the evaluation scheme in order to leave more emphasis to problem solving.

Usability. All the requirements to the user's interface (i.e. input/output format) are strictly defined in the task description. CMS assists in checking whether the submitted program meets these requirements.

Efficiency. Task description defines the exact time and memory limits that should not be exceeded by the program during its execution with a test. Memory constraints are imposed on the overall memory usage including the executable code size, a stack, a heap, etc. If the constraints are exceed when executing the program with a concrete test, then this test is recorded as failed.

Some tests are designed to check efficiency. Data size and complexity are gradually increasing in such tests. Thus algorithm-code complexes are grouped into different categories according to time and memory efficiency.

The efficiency of use of time and memory resources is not related to the evaluation scheme directly, i.e. the program which finished its execution in a shorter period of time or used less memory will not score more points for a specific test.

Maintainability. This characteristic is evaluated by assigning points for programming style. Guidelines for the programming style are prepared to the contestants. The guidelines are quite general and do not enforce any specific style, however, emphasise the most important style requirements. The programming style is evaluated holistically using an ordinal scale. This means that the source code is not evaluated in terms of each criteria separately.

In order to avoid assigning points for the programming style to programs that do not solve the task, the style is evaluated only if the algorithm-code complex scores no less than 50% of points for functionality.

Portability This characteristic is evaluated indirectly[1], and when evaluating programming style. Attachment to a particular operating system is treated negatively.

From Table 5.2 we see that three out of six characteristics are included into the evaluation scheme in the current LitIO evaluation practice. They are functionality, efficiency and maintainability. The other three characteristics, reliability, usability, and portability are not evaluated.

LitIO jury emphasises that the contestants should focus on designing the algorithm and program development. Technical knowledge should not play the main role in the contest. Therefore the reliability is eliminated from the evaluation scheme. Besides, the reliability is important for software with a long life cycle. However the life cycle and the operation phase of a submission are short.

Submission interface is clearly defined in the contest rules and the task description. CMS assists in ensuring that the submission satisfies the interface requirements. Therefore this requirement is considered as purely technical, and not requiring creativity or problem solving skills. Eliminating that from the evaluation scheme is justifiable.

Portability has many different aspects. Many of them are related to technical aspects, e.g. different versions of compilers, use of libraries, etc. The above mentioned

[1]In LitIO it is common that the contestants use their school labs equipped with Windows OS, while CMS operates on Linux server. Therefore the contestants who write OS specific programs have more difficulties while they achieve that their program operates in both environments.

attitude that the knowledge of the compilers and other technical aspects should not play the main role, is a motive to exclude portability from the evaluation scheme.

Next, we discuss evaluation of the functionality, the efficiency, and the maintainability of an algorithm-code complex.

5.3 Evaluating Functionality and Efficiency: Analysis How Much Testing Results Conform with Expected Scores

Functionality is the first and most important characteristic of the software quality model ISO-9126-1. Other characteristics have no significance if the software is not functional. Therefore if an error is detected, it is expected that software developers will fix it. However, this situation is very different in the contests. If a test shows that the program is not functional, then in IOI type contests (including LitIO) it is necessary to assign some score instead of returning it for fixing the error.

The jury sets the evaluation goals. They expect that the evaluation results correspond the evaluation goals. For example, there is a goal that incorrect programs do not pass all the tests and thus do not score full points for tests. However, it has been proved theoretically that testing cannot be used to prove program correctness. Therefore it is important to investigate how the testing works in reality, and the extent to which this (detecting all incorrect solutions) and other goals are achieved.

The main reason that initiated this part of research, was the question how reliable the black-box evaluation is in the informatics contests. Are the black-box testing goals achieved at a satisfactory level?

The evaluation goals are discussed and fixed for each task. In general, it is not possible to establish one-to-one correspondence between the points, awarded by human and by an automated graders. Therefore, acceptable ranges of points are introduced. The jury decides what is considered as an “acceptable” range of points.

The tests and the scoring schemes are designed to support the evaluation goals set for a concrete task. However, there are some general guidelines in LitIO that can be adjusted for a specific task. The guidelines state that, for example, conceptually incorrect algorithm-code complexes score no more than 30% of points, correct but inefficient algorithm-code complexes score from 30% to 60% of points, correct and efficient algorithm-code complexes score from 60% to 100% of points thus ensuring proper discrimination among the complexes of different efficiency.

In order to investigate to what extent the limitations of black-box testing distort the expected evaluation results in LitIO, it is required to evaluate a set of submissions manually, and compare to the black-box testing results. It is time consuming to perform that, because the contestants do not provide proofs of algorithm correctness and may not follow a good programming style. This might be the reason why we found only one scientific paper (master thesis) comparing black-box testing results to the results of other types of evaluation (Leeuwen, 2005; Verhoeff, 2006). While other sources only refer to separate cases, where the black-box testing results did not (or may haven't) match the expectations of the jury (Forišek, 2006; Opmanis, 2006).

This section compares the results of human evaluation and the black-box scoring of 160 programs designed during the final round of LitIO'2008. All the algorithm-code complexes were evaluated automatically following the typical black-box grading procedure, and additionally evaluated manually by analysing the source code of each algorithm-code complex. The scores of black-box testing in this section are presented in the 100 point scale in order to be more comparable to the expected scores (expressed in percentage), and to the research conducted by van Leeuwen.

5.3.1 Brief Introduction to the *Nescafé Algebra* Task

In three exam sessions of the final round of LitIO'2008 14 tasks were presented both for junior and senior divisions. One of them, *Nescafé algebra*, was chosen for research. The choice was motivated by the sample size (we were looking for a task with more than 100 submissions), and the task should be classified as a "typical" LitIO contest task. A brief task statement is as follows:

"Nescafé" offers for sale various mixes, like «3 in 1» or «2 in 1». What happens if we mix «3 in 1» with «2 in 1»? Shall we get a super-mix «5 in 2»?

Task. *Let us define that if we mix «x_1 in y_1» with «x_2 in y_2» we get a super-mix «$x_1 + x_2$ in $y_1 + y_2$». Given a sequence of N coffee mixes «a_1 in 1», «a_2 in 1», $\cdots$, «a_N in 1». Which of the given mixes (each can be used only once) have to be chosen to make a super-mix «b in c».*

Constraints. $1 \le N \le 100$; $1 \le c \le N$; $1 \le b \le 10000$; $1 \le a_i \le 1000$. *Data will be such that in tests worth 50% of total points $N \le 25$.*

A full task formulation can be found in Appendix A.1.

A trivial, correct, but not efficient solution would be exhaustive search. There are at most $\frac{N!}{(N-c)!}$ different combinations to be analysed. However, various optimizations can be considered:

- Sorting mixes in a non-increasing order.
- Analysing only different sets of mixes (which brings down the number of possible combinations to $\frac{N!}{c!(N-c)!}$).
- Stopping the search immediately after the first good solution has been found, etc.

The expected solution involves dynamic programming. A two-dimensional table <b in c> is created. The value of a table cell <x in y> is equal to the index of the last mix used in making the super-mix <x in y>.

If $c = 0$, we can get only a <0 in 0> = 0 super-mix. Assume that all the super-mixes that can be obtained using exactly k mixes, are already calculated. In order to get all the super-mixes obtained using exactly $k+1$ mix, we need to consider each super-mix $<x$ in $k>$, and add to it each $<a_i$ in $1>$ mix (constraints: $1 \leq i \leq N$; a_i was not used for getting that particular $<x$ in $k>$). After adding an extra mix, we get a new mix $<(x + a_i)$ in $(k + 1)>$. It is possible that the super-mix $<(x + a_i)$ in $(k + 1)>$ was already obtained. In that case the value of $<(x + a_i)$ in $(k + 1)>$ is not updated. The time complexity of this algorithm is $O(cbN)$. This complexity is sufficient to solve the given task within the given constraints.

Memory constraints. Memory needed to store the list of mixes used to make each super-mix would be $O(bcN)$. However, in order to calculate the value of $<k+1$ in $\cdots>$, only the values of $<k$ in $\cdots>$ are referred to. Therefore it is enough to store the list of mixes only for all the values of k. Then memory consumption goes down to $O(bN)$.

5.3.2 Quantitative Overview of Submissions to *Nescafé Algebra* Task

176 students participated in the senior division of the on-line contest of the final round of LitIO'2008. 160 of them attempted to solve the task, however 6 of them submitted only verbal descriptions of algorithms (which have been evaluated even if the accompanying program was not presented) and 1 presented the program which did not compile. In total there were 153 programs suitable for automated evaluation. The data are presented in Table 5.3.

		No of cases	Percentage
Final round participants		176	100%
Presented a solution		160	91%
	not suitable for automated evaluation	7	4%
	suitable for automated evaluation	153	87%

Table 5.3: Suitability of the submitted solutions for automated evaluation

All the 153 algorithm-code complexes eligible for automated evaluation were thoroughly analysed. According to the algorithms designed by the contestants, the submissions were sorted into the categories presented in Table 5.4. A verbal algorithm description was the required part of submission and it was used as an additional reference to determine what algorithm was intended to be designed by a contestant. We also analysed the quality of implementation, but did not take into account in this categorisation. It means, that e.g. the dynamic programming solution which contains some critical implementation errors and therefore would receive low score when evaluating both manually and automatically, in Table 5.4 still falls into the category of correct algorithms (strategies).

The choice of the algorithm is expected to define the upper bound for the points, but not the lower one. The data show that about 35% of submissions implemented or attempted to implement incorrect algorithms, while about 50% of contestants implemented or intended to implement correct algorithms.

Solution strategy		**No of cases**	**Percentage**	**Cumulative percentage**
Incorrect strategy	Incomplete solutions (no clear algorithm)	21	13.7%	13.7%
	Random strategy	11	7.2%	20.9%
	Heuristic strategy	4	2.6%	23.5%
	Other	18	11.8%	35.3%
Partially correct	Analysis of separate cases	22	14.4%	49.7%
Correct, but inefficient	Exhaustive search	58	37.9%	87.6%
Correct and efficient	Dynamic programming	19	12.4%	100%

Table 5.4: Classification of solution strategies applied by the contestants

A set of 10 test cases was designed with in total 43 test runs. The tests were designed with the intention that the exhaustive search solution with some optimization would score around 50% of points. We tested by performing the black-box grading procedure presented in Section 2.10.2. During the contest a *partial scoring* scheme was applied. During this research *all-or-nothing batch scoring* was also applied to compare both schemes.

The distribution of points, awarded using both scoring schemes is presented in Fig. 5.3. The mean value of the scores obtained by applying partial scoring is 21.5, the median is 10, and the standard deviation is 25.3.

It should be noted that over 50 programs ($\approx 33\%$) scored zero points and that this task is one of the two presented in this exam session. The participants of the final on-line round of the national contest mainly studied the programming and algorithms themselves or in after-curricula classes with no real scientific support from the teachers (Dagienė and Skūpienė, 2007). At this level such a high number of zero scores might have a negative impact to their motivation.

In IOI *a relative task difficulty measure* is used. It is based on the percentage of contestants who solved the task "fully", i.e. scored 90% or more. There are three main difficulty levels: *easy* ($>$ 40% solved the task "fully"), *medium* (between 40% and 10% solved the task "fully") and *hard* ($<$ 10% solved the task "fully") (Verhoeff, 2009). According to this measure the task is hard, because only 6 programs (4%)

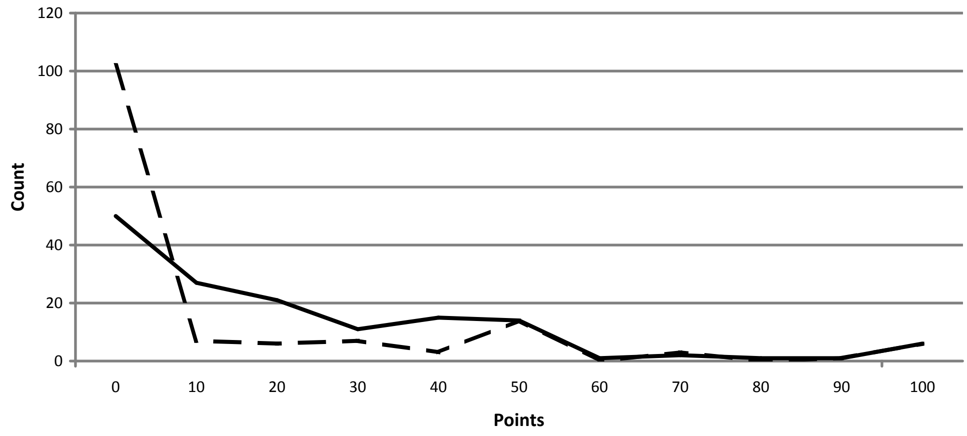

Figure 5.3: Comparison of partial scoring (straight line) and all-or-nothing batch scoring (dotted line) results

solved the task “fully”. According to another source (Nec, 2007), the task difficulty measure is calculated as:

$$\text{Task\_difficulty} = \frac{\text{The sum of possible scores of all contestants}}{\text{The sum of maximal possible scores of all contestants}} \tag{5.1}$$

The task is considered *easy* if the difficulty is $\geq 70\%$, *average* if it is around 50%, and *hard* if it is $\leq 30\%$. The difficulty of *Nescafé algebra* is equal to 21% and thus the task is considered *difficult* under this difficulty measure as well.

One of the important evaluation goals in informatics contests is to discriminate the contestants in order to deliver ranking and determine the best ones. *Item discrimination* (Ite, 2010) or *IDis* is a measure used to evaluate that. To calculate the item discrimination, the contestants are ranked according to their scores. Then the top 27% and the lowest 27% in terms of the total score are selected, and the percentage of correct answers in both items calculated. The formula is:

$$\text{IDis} = (\text{Upper Group\%Correct}) - (\text{Lower Group\%Correct}) \tag{5.2}$$

A negative IDis is considered unacceptable, from 0% to 24% is usually acceptable, from 25% to 39% is a good item discrimination and from 40% to 100% make an excellent item discrimination.

In this task $IDis = 0.56$ which is considered as an excellent overall discrimination. We also calculated the discrimination of the top 50 submissions (good discrimination of the top contestants is also very important): $IDis = 0.45$ and it is still considered as an excellent discrimination. However, this task does not discriminate the bottom part at all. This is not acceptable from the educational point of view and is in contradiction with the contest and evaluation goals (Kemkes et al., 2006).

Another interesting characteristic is relationship between the choice of the algorithm and the points obtained by partial scoring. This relationship is presented in Fig. 5.4. The most questionable points are that of heuristic approach, since this approach is considered to be incorrect. The results of each of these categories will be analysed further.

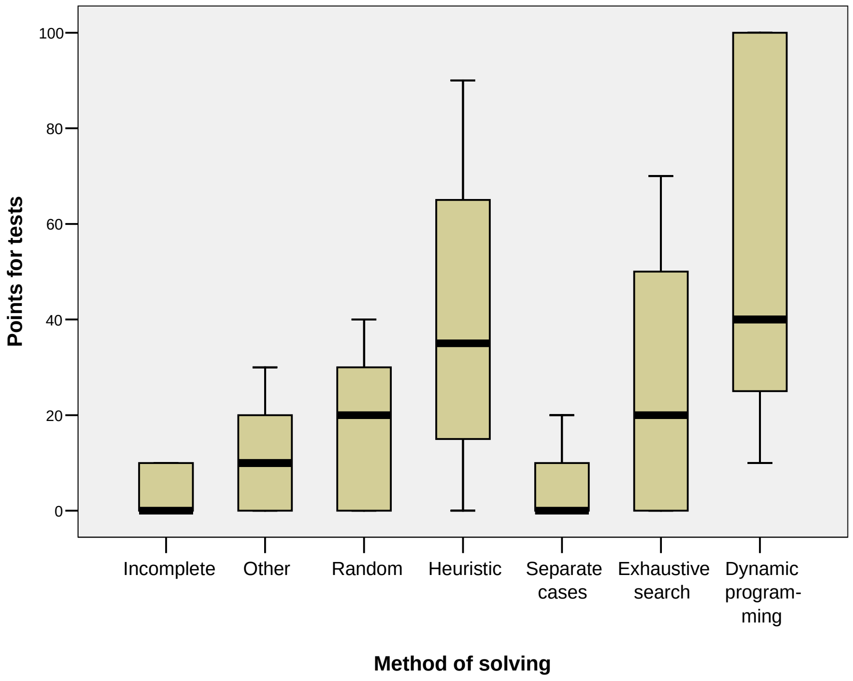

Figure 5.4: Box-plot of points assigned by partial scoring

5.3.3 Analysis of Algorithm-Code Complexes with Incorrect Strategies: Incomplete Algorithm-Code Complexes

There were submitted 21 (13.7%) incomplete programs to the *Nescafé Algebra* task. Basically these are programs which read input data, some of them contain just a few lines of program (too few to decide what kind of algorithm it was supposed to be) and output either zero, or the value of (uninitialized) data structure for storing the solution. Under partial scoring 14 submissions received zero scores, and 7 submissions received 10 points. The latter submissions were those which in all cases output zeroes. In this task zero output represents the case when there is no solution. It can be concluded that the points during the contest were given against the expectations of the jury, because only those who correctly determined that the solution to the given input does not exist (i.e. there is no way that number b can be obtained by adding c numbers) should have been awarded points for those tests.

Under the all-or-nothing batch scoring scheme, all the incomplete algorithm-code complexes did not get any points.

The contestants had to provide a verbal description of their algorithm as part of a submission and its was evaluated by human evaluators. The mean score for a verbal description accompanying incomplete algorithm-code complexes was 3.6 points out of 20 (*min* value 0,*max* value 7), and the median value was 4. This indicates that the authors of the submissions did not have a good idea how to solve the task. The question whether such incomplete algorithm-code complexes should score some points (e.g., for being able to read data) or not is the topic of another discussion. Currently the jury expects that such algorithm-code complexes score no points.

It can be concluded that a partial scoring scheme assigned a few points to some incomplete algorithm-code complexes, while the all-or-nothing batch scoring scheme did not give any points for the incomplete algorithm-code complexes, i.e. it fully corresponded the evaluation expectations.

5.3.4 Analysis of Algorithm-Code Complexes with Incorrect Strategies: Random Strategies

There were 11 (7.2%) contestants which applied a random strategy to solve the task. In order to make a super-mix «b in c», they randomly selected c different mixes out of N available ones and calculated their sum. If the sum was equal to b, the solution was found, if not – another random c mixes were chosen. Even though at first sight a random approach does not seem to be very promising, the highest score was 40% (Fig. 5.5).

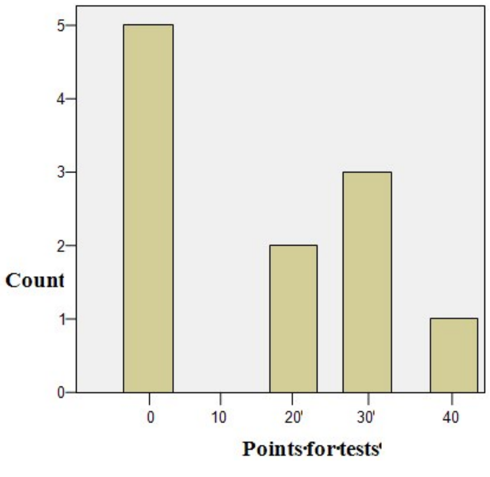

Figure 5.5: Histogram of points assigned by partial scoring to algorithm-code complexes with a random strategy

There was one notable difference among the submissions, i.e. the number of times a random selection was repeated. Some algorithms were going on until they found a solution, some stopped after 50 times, some – after$10,000$ times, some stopped after N^3 times. The programs which allowed the random search to be repeated more times scored more points. All the points were scored for tests designed to check algorithm correctness (i.e. with $N \leq 25$).

In LitIO the acceptable range of points for random strategies is from 0 to 10% of points. Therefore the partial scoring scheme for *Nescafé Algebra* was too generous to such submissions.

After retesting by means of the combined scoring scheme, only 3 submissions out of 11 were awarded 10 (once) and 30 (twice) points. The total sum of points scored by all the algorithm-code complexes with a random strategy dropped from 170 to 70. Nevertheless, there were left two submissions the scores of which remained unacceptable. The verbal descriptions of solution strategies also received very low grades (mean 3.3 and median 3 out of 20).

None of the two scoring schemes completely correspond to the expected scoring ranges. More than a half of submissions received too high score using partial scoring. Only 2 out of 11 submissions received too many points using all-or-nothing batch scoring.

5.3.5 Analysis of Algorithm-Code Complexes with Incorrect Strategies: Heuristic Strategies

There exist various definitions of a heuristic algorithm. It can be described as an algorithm that always provides some kind of solution to the problem, though it may fail to provide an optimal solution (Brassard and Bratley, 1996). It gives up finding an optimal solution for the improvement in run-time. Many efficient, but incorrect strategies (like selecting random coffee mixes) fall under this definition. However, in this research we distinguished heuristic strategies with optimization, i.e. the ones that find a solution close to optimal with high probability. For example, randomly selecting c mixes does not guarantee that their sum will be close to b. While the strategy which sorts the mixes in a decreasing order and then takes the first ones which do not cause the sum to exceed b, it is expected to produce a solution close to optimal.

The tasks in LitIO (as well as in IOI) are designed so that that each of them has an elegant and efficient correct algorithm (or several ones). Heuristic strategies are usually much shorter and simpler to implement than the correct ones, their success also depends a lot on test designers. Therefore heuristic algorithms are not considered as acceptable ones in LitIO. Some efforts were put to decrease the possibility that such strategies score more points than the correct, but less efficient ones (Forišek, 2006). In LitIO it is expected that algorithm-code complexes with heuristic solutions implemented score no more than 30% of points.

Only 4 heuristic algorithm-code complexes were submitted. They scored 0, 30, 40, and 90 points under the partial scoring scheme. The mean grade for the verbal description accompanying heuristic algorithm-code complexes is 3.3 points out of 20 (min value 0, max value 5) 0 and the median value is 4.

Three of them applied the greedy strategy as it was described above. The fourth algorithm-code complex introduced the following heuristics: select any c mixes and then repeatedly look for pairs of mixes where one is already among the c selected mixes, the other is not, however swapping them yields in a more optimal solution. Then the two mixes are swapped and the search is performed again. The all-or-nothing batch scoring scheme assigned 0 points to the first three algorithm-code complexes and 70 points to the fourth one. This score falls out of the expected range and is the most questionable score among all the analysed submissions.

That reveals one of the largest problems of automated grading: heuristic algorithms with optimization might not be identified by the grader (i.e. by the designers of the test cases) and the assigned score might be much higher than expected by the task designers.

5.3.6 Analysis of Algorithm-Code Complexes with Incorrect Strategies: Other Strategies

There were 18 (11.8%) algorithm-code complexes that fell under this category. The majority of them (14) considered as possible solutions only sequences of consecutive mixes. For example, if the given mixes are 1 5 4 3, and $c = 3$, they only considered 1 5 3 and 5 4 3 combinations, but not 1 4 3. Possible reasons for that might be either incorrect understanding of the problem, or inability to encode a more complicated search. The remaining incorrect solutions included one algorithm-code complex that solved completely different task, and three algorithm-code complexes that applied the divide and conquer strategy. Fig. 5.6 shows the distribution of partial scoring points obtained by such algorithm-code complexes.

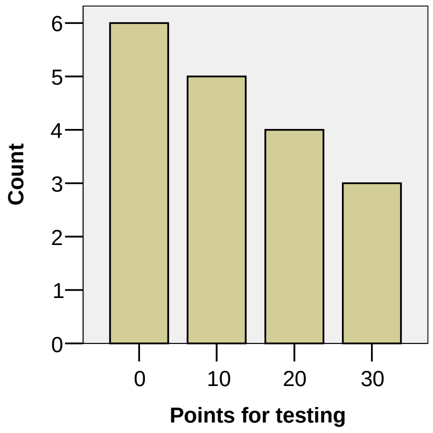

Figure 5.6: Histogram of points assigned by partial scoring to algorithm-code complexes with other incorrect strategies

The points were mainly awarded for the smallest and easiest tests, where the analysis of consecutive mixes yields a solution, and/or for a small *no-solution* test.

Only one algorithm-code complex out of 18 got points for one non-trivial test. After reevaluating by the all-or-nothing batch scoring scheme, only two algorithm-code complexes out of 18, were awarded 10 and 20 points. That falls within the acceptable range (0 to 20% of points).

The mean grade for the verbal description accompanying these algorithm-code complexes is 2.2 points of 20 (min value 0, max value 8), and the median value is 2.

It can be concluded that partial scoring assigned more points to some algorithm-code complexes than it was expected, but the all-or-nothing batch scoring scheme was applicable for evaluating such algorithm-code complexes.

5.3.7 Analysis of Partial Solutions

By partial solutions we mean algorithms that correctly solve a partial task (e.g. only when $c \leq 4$) and do not intend to solve the whole problem. The general attitude towards partial solutions among members of the scientific committee of LitIO is the following: partial solutions, even with small constraints, should score some amount of points.

In this case, test designers thought of and stated in the task two benchmarks: $N \leq 100$ (dynamic programming, full points) and $N \leq 25$ (an exhaustive search, 50% of points). However, even smaller benchmarks should have been considered when combining tests to batches.

There were 22 (14%) such solutions that were assigned from 0 to 20 points during the formal evaluation in LitIO using partial scoring and this fell within the acceptable range taking into account implementation errors. The all-or-nothing batch scoring assigned zero points to all the algorithm-code complexes. The mean grade of the verbal description accompanying these solutions is 2.7 points out of 20 (min value 0, max value 8), and the median value is 3.

In this case, partial scoring better corresponded to the expected scores. However, if smaller benchmarks were introduced, both scoring schemes could correspond to the expectations of evaluating partial solutions.

5.3.8 Analysis of Correct Solutions: Exhaustive Search

An exhaustive search strategy is considered to be correct but inefficient in informatics contests. Algorithm-code complexes that implement it, score approximately up to 60% of points.This is a typical problem solving strategy applied by the contestants who have less experience in problem solving and cannot think of a more efficient algorithm.

An exhaustive strategy was implemented in 58 algorithm-code complexes. Points assigned by the partial scoring scheme to them are presented in Fig.5.7.

The tests were designed with the intention, that the well implemented solutions with optimization score around 50% of points, and no more than 60% of points. However, 2 solutions scored 70% of points using partial scoring scheme. The two high scores were incidental and were influenced by the direction in which the program of the contestant was traversing the search tree. This could be easily corrected by applying all-or-nothing batch scoring. In all-or-nothing batch scoring tests of the same difficulty but with different search trees are combined into the same test case.

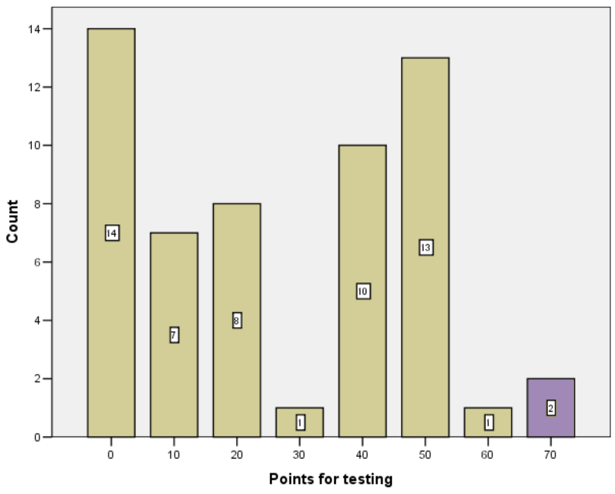

Figure 5.7: Histogram of points assigned by partial scoring to algorithm-code complexes that implemented exhaustive search

The algorithm-code complexes, which could score incidental points for one of the tests, do not score the points for the test case.

The exhaustive search is slow in its nature. Therefore all-or-nothing IOI scoring works as a good upper bound in the scoring scheme. Practically there is no possibility for such a solution to earn more points than it is expected to.

However there is another danger: less experienced students make minor implementation errors and the automated scoring scheme may give fewer points than expected. Among the analysed solutions 10 such cases were discovered. Partial scoring gave no points to five of them. Due to minor mistakes the contestants lost from 20 to 40 points per algorithm-code complex. Even though this problem could be solved through appeals, at present this is not practiced. Because it is very hard to put clear boundaries when the error is considered as minor and when it becomes the major issue. Providing more feedback during the contest might be another option how to solve this issue.

5.3.9 Analysis of Correct Solutions: Dynamic Programming

There were 19 contestants (12%) which selected dynamic programming as their solution strategy. Their algorithm-code complexes scored from 10 to 100 points depending upon the quality of implementation. After reevaluation with all-or-nothing batch scoring scheme, the scores slightly went down, however the total score precision remained at the acceptable level.

One case was observed where due to a minor error (swapped indexes), the contestant lost 80% of points. This mistake had serious consequences to the contestant:

he lost the possibility to compete for medals. It was practically impossible to detect this kind of mistake automatically. Providing more feedback during the contest might be helpful in such cases.

The mean grade for the verbal description accompanying these solutions is 11.6 points (min value 0, max value 20) of 20 and the median value is 12.

5.3.10 Comparison of Black-Box Scoring Results with the Expected Scores of Submissions to *Nescafé Algebra*

We analysed 153 algorithm-code complexes designed during the final round of LitIO held in 2008. All the algorithm-code complexes were tested using black-box testing and graded using two scoring schemes: partial scoring (used during the real contest) and all-or-nothing batch scoring (currently used in IOI). The ranges of expected scores for different types of solutions were determined by the jury in advance.

The scores obtained by the two scoring schemes differed from the expected scores in two ways: algorithm-code complexes were assigned more points than they ought to or they were assigned fewer than the expected ranges of points. The data about the two types of score deviations are presented in Table 5.5.

			No of unjustified scores			
			Partial scoring		**All-or-nothing batch scoring**	
Solution strategy		**No of cases**	**Score too high**	**Score too low**	**Score too high**	**Score too low**
Incorrect strategy	Incomplete solutions (no clear algorithm)	21	7	–	–	–
	Random strategy	11	6	–	2	–
	Heuristic strategy	4	2	–	1	–
	Other	18	3	–	–	–
Partially correct	Analysis of separate cases	22	–	–	–	10*
Correct, but inefficient	Exhaustive search	58	2	10	–	10
Correct and efficient	Dynamic programming	19	–	1	–	1
	TOTAL	153	20	11	3	21

Table 5.5: Statistics of unjustified scores in the analysed submissions

It is difficult to detect the first type of deviation (assigning too many points) during the contest. When applying partial scoring, 20 algorithm-code complexes (13%) received higher than the expected score ranges. Out of those 7 (4.5%) algorithm-code complexes obtained too high score by 20 or more points. Only 3 algorithm-code complexes (1.9%) got too many points with all-or-nothing batch scoring. However, all the 3 algorithm-code complexes got extra 20 or more points.

Too low score can be detected easier. Because the contestants write appeals and complain.[1] We found 11 algorithm-code complexes (7.2%) which obtained too few points from partial scoring. One of them lost 80% of points due to a minor error. Even though all-or-nothing batch scoring in its nature tends to lower the scores, it did not have a strong effect on those algorithm-code complexes. Because their scores were low enough anyway.

In total 20.2% (partial scoring) and 9.1% (all-or-nothing batch scoring) could not be justified. 8.4% (partial scoring) and 5.8% (all or nothing batch scoring) of scores differed from the expected scores by 20 points or more. Therefore, for this task, all-or-nothing batch scoring better corresponds to the evaluation expectations. However, this task was too hard. It would have been more appropriate to give it in the last exam session of the final round, where only the top 30 finalists compete, and not in the first session, as it was done.

At this point it is interesting to compare the research results to that of (Leeuwen, 2005). His research concerned the tasks *Median* (from IOI'2000) and *Phidias* (from IOI'2004). He analysed 504 submissions, which were graded using the partial scoring scheme during the contests. For reevaluation he chose a very strict scoring scheme. The scheme assigned zero score to all the incorrect algorithm-code complexes. Following such a grading policy, over 30% of task *Median* scores and over 50% of task *Phidias* scores could not be properly justified.

A detailed analysis of task *Median* was presented in (Horvath and Verhoeff, 2002) after IOI'2002. This means that the problem and various algorithmic approaches were investigated scientifically. Nevertheless among the submissions additional interesting algorithms were discovered during the reevaluation.

After reevaluating it was suggested that some algorithm-code complexes that scored 60 points for lack of efficiency, could be qualified for full points. Because the constraints were too strict for a particular combination of the programming language and solution technique. There were found 10 incorrect algorithm-code complexes of task Median and 57 incorrect algorithm-code complexes of task *Phidias* not identified as such during official grading.

Both investigations confirm that black-box testing does not fully correspond to score expectations. The deviation of partial scoring from the expected score ranges is much higher than that of all-or-nothing batch scoring.

On the one hand, some wrong solutions are not discovered. On the other hand, there might be interesting algorithmic approaches designed by the contestants and

[1]These low scores appeared due to incomplete taxonomy and subsequently inadequate choice of tests. Therefore this number is not taken into account when calculating the percentage of the +deviation.

not thought of in advance by task designers. To improve the situation we suggest to provide more feedback for correctness tests and to use a motivated evaluation scheme not limited to black-box testing.

5.4 Evaluating Maintainability (Programming Style)

5.4.1 Introduction

In terms of qualitative characteristics of software following the standards ISO-9126, the programming style can be related to software maintainability. *"Maintainability can be described as the ease at which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"* (IEE, 1990). ISO-9126 standards describe that maintainability is affected by source code readability, complexity and modularisation (ISO, 2010). However the definition does not guide to any hints about measuring maintainability.

We found several approaches towards measuring software maintainability and no common approach. (Land, 2002) introduces two understandings of software maintainability: either it is very informal, or it is a measure directly derived from a source code. In case it is considered as a formal measure, it is expressed as a function of directly measurable attributes: $M = f(T_1, T_2, \cdots, T_k)$. At the same time (Land, 2002) notices that formula that describe maintainability are of rather limited use. There are serious difficulties in measuring separate attributes, weighting them against one another and combining them in a function.

This reveals that there is no standard commonly accepted way of measuring software maintainability. This fact gives more freedom for constructing the concept of quality of the programming style in informatics contests. In informatics contests the term *programming style* is used to represent what is understood as software characteristic maintainability. However, in the contests it has a more limited meaning.

The programming style has been evaluated in LitIO since the first LitIO. A detailed description of this grading practice can be found in Subsection 2.10.3. We found no written materials of such extensive grading of the programming style in other informatics contests. The experience of evaluating programming style in programming courses was discussed in Subsection 2.11.4.

We came across many concerns about negative impacts of ignoring the programming style in the evaluation schemes. The discussion on whether the programming style should be included into the evaluation scheme of the informatics contest is active. *"It is possible to win the contest without writing a piece of code that meets even the guidelines of structured programming, let alone a more modern philosophy"* (Andrianoff and Hunkins, 2004). The absence of an enforcement mechanism makes it difficult to promote good programming practices.

The positive part of evaluating the programming style is as follows: program simplicity and clarity and elegance (which are essential feature of good programs at all levels of computing) will be taken into account and two equally correct and efficient programs but only with a different quality of the programming style will not be equalized. Therefore the grading will be fairer.

There are many other reasons why there might occur a need to analyze the program source by others than the author of the program during the contest. The most obvious one - analyzing appeals. It is known that the programming style has much influence on its comprehensibility (Mohan and Gold, 2004; Oman and Cook, 1990).

One of the reasons why the implemented algorithm is not analyzed during evaluation in LitIO is time constraints. It would take too much time to decide which algorithm is implemented, is it correct and how well it is implemented by simply analyzing the program source. That becomes especially difficult if the programming style is far from perfect. However, this is not true about the programming style. It does not take too much time to decide whether the program is written in an understandable manner or not.

There are two other concerns about the inclusion of the programming style into the grading scheme. One of them is reappearance of the subjectivity factor in informatics contests, the other one – relating grading of the programming style to program (or algorithm) correctness.

We have already overview the research in the area of automated evaluation of programming style. We arrived at the conclusion that the experience of automated evaluation can be potentially transferred to informatics contests, however it requires a separate study and cannot be conducted within this thesis. Therefore we will analyse the applied currently in LitIO holistic approach when the evaluation is performed by human evaluators. We must add that the holistic approach also has its own advantages. With automated evaluation there arises a danger that the contestants might start treating the programming style evaluation criteria as a collection of language specific rules. (Oman and Cook, 1990) remind that the students should be taught the principles of programming style according to the taxonomy rather than the sets of separate rules.

5.4.2 The Subjectivity Factor in Evaluating Programming Style

One of the negative features of including evaluation of programming style into evaluation scheme is that it brings back the subjectivity factor to the contest. The same program might look clearer to one jury member than to another despite the existence of formal criteria for evaluating programming style. Any personal decision in grading might be a cause to appeal. An example could be IOI where there is no human grading and almost no appeals have been in the last few years.

However, there also exist different opinions: *"the desire for automation as carried out in the past has blinded the IOI community: the real work of the contestants remained invisible"* (Verhoeff, 2006). The subjectivity factor should not be feared, for subjective decisions may lead to more justifiable evaluation than impartial black-box grading. Besides, in other science olympiads, for example in International Mathematics Olympiads (IMO), grading is not automated (Verhoeff, 2002) and therefore includes subjective decisions. However these contests are not considered too subjective just because of the absence of automated grading.

In LitIO the programming style has been part of a grading scheme since 1989, the contestants could be awarded up to 10 points (out of 100) for it. Despite the existence of formal criteria for evaluation, the practice shows that different jury members might treat the same program in a different way. Usually this difference is 1-2 points and only in very few worst cases it is 4-5 points. In the case the difference is larger than 2 points, several more jury members are involved into evaluation and the score is reconsidered.

The answer to the subjectivity problem can be obtained by answering the following question: whether it is more justifiable to treat programs with a bad and good programming style as of equal quality (if their performance is the same) or to make a distinction between them but with 1-2 point error possibility.

5.4.3 Relating Evaluation of Programming Style to Program Correctness

Another important issue is how to relate the evaluation of programming style to functional correctness. It is obvious, that a certain relationship should be invented.

For example, a three line program which prints a random number (therefore is neat, clear and simple) and does not solve the task at all should not be awarded points for program clarity.[1] In LitIO there is an accepted rule, that programming style is evaluated only if the algorithm-code complex scores more than 50% of points for tests. This is not quite justifiable because black-box grading covers only functional correctness. There were cases observed where a minor implementation mistake resulted in low scores (Forišek, 2006) and it would not be fair to lose 10 more points.

There might be another approach to this issue: to look through all the programs, submitted by the contestants, and to evaluate all those that try to solve the task. If the program does not try to solve the task (e.g. outputs random numbers or just solves the sample tests), it can be easily identified by the evaluator by just looking at the text of the program. If the program does not try to solve the task, but is obfuscated on purpose (even such cases were observed by (Verhoeff, 2006)) or is so messy, that it is impossible to determine whether it tries to solve or not, then the clarity and understandability of such a program would not be worth any points at all.

To sum it up, certain conditions must be set to algorithm-code complexes which have to be satisfied in order to evaluate programming style. The conditions can either be related to functional correctness or to the efforts of implementing a solution.

5.4.4 Case Study: Analysis of Evaluating Programming Style of one LitIO'2006 Task

An investigation of relationship between programming style and an algorithm-code complex correctness was performed by (Grigas, 1995). Programming style of nearly 700 hundred algorithm-code complexes developed by contestants from various countries was analysed. Correlations between functional performance of algorithm-code complex and programming style were calculated in various ways. However, the dependencies received were medium.

[1] Actually, it should not be awarded any points at all. It is just that the black-box testing does not always guarantee that.

We repeated the research of (Grigas, 1995) with algorithm-code complexes designed in LitIO. The task, *Acorns*,[1] given in the final round of LitIO in 2006 was chosen for analysis. Since the task is a typical dynamic programming batch task and does not imply any specifics in terms of programming style, we will not provide a full task description. There were submitted 130 algorithm-code complexes to this task and 34 of them were identified as not solving the task at all or as being in the initial phase of development where no output was produced. Those programs were not included into the research. The programming style of all the remaining 96 programs was evaluated. Figure 5.8 shows the relationship between the points obtained for programming style and points, obtained for black-box testing. The results obtained here are very similar to those obtained by G. Grigas.

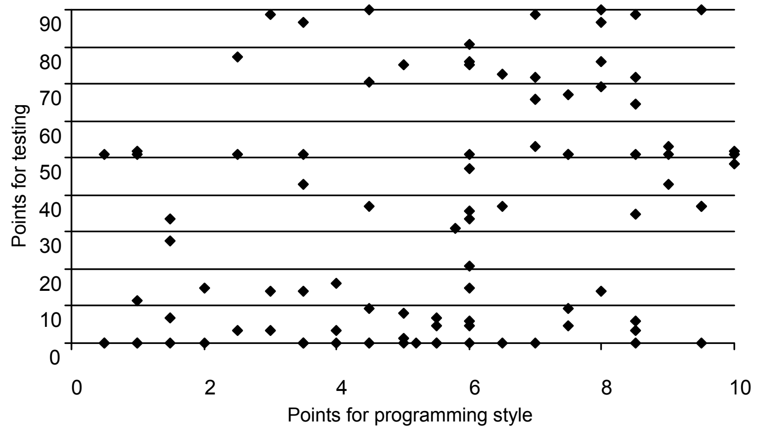

Figure 5.8: Scatterplot of the points for testing and the points for programming style The correlation between these two values is 0.33.

The statistics shows that the programming style of nearly half of the contestants is lower than satisfactory (i.e. below 5 points). We compared the programming style of algorithm-code complexes solving different tasks, but designed by the same contestants in the final round in LitIO'2006. There was no real difference between the programs submitted for the tasks where the programming style was evaluated and for the tasks where it was not included into the evaluation scheme (the contestants knew the scoring function in advance). The only visible difference was in the amount of comments – there were no or considerably less comments in the programs where the programming style was not evaluated. This can lead us to the conclusion that, in the contest environment, the students concentrate on the task and apply the

[1]The idea of the task is as follows: a squirrel is jumping from one position on a branch (a branch corresponds to a segment) to another. It takes 1 time unit to jump to a neighbouring position. The acorns are falling from the top of the tree in various positions of the branch at various time moments. The task is to count the maximum number of acorns the squirrel can catch.

programming style they follow in everyday life and only add some extra comments to improve it.

The results of testing the functional performance reveal (or are expected to) a combination of two different things: 1) suitability of the algorithm to solve a particular task; 2) correctness of the algorithm implementation. The correlation above shows the relation of programming style with a combination of these two characteristics. In order to get more exact results, we tried to separate those two features in the algorithm-code complexes and to calculate the correlation between the programming style and the correctness of implementation of algorithms.

Therefore we evaluated the quality of the implementations of algorithms of task *Acorn*. In this task the submission involved a verbal description of the algorithm as well. Therefore, in most cases it was possible to determine what kind of algorithm was implemented (or intended to implement) in the algorithm-code complex. Under the new evaluation performed during the investigation, the algorithm-code complex got full marks if it implemented the algorithm correctly, irrespectively of whether the algorithm itself is correct and effective or not. Figure 5.9 shows the relationship between the correctness of implementation and the programming style. It shows a stronger tendency that the better the programming style, the better the contestant succeeds in implementing his algorithm (not necessarily a correct or efficient one).

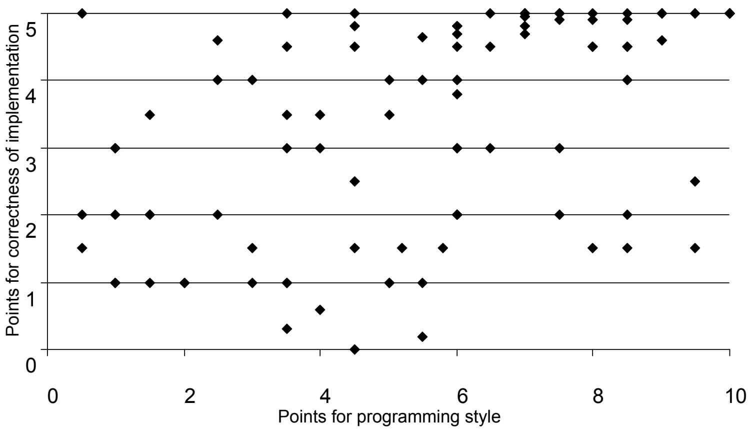

Figure 5.9: Scatterplot of the points for implementation correctness and the points for programming style The correlation between these two values is 0.468, the implementation correctness is measured in five points scale, the quality of programming style – in ten point scale.

5.4.5 Good Programming Style in Informatics Contests – Advantage or Necessity

B. W. Kernigan and R. Pike made a remark on the style of programs: *"In a world of ... relentless pressure for more of everything, one can lose sight of the basic principles – simplicity, clarity, generality – that form the bedrock of good software"*

(Kernighan and Pike, 1999). This should not happen in informatics contests. Such contests are the place where the top high school students come to compete. The lesson they might take from them is that an algorithm, implemented in "quick and dirty" style, and giving the satisfactory output, is as good as a clear and elegant program. This might take this lesson in their further professional lives.

Therefore the approach to force a good programming style as a necessity – to reject submissions (programs) that do not meet even the minimal requirements to program clarity – should be considered.

5.5 Conclusions

We compared the life-cycles of a software product and a submission using the waterfall model and identified differences. The first difference is that the phases *system and software requirements* and *operation* have only one way connection to the other phases. Another significant difference is that *the operation* phase of a submission is very short and no maintenance and support is required.

The software quality model ISO-9126-1 provides six software quality characteristics: functionality, reliability, usability, efficiency, maintainability, portability. Current evaluation scheme includes three characteristics: functionality, efficiency, and maintainability. We examined under what conditions and for what reasons each of the characteristics is either included or excluded from current evaluation scheme and found that the decision to include only three characteristics is motivated.

Two case studies has been performed on the evaluation of functionality, efficiency, and maintainability. During it around 250 submissions designed during LitIO'2006 and LitIO'2008 were analysed and classified manually.

In the first case study the expected score range of each submission was identified. The results were compared to the scores obtained by partial and all-or-noting batch scoring. The scores obtained by the two scoring schemes differ from the expected scoring in two ways: either more or fewer than points than expected are assigned. In total 20.2% (partial scoring) and 9.2% (all-or-nothing batch scoring) differed from the expected score ranges, 3.2% (partial scoring) and 2.6% (all or nothing batch scoring) of scores differed from the expected score ranges by 20 points or more.

The results corroborate that black-box testing does not fully correspond to score expectations. The percentage of unjustified scores in all-or-nothing batch scoring should be considered as more acceptable. However we suggest that providing more real time feedback during the contest and considering other forms of evaluation would anticipate the overall score to the expected one even more.

The second case study is related to evaluating maintainability. We found no commonly accepted way of measuring software maintainability. In informatics contests it is assessed in the form of evaluating the programming style. During the study we calculated the correlation between the points for implementation correctness and the points for programming style: it is 0.468. We conclude that the contestants who follow better style habits tend to implement their algorithms more successfully.

6 Improvement of Evaluation in the LitIO Scheme Using MCDA

In Section 3.3, it has already been shown that search for an improved evaluation in LitIO scheme can be approached using MCDA methods and algorithms. In Sections 3.4 and 3.5, the category of the problem and the roles involved in the MCDA process were defined. In this chapter, we will perform the main MCDA stages, presented in Section 3.6 as well as pilot the proposed evaluation in the LitIO model.

6.1 Problem Structuring

6.1.1 Introduction

The first step of the problem structuring phase is defining the decision context, i.e., collecting and structuring information and attitudes towards the problem under consideration, screening the concerns and priorities. That was completed in Chapters 2, 3, and 5. The concept of informatics contest was defined and screened, the contest goals, structure, domain of problems, and the current evaluation model were analysed. The experience of evaluation of algorithm-code complexes was also investigated in a similar situation, i.e., in programming courses of undergraduate studies together with the intention to get the answer whether such an experience can be transferred to informatics contests. The point of view was also motivated and selected, i.e., the quality of a submission and the concept of quality were analysed. We have identified that both problem solving and problem developing skills interrelate in a submission and both aspects should be taken into account during the evaluation. Before continuing to the construction of the concept of a submission and the quality of submission, the current evaluation in the LitIO model was analysed from the point of view of the existing software quality standards.

The main goal of the remaining problem structuring part is identifying objectives, creating a hierarchical model of objectives, and specifying the evaluation criteria and the concept of alternatives (submissions). After looking over many formal problem structuring techniques, we came to a decision to apply the GQM framework to this purpose. That was motivated in Subsection 3.7.2.

As we have concluded in Section 2.8, the quality is either conformance to very concrete specifications or conformance to the needs of the users. However, in this case, there is only one acting subject. It is the scientific committee which determines the specifications and at the same time is the only user of the submitted solutions. Therefore, the scientific committee has the final word in determining the quality of the submission and should take into account the contest goals and resource limitations, the problem solving part of the contest and the software quality models.

However, even though the final decision about the evaluation model will be made by the scientific committee, we decided that involvement of experts in the process of structuring will be beneficial and would provide a less biased result.

To participate in the problem structuring process we invited a group of ten experts. By an expert we defined a person having the background in informatics and at least five-year experience of working in national, regional and/or international informatics contests either as a member of the scientific committee, or as a jury member. Eight out of ten experts have been involved in the contests for more than ten years. We made an exception and invited as an expert one person who had one-year experience of being a member of the scientific committee. However, he had been the participant and the winner of many national and international Olympiads before he joined the scientific committee of LitIO.

Even though the object of discussion is the evaluation in LitIO, in order to discuss it from a broader perspective, the group of experts consisted of half the Lithuanian and half of international members, all having experience in various national, regional and international contests. Since the experts were located remotely, there was no interaction between them as a group during the work.

6.1.2 The Background of Evaluation in the LitIO Problem Provided to the Experts

The experts were provided all the relevant information about the structure, scope, and available resources for LitIO. The main part of creating the evaluation scheme consists of identifying aspects that need to be measured and defining metrics that measure each aspect. The following hierarchical model (created following the GQM framework) was designed and used as the basis of the questionnaire distributed to the experts (Fig. 6.1).

At the conceptual level, the concept of a goal (in GQM) or an objective (MCDA terminology) was replaced by the concept of *submission.* Strictly following the GQM framework, *submission* should be considered as the object of measurement. However, the main goal (measuring the quality of a submission) it already defined. By placing *submission* as the central item at the conceptual level, we attract the attention of the experts to the concept of *submission* and ask for critical attitude and suggestions for a modification of the current understanding of *submission.*

The second level is the attribute level. It corresponds to the operational level of the GQM framework. The third level is quantitative, it contains the metrics for measuring these attributes. The same metric can be applied for measuring several attributes.

We asked the experts to answer the following questions:

1. *What attributes of a submission are most relevant for determining a score and can be objectively measured? You can restrict yourself to what you consider the five most relevant attributes.*

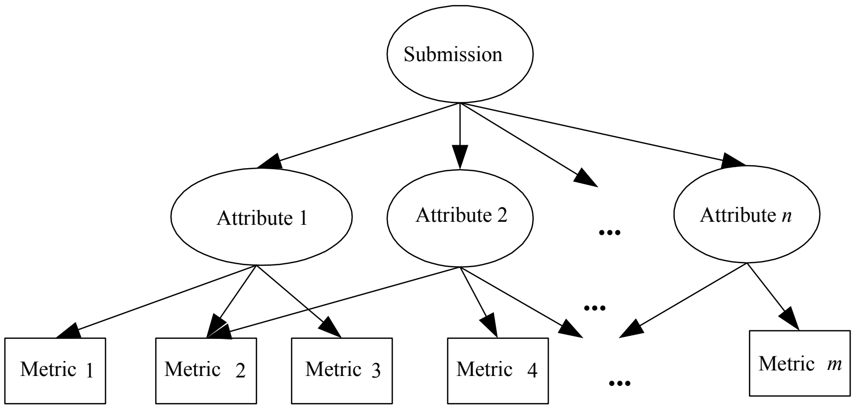

Figure 6.1: The framework of the evaluation scheme

2. *What metrics would you propose to measure these attributes (more than one metric could be used to measure each attribute). Define each metric as precisely as you can.*

3. *How would you suggest to implement each metric (taking into account the resources and limitations described below).*
   Metrics can be implemented by a manual measurement procedure, or by an automated procedure, or by their combination. Describe each procedure as precisely as you can.

4. *How would you suggest to integrate separate metrics to get one score (for one submission).*

All the material (the background and questionnaire) provided to the experts at the beginning of work can be found in Appendix A.2.

6.1.3 Defining the Concept of a Submission

Currently a submission consists of a verbal description of an algorithm and the source code of the implemented algorithm. It should be noted that the algorithm presented in the verbal description does not have to match the implemented algorithm. Therefore, the verbal description is graded independently of the implementation. It is graded even if the implementation is not submitted.

The experts came with two types of suggestions which could be classified into two totally opposite categories. One group of experts suggested to shrink a submission to the source code only, while the other group suggested to expand it by including test data as a new element of the submission.

The verbal description of an algorithm was the first submission element to be discussed.

The opinions expressed by the experts were totally opposite (Fig. 6.2). Some of them suggested removing the verbal description from the submission, motivating that *"the main skill of a contestant is whether he can make work his program or not; any other efforts outside this (e.g. verbal description with impressive ideas) are of a little help if the program does not work."* Others were more lenient, but strongly emphasized that these are of secondary importance and *"whatever is included into the evaluation model, functional correctness and efficiency are the main attributes to be evaluated, ... verbal algorithm description is for those who did not have enough time to implement their solution, but not for those who have failed to implement it. The contestants should submit either a verbal description or the implementation, but not both."* In the responses of some experts it was mentioned that it was most likely that the verbal description of an algorithm is used for identifying heuristics. This helps to achieve that submissions with heuristic algorithms would not get a full score. The experts suggest other ways to try to avoid this, rather than through a verbal description.

The other experts suggested to include an implementation description or reasoning on the design. Currently the required algorithm description may have no connection with the implementation. Reasoning on the design might come in the form of a separate text or comments and *"ideally, once the design decisions have been established, the program code is a straightforward derivative (synthesis)."*

None of the experts clearly supported the current practice, where the verbal algorithm description may have no connection with the implementation. There had been lengthy discussions in the scientific committee of LitIO whether it should be required to describe the implemented algorithm or just any algorithm which solves the given problem. The decision not to connect the description and the implementation was motivated by a simpler evaluation process. Another reason was the possibility for the contestants to come up with a better solution, once they have implemented their solution and realized that it was not good enough.

One more suggestion found in the responses of several experts was to include the test data, preferably with motivation. Each test should consist of an input file and the corresponding output file. One of the experts commented that *"when equal programs have been developed by two contestants (or companies), I would have more trust in the program that was systematically tested over one that was not."* One suggestion was to include test data in the form of a challenge phase like in the TopCoder contest (Top, 2010; Opmanis, 2006; Skienna and Revilla, 2003), i.e., the participants would have to provide test data intended to break other submissions (prepared by the jury or those of other contestants). Different approaches of the experts represent the existing variety of views in the community of informatics contests (Cormack, 2006; Pohl, 2008; Verhoeff, 2006).

To choose a particular submission model, we decided to focus on the goals of LitIO as the key factor. A very important goal in LitIO is an educational goal, i.e., to disseminate good programming practice. Even though we agree with the point of one of the experts that *"contest as an educational event... sounds strange;... you learn a lot of things before or after, but not during the contest"*, it must be taken into account that there are few qualified teachers in Lithuania, who have experience in applying and teaching good programming practices and who would know how to write a program in conformance with the academic standards. Therefore two

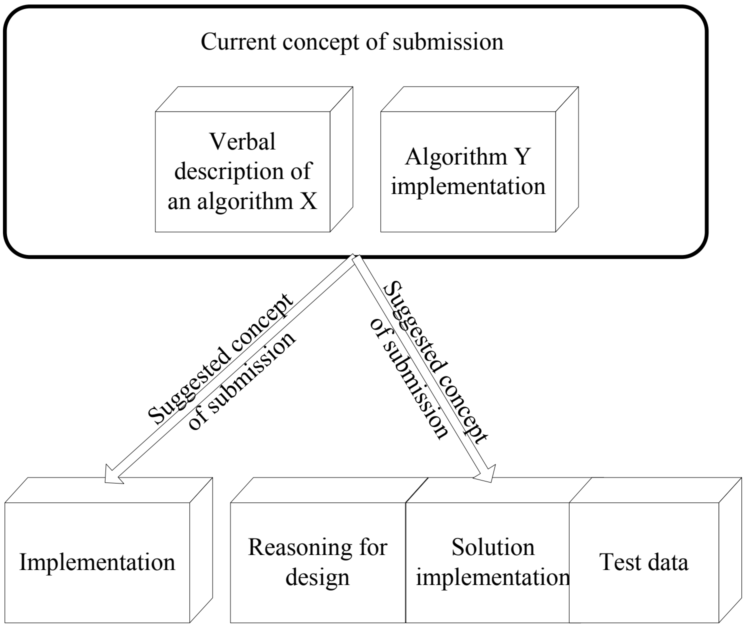
Current concept of submission
Verbal description of an algorithm X
Algorithm Y implementation
Suggested concept of submission
Suggested concept of submission
Implementation
Reasoning for design
Solution implementation
Test data

Figure 6.2: Suggestions of the experts to modify the current submission concept

major events for high-school students, namely, the maturity exams in programming (Blonskis and Dagienė, 2006) and the Lithuanian Olympiads in Informatics serve as guidelines and a kind of "reference" for the teachers.
The proposed submission model consists of:

- *reasoning on the design* which replaces the verbal description of an algorithm,
- *solution implementation* presented as the source code in one of the allowed programming languages; it should be based on reasoning on the design, if such is provided,
- *a set of test cases with motivation.*

A long experience of the author working in LitIO has shown that it took many years for the contestants of LitIO and their coaches to get used to providing material other than the source code (i.e., writing verbal descriptions of an algorithm). In order to avoid reluctance towards the new element of submission, i.e., a set of test cases, it would make sense to look for different options to implement the new concept of submission (for example, to arrange it as some kind of a challenge phase rather than simply ask for a set of test cases for checking submissions).

There is one more practical motivation for that. Two attributes might be checked for each separate test case (input data and the corresponding output). One of them is validity: is it a valid input file and the correct valid output to this input. Another attribute is whether the test is really evaluating the feature it claims to. Checking the latter might be complicated or even require writing several different checkers. For example, if it is claimed that the test evaluates the performance of a solution when the input graph satisfies specific conditions, then the jury might need to code a checker to verify whether the graph modeled in the input file really has those specific properties.

Another practical issue is related to the contest structure. Flexibility has always been present in LitIO to reasonably distribute the efforts of the contestants and the jury. Sometimes two tasks instead of three are given in an exam session, for some tasks it is not required to present verbal descriptions of the algorithm (those tasks are chosen very carefully) and for some tasks the programming style is not graded, as it has been noticed that the programming style of submissions is not influenced by a concrete grading scheme (i.e., whether the points are awarded for style or not). Complementing a submission with test cases would require even more flexibility since generating test cases might require a lot of extra time depending upon the task.

6.1.4 Submission Attributes

The current grading model presents three measurable attributes of a submission (Fig. 6.3. They are the quality of a verbal description of an algorithm (sub-attributes: correctness and efficiency of the described solution and quality of the description), performance of the already compiled program code (sub-attributes: functional correctness and time and memory efficiency), and programming style (sub-attributes: consistency, clear layout, use of proper names, suitable explicit substructures, absence of magic numbers, proper comments).

Concerns of some of the experts about the elements of submission other than implementation were reflected in the proposals about the attributes. Those who restricted a submission to the source code only were against any attributes except for the performance of already compiled code. They especially stressed the programming style. *"Once the style is bad enough, the contestant will leave a bug and will bear consequences. If the implementation is fully correct this means that the style was good enough"* . Here is the opinion of another expert who refers to the research of (Grigas, 1995) who investigated the relationship of programming style and the IOI score in IOI'1994: *"goto was used by the best and worst students therefore it is hard to say how particular programming construction influences achievement of the contestant... defining some formal criteria of what is a good style and what is not seems to be extremely hard; better style leads to better programs and therefore to better results"*.

Other experts expressed different attitudes. Another interesting model consisting of five attributes was proposed [1]. The first attribute is *the quality of the solution idea*. The next three attributes refer to concrete parts of submission, i.e., *the quality of description of the solution idea*, *correctness of implementation (with respect to*

[1] A similar model is applied in *Bundeswettbewerb Informatik* (Bun, 2010).

the solution idea), the quality of implementation (source code quality). The last attribute, - *conformance to the requirements of the task description*, - refers to the entire submission.

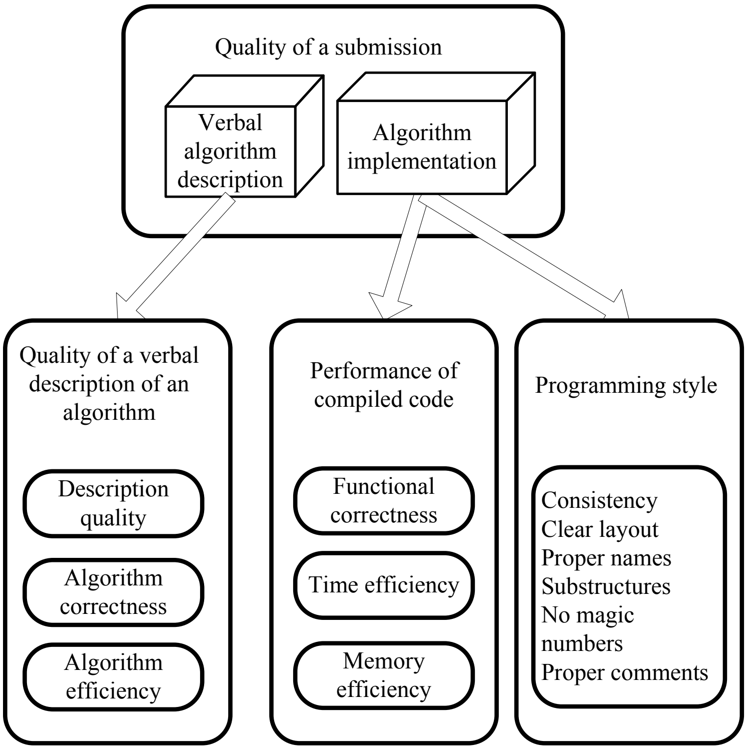

Figure 6.3: Attribute level of the current evaluation scheme The sub-attributes of programming style are not separated because the score is not a direct combination of evaluation of each of the attributes.

We found this model interesting because it considers the submission as an integral entity rather than a set of separate submission elements. From all the proposed models it seems to be most educationally motivated. The quality of the solution idea is one concept, not divided between the reasoning on design and implementation. The implementation correctness is related only to implementation, but not to a combination of solution and implementation correctness. The model seems to separate problem solving and engineering, which is the ongoing problem of the current model (small implementation mistake resulting in the loss of many points). Despite its attraction, the most questionable point in this model is its practical implementation within contest time pressure. For example, *"the independent grading of the quality of the solution idea can be done in the manual way only... otherwise it will be mixed with implementation correctness"*. The model (the expert provided the whole model not only the attributes) seems to be a good choice for the maturity examination in programming or for informatics contests smaller than LitIO. This model might also be implemented for some separate tasks in the finals of LitIO, where the number of submissions of one task may be rather low.

Several experts suggested the same decomposition of the quality of the verbal description of an algorithm as it is used now in LitIO, i.e., to emphasize the correctness and the efficiency of the described algorithm. However, one of the experts suggested a different approach by putting emphasis on the design issues. The attribute *quality of design reasoning* can be decomposed into *"story" organization* (i.e., appropriate separation of concerns, introducing appropriate concepts and notation), *effective reuse of prior knowledge* (e.g., standard algorithms and data structures) and *the level of formality and convincingness.* This decomposition seems to be more attractive as it puts emphasis on the design issues to reveal which is the main purpose of the written material rather than serving as double award or punishment for incorrect or inefficient solutions.

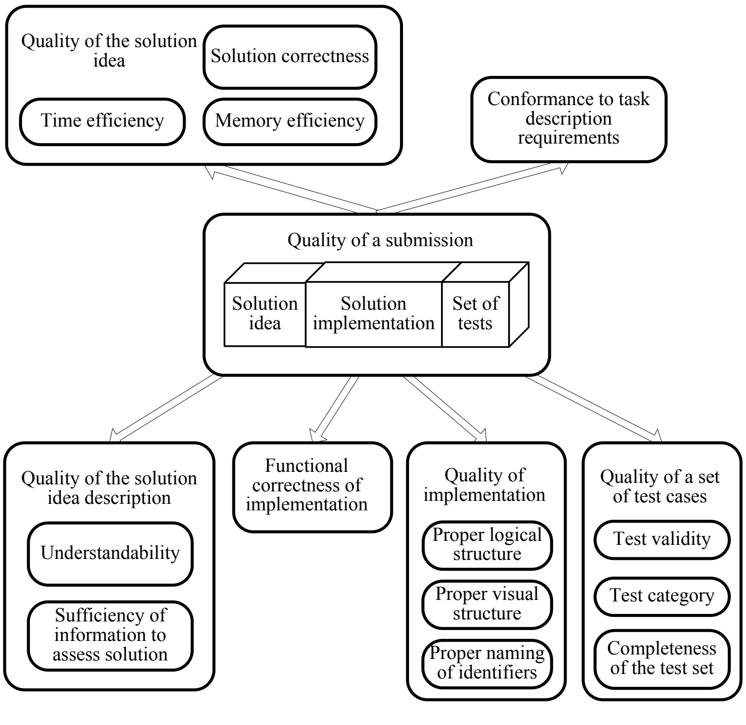
Quality of the solution idea
Solution correctness
Time efficiency
Memory efficiency
Conformance to task description requirements
Quality of a submission
Solution idea
Solution implementation
Set of tests
Quality of the solution idea description
Understandability
Sufficiency of information to assess solution
Functional correctness of implementation
Quality of implementation
Proper logical structure
Proper visual structure
Proper naming of identifiers
Quality of a set of test cases
Test validity
Test category
Completeness of the test set

Figure 6.4: Attribute level of the evaluation model which separates the problem solving part from the engineering part It is complemented with attributes to evaluate the quality of test cases.

With a shift of emphasis, there still remains the question whether the correctness and efficiency of design decisions should be evaluated or not. As this is the only place in the model where the solution idea is evaluated explicitly (it is evaluated implicitly when testing the implementation), we decided to leave that as a sub-attribute.

Since tests were included into the submission, some attributes should reflect that. Three sub-attributes referring to a test set can be measured. They are: test validity and belonging to some category (measured of each test separately), and completeness of the whole test set.

The attribute *conformance to the requirements* of the task description was proposed by several experts. The attribute should be in the form of a checklist and also might act as the coordinator between other attributes. For example, the item of this attribute might be the correspondence of reasoning on design to the implementation.

None of the experts suggested *time spent on solving the task as an attribute*, which is common in ACM-ICPC type contests, where each minute, from the contest starting time till the moment the submission is accepted, is turned into one penalty point (ACM, 2010a) or if stated in a more positive way, – the participant gets a bonus for each minute from the submission time till the end of the contest (Myers and Null, 1986).

The full scheme of the suggested attribute level is presented in Fig. 6.5.

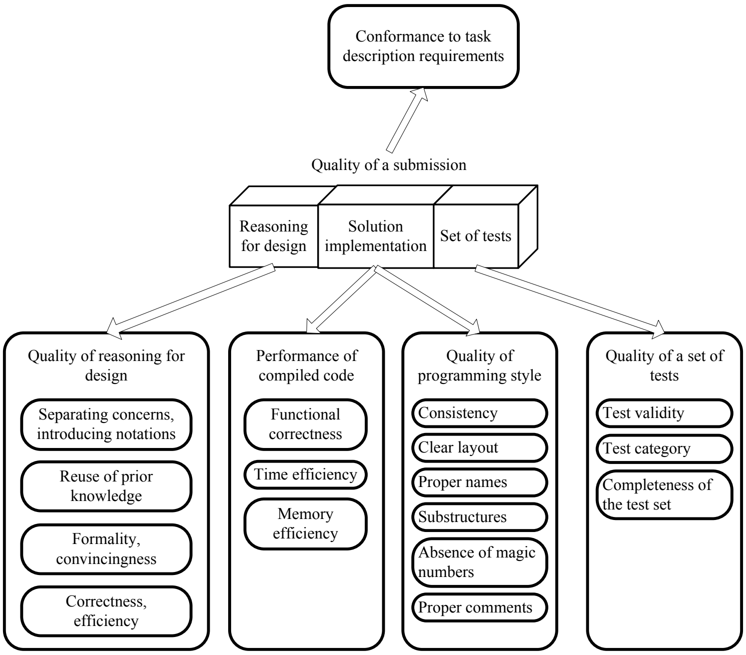

Figure 6.5: The proposed attribute level of the evaluation scheme

6.1.5 The Choice of Metrics for the Attributes

The proposed evaluation scheme includes five attributes that can be measured to evaluate the quality of the submission. Some of those attributes require manual grading. All the experts provided suggestions regarding the measuring performance of the compiled code. About half of the experts provided suggestions of the metrics for measuring other attributes. We will review all the attributes one by one.

6.1.5.1 Quality of Reasoning on Design

Four sub-attributes were identified in the reasoning on design: *separating concerns and introducing notation, reuse of prior knowledge, formality and convincingness, and correctness and efficiency* (Table 6.1). All these sub-attributes require manual grading. A concrete evaluation scheme that presents the taxonomy of various approaches to solving the problem should be developed for each task. While it is accepted that tests are designed prior to the contest and not changed during evaluation, *"the taxonomy for grading reasoning on design might have to be adapted during grading as blind spots might be discovered. This resembles IMO grading"* (Verhoeff, 2002).

There were many suggestions to include clarity and understandability in the evaluation model. However, clarity and understandability seemed to be more important in the previous model, where the only other sub-attributes were correctness and efficiency. In this model, clarity as a separate metric seems to be less important.

Metrics	**Measuring**	**Scale**	**Comments**
C_1 Level of clarity and understandability	Manual	Ordinal scale	Applies to the whole attribute; might influence other metrics of this attribute.
C_2 Level of story organization and concern separation	Manual	Ordinal scale	Proper taxonomy should be developed for each task
C_3 Level of reuse of prior knowledge	Manual	Ordinal scale	
C_4 Level of convincingness and formality	Manual	Ordinal scale	
C_5 Level of correctness of design decisions	Manual	Ordinal scale	
C_6 Level of efficiency of design decisions	Manual	Ordinal scale	

Table 6.1: Suggestions for metrics to measure quality of reasoning for design

6.1.5.2 Performance of the Compiled Code

Three sub-attributes were identified in the performance of the compiled code: functional correctness, time efficiency, and memory efficiency. All the three sub-attributes are part of the current evaluation scheme, and the experts approved the current metrics.

It is accepted to check the functional correctness automatically, using the black-box testing strategy, even though this does not guarantee that all the faults will be discovered (Williams, 2006). Each submission is executed with each test input. The test is considered to be passed successfully, if the program finishes its execution within the given time and memory limits and provides a correct output to the input. Time and memory efficiency is measured indirectly. Tests are designed in such a way that they would benchmark solutions with different time or memory efficiency, i.e., the solutions with a certain efficiency are expected to pass a certain subset of tests.

Sub-attribute	**Metrics**	**Measuring**	**Scale**	**Comments**
Functional Correctness	C_7 Does the program try to solve the task?	Manual	Boolean: Yes/No	It makes sense to use the metrics if the tests are not grouped
Functional correctness	C_8 Small correctness tests focusing on the specific basic categories of input data	Automated	Boolean: Pass/Fail for each test	All these tests cases should be solved correctly in order to score some points for the performance
Functional correctness	C_9 Correctness tests focusing on different categories of input data	Automated	Boolean: Pass/Fail for each test	Grouping test cases was implied by most experts
Time efficiency, memory efficiency	C_{10} Efficiency tests sorted into groups to benchmark the submissions of different efficiency	Automated	Boolean: Pass/Fail for each test	Grouping test cases was implied by most experts
Time efficiency, memory efficiency	C_{11} Efficiency tests sorted into groups to benchmark the submissions of different efficiency	Automated	Ratio: exact execution time for each test	Execution time might be used either as Boolean or ordinal metrics

Table 6.2: Suggestions for metrics to measure performance of the compiled code

However, if the program fails while executing some test, then, in general, without a closer analysis it is impossible to determine whether it failed due to functional incorrectness or due to low time or memory efficiency. One of the experts wrote *"if the program fails correctness tests, most of the time it will fail the efficiency tests*[1]*, not to mention the fact that in some cases the line between an incorrect solution and an inefficient solution is unclear"*. It is possible to measure the performance of the compiled code as an attribute; however it is not always possible to measure separately each sub-attribute.

Despite that, a majority of the experts suggest to measure two sub-attributes separately, i.e., to differentiate the tests into functional correctness and efficiency tests (both efficiency sub-attributes should be merged). Some of them proposed to introduce small correctness tests. They should test the very basic properties which the submission would have to solve correctly. I.e., in order to score any points for the performance, the program *must* solve correctly some very basic cases of the whole problem.

Also there were suggestions to measure the exact running time of a program with each test (i.e. to use it as a metrics) and take it into account when calculating the score. Measuring the exact program execution time with the purpose to identify its complexity is a sensitive issue not just due to the choice of test data. Hardware issues, compiler options, differences between programming languages influence the program performance.

In our model, we will distinguish correctness and efficiency tests in order to leave room for exploring various score aggregation models where the points for efficiency tests are related to the points for correctness tests.

There were suggestions to add one more metrics to functional correctness. The experts suggested to manually identify the programs which do not try to solve the task, but output the same answer all the time (e.g. *no solution*) or simply guess the answer (we do not refer to using randomization as part of the solution strategy), as well as to assign zero score for the performance. Indeed, if there are situations where for some reasons grouping is not used, then this category of submissions may score an inadequate amount of points. The overview of metrics proposed for measuring the performance of the compiled code is presented in Table 6.2.

6.1.5.3 Quality of the Programming Style

The quality of programming style has several sub-attributes (consistency, clear layout, proper naming, suitable substructures, absence of magic numbers, and proper comments). The experts proposed three different metrics for evaluating the quality of programming style and suggested to use both manual and automated grading.

One metric assumed the evaluation of the quality of programming style as a whole, taking into account all the sub-attributes, but not evaluating them separately. This should be manual grading with the grading results presented on an ordinal scale. Another suggestion was to measure the sub-criteria separately, presenting the results on an Ordinal scale again and afterwards using each measure to aggregate into one

[1]We have not found corroborating empirical evidence in the literature.

Metrics	Measuring	Scale	Comments
C_{12} Level of the Quality of programming style	Manual	Ordinal scale	This metrics should be used iff the other metrics from this table are not used.
C_{13} Level of Consistency	Manual	Ordinal scale	Possibilities for replacing/combining with automated measuring might be investigated
C_{14} Layout clarity	Manual	Ordinal scale	
C_{15} Level of proper naming	Manual	Ordinal scale	
C_{16} Suitability of substructures	Manual	Ordinal scale	
C_{17} Absence of magic numbers	Manual	Boolean scale: Yes/No	
C_{18} Suitability of comments	Manual	Ordinal scale	

Table 6.3: Suggestions for metrics to measure the quality of programming style

score for the quality of the programming style. Actually, there is no need to choose one approach. Both of them can be used depending upon available resources, as the first approach requires much less time, while the results based on the second approach would be much clearer to the contestants.

The third approach (suggested by several Lithuanian and foreign experts) included a combination of manual and automated grading. However, it has been emphasized that the possibilities of automated grading of programming style must be researched first and the proper tools developed. Such an approach is applied in the evaluation procedure of the Lithuanian maturity exam in programming, and special software was developed for that. The exam submissions are much simpler in complexity and shorter in length than the contest submissions, and the only available compiler is FreePascal (Skūpas and Dagienė, 2008). However, none of the experts provided concrete metrics for performing this type of semi-automated grading.

The overview of metrics proposed for evaluating the quality of programming style is presented in Table 6.3.

6.1.5.4 Quality of a Set of Tests

The evaluation scheme proposed three sub-attributes to measure the quality of the submitted test set. The first sub-attribute is the validity of each test (a pair consisting of an input and the corresponding output). The test validity means that the input is a valid input according to task specifications and the output is a correct output to the given input.

Metrics	Measuring	Scale	Comments
C_{19} Test validity	Automated	Boolean: Yes/No for each input/output pair	
C_{20} Belonging to a certain test category	Manual, Automated	Boolean: Yes/No for each test and accompanying motivation	Motivation is evaluated manually
C_{21} Completeness of the whole test set	Automated	Ratio: Percentage of coverage statistics	Special plug-in for the contest system might be needed. Percentage of coverage statistics

Table 6.4: Proposals for metrics to measure the quality of a set of tests

The next sub-attribute is a test category. Each test should be submitted with motivation, explaining what category of input it targets at. The most sophisticated challenge seems to be verifying that the provided test really targets at the category that it claims to be. Checking this automatically might require too many resources for some tasks.

The final sub-attribute is completeness. This attribute refers to the whole test set and it checks to which extent the submitted test set covers the domain (Table 6.4).

6.1.5.5 Conformance to the Task Description Requirements

The measurement of the conformance to the task description requirements should be arranged in the form of a checklist. The checklist might depend on a concrete task. In some cases, it might be of secondary importance or not needed at all, but if the absence of some item on the checklist makes it impossible or difficult to evaluate another item, then a low grade for this attribute will result in a low grade for another attribute as well.

An obvious item to be included in the checklist is the correspondence of the reasoning on design to the implementation. Another item to be included is the presence of motivation for test-cases (Table 6.5).

Metrics	Measuring	Scale	Comments
A checklist based on the specification of a concrete task	Manual, Automated	Boolean:Yes/No for each item of the checklist	The need for this attribute in general and its use in the evaluation scheme depends upon a concrete task

Table 6.5: Proposals for metrics to measure the conformance to task description requirements

While working on this model, we tried to be reflective, and discuss and incorporate as many ideas and suggestions as possible that we found in the responses of the experts, even if they were not included into the proposed model. The model has room for flexibility and for tailoring through score aggregation methods.

6.2 Model Building and Piloting

In the previous section, we performed problem structuring and defined a wide set of evaluation metrics. They can be applied and modified in various situations, depending upon technical resources, and the preferences of the scientific committee.

The second step is the choice of an MCDA model. The choice was discussed and made in Section 3.8.

Next we have to illustrate how to adapt the proposed general evaluation scheme to a concrete task. For this purpose we selected four tasks and gave them to solve during a small contest held in a training camp. The task descriptions can be found at (Lit, 2010) (*training camp'2010, day1*).

In this section, we present all the steps how the evaluation scheme, developed in the thesis, was applied in practice.

We will not provide the motives for each choice we have made (e.g., why we decided to include one or exclude another criterion to/from the evaluation scheme). The decision was made taking into account the goals of the training camp, the nature of the tasks, and the available human resources for the evaluation and task preparation, which is not relevant in terms of this piloting. The details would be excessive, because the main goal of this section is to experiment and illustrate how the evaluation scheme could be applied in practice.

6.2.1 Specification of the Components of the Evaluation Scheme for Concrete Tasks

The evaluation scheme, adjusted to a concrete task, should contain the components, presented in this subsection. Note that some components might be calculated once and remain valid for more than one task. For example, the weights of importance of the jury members can be calculated at the beginning of the school year and used for all the tasks given in the same contest season.

6.2.1.1 Scales for Linguistic Variables

The scales, to be used in the evaluation scheme, could have been defined when we chose the MCDA algorithm. However, the choice of scales is considered to be *intuitive* and left for a decision maker (Chou et al., 2007; Rao, 2007). Therefore, we decided to select the scales here, thus saying that other scales could be used as well.

The chosen GDM method uses three linguistic scales.

The first scale is used for assigning weights of importance to decision makers. We will use the four-point scale presented in Table 6.6 and Fig. 6.6. Note that the scale is empty on the right side, i.e., there will be no jury members whose opinions will be considered as of a very low value. The crisp values given in Table 6.6 were

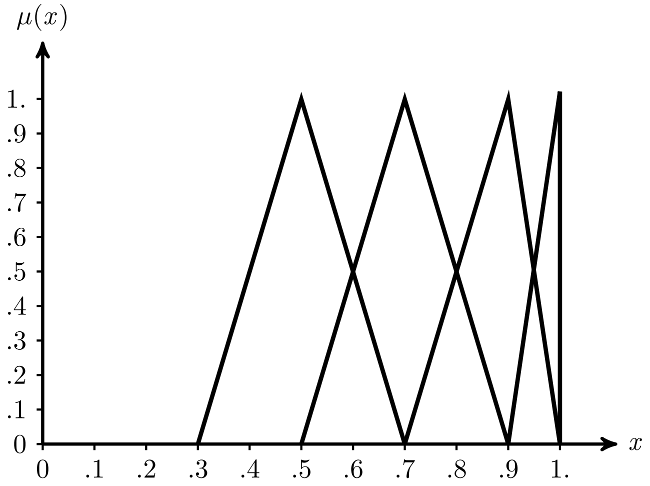

Figure 6.6: The scale for determining a relative importance of decision makers (Lu et al., 2007)

Item of linguistic scale	Numerical weights	Crisp values
Normal	$(0.3,\ 0.5,\ 0.7)$	0.5
Important	$(0.5,\ 0.7,\ 0.9)$	0.67
More important	$(0.7,\ 0.9,\ 1)$	0.83
Most important	$(0.9,\ 1,\ 1)$	0.95

Table 6.6: Weights of the linguistic scale for determining a relative importance of decision makers

calculated as it has been described in Subsection 3.8.5.3. (Lu et al., 2007) suggest to use this scale for determining a relative importance of the decision makers.

The second linguistic scale is required for expressing relative importance weights of evaluation criteria. For this purpose we will use the eleven-point scale proposed by (Chen et al., 1992). We have deliberately chosen a large scale, because in the end the weights for criteria will be distributed in a 100 point scale, therefore providing a larger scale gives more flexibility for decision makers. The scale is presented in Table 6.7 and illustrated in Fig. 6.7.

The third scale is required for performing the evaluation, i.e., expressing the scores of manual holistic qualitative evaluation. We will use the nine-item scale proposed by (Sule, 2001) and already presented in Table 3.1 and Fig. 3.8. This scale will be repeatedly used in the evaluation of each submission by each jury member involved in the evaluation. The choice of scale size is motivated by (Miller, 1956) (who showed that individuals cannot simultaneously compare more than seven objects plus minus two), and the fact that until now a ten or twenty-point scale was used for a qualitative holistic evaluation in LitIO.

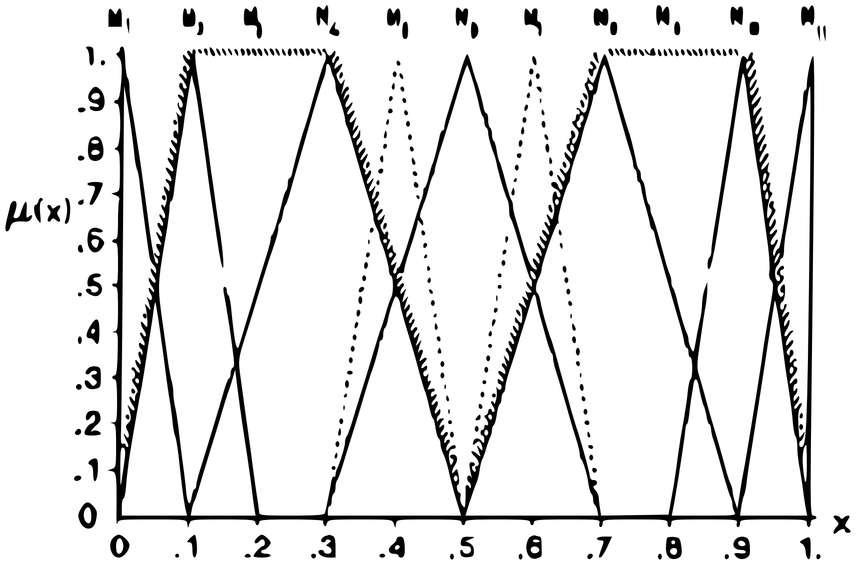

Figure 6.7: The scale for determining a relative importance of the evaluation criteria (Chen et al., 1992)

Item of linguistic scale	Numerical weights	Crisp values
Exceptionally low (EL)	$\big(0,\ 0,\ 0,\ 0.1\big)$	0.05
Extremely low (XL)	$\big(0,\ 0.1,\ 0.1,\ 0.2\big)$	0.14
Very low (VL)	$\big(0,\ 0.1,\ 0.3,\ 0.5\big)$	0.25
Low (L)	$\big(0.1,\ 0.3,\ 0.3,\ 0.5\big)$	0.33
Below average (BA)	$\big(0.3,\ 0.4,\ 0.4,\ 0.5\big)$	0.41
Average (A)	$\big(0.3,\ 0.5,\ 0.5,\ 0.7\big)$	0.50
Above average (AA)	$\big(0.5,\ 0.6,\ 0.6,\ 0.7\big)$	0.59
High (H)	$\big(0.5,\ 0.7,\ 0.7,\ 0.9\big)$	0.67
Very high (VH)	$\big(0.5,\ 0.7,\ 0.9,\ 1\big)$	0.75
Extremely high (XH)	$\big(0.8,\ 0.9,\ 0.9,\ 1\big)$	0.86
Exceptionally high (EH)	$\big(0.9,\ 1,\ 1,\ 1\big)$	0.95

Table 6.7: Weights of the linguistic scale for determining a relative importance of the evaluation criteria

6.2.1.2 Weights of Importance of the Jury Members

The weights of relative importance of the decision makers are either decided in a group discussion or by a higher management level (Lu et al., 2007). We have already emphasised that, in this work, we will not involve the decision makers into discussions. Therefore we decided to assign the weights based on their experience in LitIO:

Normal. The jury members whose activity was very limited in the last three years.

Important. Active jury members who have been involved in LitIO for less than five years.

More important. Active jury members who have been involved in LitIO from five to ten years.

Most important. Active jury members who have been involved in LitIO for more than ten years.

Currently there are thirteen jury members and the assignment of the importance weights is presented in Table 6.8.

Jury member	His/her relative importance	Crisp weight
M_1	Most important	0.95
M_2	More important	0.83
M_3	More important	0.83
M_4	Important	0.67
M_5	Most important	0.95
M_6	Most important	0.95
M_7	Normal	0.5
M_8	Normal	0.5
M_9	Important	0.67
M_{10}	Normal	0.5
M_{11}	More important	0.83
M_{12}	Normal	0.5
M_{13}	Normal	0.5

Table 6.8: Relative importance weights of jury members

Three jury members, whose linguistic values of importances were: *more important*, *more important*, and *normal*, participated in the piloting.

6.2.1.3 List of Required Components of Submission

LitIO allows flexibility, and for different tasks it might be required to submit different sets of submission components. A submission can have from one to three components. Therefore we define the possible templates in advance (Table 6.9). Note that an algorithm-code complex is always required to be submitted.

The list of the required components for each pilot task is presented in Table 6.11.

Template	ID	Reasoning on design	Algorithm-code complex	Test cases
1	SRAT	Included	Included	Included
2	SAT	—	Included	Included
3	SRA	Included	Included	—
4	SA	—	Included	—

Table 6.9: Templates for a set of required submission components.

6.2.1.4 List of Attributes to be Evaluated

The evaluation scheme should include the list of submission attributes the quality of which will be measured. The flexibility of LitIO allows different sets of attributes to be included into the evaluation scheme. Therefore we have made a list of possible templates and presented them in Table 6.10. The list of attributes the quality of which will be measured for each pilot task is presented in Table 6.11.

Template	ID	Reasoning to design	Performance	Programming style	Test cases	Conformance to task description
1	RPSTC	Included	Included	Included	Included	Included
2	RPST	Included	Included	Included	Included	—-
3	RPSC	Included	Included	Included	—	Included
4	RPTC	Included	Included	—	Included	Included
5	PSTC	—	Included	Included	Included	Included
6	RPS	Included	Included	Included	—	—
7	RPT	Included	Included	—	Included	—
8	RPC	Included	Included	—	—	Included
9	PSC	—	Included	Included	—	Included
10	PST	—	Included	Included	Included	—
11	PTC	—	Included	—	Included	Included
12	PC	—	Included	—	—	Included
13	PT	—	Included	—	Included	—
14	PS	—	Included	Included	—	—
15	RP	Included	Included	—	—	—
16	P	—	Included	—	—	—

Table 6.10: Templates of the evaluation scheme at the attribute level

6.2.1.5 Weights of the Evaluated Attributes

During problem structuring five attributes were identified: quality of reasoning on design, performance of the compiled code, quality of the programming style, quality

Task No	**Submission Template ID**	**Attribute template ID**
Task 1	SRAT	RPT
Task 2	SA	PS
Task 3	SA	PS
Task 4	SA	P

Table 6.11: List of required submission components and the measured attributes for the piloted tasks. The explanations of submission and attribute ID's are given in Tables 6.9 and 6.10.

Jury member	**Reasoning on design**	**Perfor- mance**	**Program- ming style**	**Test cases**
M_1	A	EH	EL	L
M_2	BA	XH	VL	VL
M_3	BA	H	L	BA
M_4	A	H	L	A
M_5	L	EH	EL	BA
M_6	A	VH	BA	AA
M_7	XH	VH	H	A
M_8	VH	XH	A	H
M_9	BA	EH	L	AA
M_{10}	H	VH	H	A
M_{11}	A	XH	L	A
M_{12}	BA	H	AA	AA
M_{13}	L	EH	XL	EL

Table 6.12: Relative importance of the four attributes proposed by the jury members

of a set of test cases, and conformance to the task description requirements.

The last attribute is highly task-specific. It should be used for ensuring the required relations between other attributes (i.e., partial value inter-criteria functions), but there will be no points assigned to its criteria separately. An example of a criterion for this attribute might be: *does the algorithm-code complex implement the algorithm described in the reasoning on design.* In the case of the negative answer, the points for the reasoning on design are canceled.

Weights of attributes is a subject of serious discussions in LitIO, therefore we involved all the jury members in the decision. The jury members were provided the list of the four attributes and were requested to propose relative weights of their importance, using the 6.7 scale. The responses of the jury members are presented in Table 6.12.

Basing on the responses of LitIO jury members, their relative weights, formula 3.14, we calculated the aggregated attribute weights and developed eight templates of distribution of the points between the attributes.

No	Template ID	Reasoning onr design	Perfor- mance	Program- ming style	Test cases
1	RPSTC, RPST	0.23 (0.25)	0.39 (0.40)	0.17 (0.15)	0.21 (0.20)
2	PSTC, PST	—	0.50 (0.50)	0.22 (0.20)	0.28 (0.30)
3	RPTC, RPT	0.28 (0.30)	0.47 (0.45)	—	0.25 (0.25)
4	RPSC, RPS	0.29 (0.30)	0.49 (0.50)	0.22 (0.20)	—
5	PTC, PT	—	0.65 (0.65)	—	0.35 (0.35)
6	RPC, RP	0.38 (0.40)	0.62 (0.60)	—	—
7	PSC, PS	—	0.70 (0.70)	0.30 (0.30)	—
8	PC, P	—	1	—	—

Table 6.13: Templates of possible weights of the attributes. The rounded attribute weights are presented in parentheses. The column *Template ID* indicates for which attribute templates in Table 6.11 the weights in the corresponding row are valid.

The aggregated attribute weights calculated using formula 3.14 do not necessarily sum up to one, and the method does not require it, because the normalisation is performed in 6.3. However, the attribute weights must be presented to the contestants, therefore it is reasonable to normalise them before performing the next step. It also makes sense to round off the calculated weights to look them more natural to the contestants. The calculated normalised weights (precise normalised values and the rounded values) are presented in Table 6.13.

Evaluation in the LitIO problem is *repeated*, which means that the jury members are free to change their opinions, the composition of the jury may also change, and the values of aggregated attribute weights can be recalculated following the formula. Moreover, it might happen that a task is very specific, and the jury might be willing not to use the templates, but to assign and recalculate weights taking into account the task specifics.

6.2.1.6 Inter-Attribute Function for Relating the Programming Style to the Performance of the Compiled Code

The attribute *quality of programming style* is expected to be related to the attribute *performance of the compiled code* and the relationship has to be established by the jury. Each jury member was asked to propose an inter-attribute function, and three types of proposals came up:

- Programming style is evaluated only if the score for the program performance is not smaller than $X\%$. Otherwise the score for the programming style is zero.

- Programming style is evaluated only if some pre-defined set of tests (e.g., at least one test, small functional correctness tests, all functional correctness tests) is passed. Otherwise the score for the programming style is zero.

- The final score for the programming style is calculated using the formula: $Score_{style_final} = Score_{style} \times P$, where P is the percentage of points scored for the performance of the compiled code.

We did not expect that the proposals will be so different, and asked the jury members to express their opinions about the proposals in a four-point linguistic scale as pairwise comparisons. Thus, we modelled this as a small independent MCDA problem and decided to apply the *Analytic Hierarchy Process* (AHP) (Lu et al., 2007). The method uses pairwise comparisons to find the most wanted alternative and assumes one decision maker and several criteria. To apply the method, we modelled the problem in the following way.

We had no explicit criteria, just the opinions of all the jury members expressed as a pairwise comparison of acceptability of alternatives. Therefore we decided that the jury members act as criteria, and their opinions act as performance of alternatives in terms of criteria. We constructed a square matrix of relative importance of the criteria (jury members) and a pairwise comparison matrix of the three alternatives for each jury member. The weight of each element in each matrix was retrieved by calculating the geometric mean of a row and normalising the resulting vector (as suggested by (Saaty, 1980)). The contribution of each alternative to the overall goal was calculated and the overall priority for each alternative was obtained by summing the product of the attribute weight and the contribution of the alternative with respect to that attribute.

After completing the calculations, the following values for each proposed alternative were obtained: 0.3442, 0,3655, and 0,2904. It means that none of the alternatives was dominating. The most preferred alternative is the second one, and the first one is just slightly behind the second.

From the three options of the second alternative (at least one test is passed, a small set of tests for the functional correctness is passed, and all the tests for functional correctness are passed) the majority of jury members preferred the second alternative.

Thus we established the following procedure for assigning points for the quality of programming style:

- The criterion *small correctness tests focusing on specific basic categories of input data* (the 2'nd proposed criterion of the attribute *performance of the compiled code*) is included into the evaluation scheme.

- The following inter-attribute partial value function is introduced:

$$v_{final\_style}(A_i) = \begin{cases} v_{style}(A_i), & \text{if } v_8(A_i) = v_8^{max}, \\ 0, & \text{if } v_8(A_i) < v_8^{max}. \end{cases} \tag{6.1}$$

where v_8^{max} is the maximum score for the criterion C_8 (*small correctness tests focusing on basic categories of input data*), $v_8(A_i)$ is the actual score of submission A_i for the same criterion.

Attribute	$Task_1$	$Task_2$	$Task_3$	$Task_4$
Quality of reasoning on design	30	-	-	-
Performance of the compiled code	45	70	70	100
Quality of the programming style	-	30	30	-
Quality of a set of tests	25	-	-	-
TOTAL	100	100	100	100

Table 6.14: The points for each attribute are taken from Table 6.13, and match the chosen template.

Criterion	**Weight** w_j	**Points**
C_1 (clarity)	0.166	5
C_2 (story)	0.166	5
C_3 (knowledge)	0.166	5
C_4 (convincingness)	0.166	5
C_5 (correctness)	0.166	5
C_6 (efficiency)	0.166	5
SUBTOTAL	1	$w^{pt}_{reasoning} = 30$

Table 6.15: Weights of criteria for the attribute *Quality of reasoning on design.* The criteria are taken from Table 6.1. The jury decided not to reveal the scores for separate criteria to the contestants, therefore there was no need to round up the points.

6.2.1.7 List of Criteria Selected for Evaluation of the Quality of Attributes

The next step is to select a set of criteria for each attribute. For the *quality of reasoning on design* and *quality of a set of test cases* we selected all the criteria available in Tables 6.1 and 6.4, i.e. from C_1 to C_6 and from C_{19} to C_{21}.

For the *performance of the compiled code* we selected three criteria: *small correctness tests focusing on specific basic categories of input data* (C_8), *correctness tests focusing on different categories of input data* (C_9), and *efficiency tests sorted into groups to benchmark submissions of different efficiency* (C_{10}, Boolean scale)

For the *quality of programming style* we selected five criteria: *level of consistency* (C_{13}), *layout clarity* (C_{14}), *level of proper naming* (C_{15}), *suitability of substructures* (C_{16}), and *absence of magic numbers* (C_{17}).

6.2.1.8 Weights for the Evaluation Criteria

The contest where we performed piloting was small, therefore we decided to assign the weights to each criterion by discussing them among the three involved jury members. Note that the weights were selected so that the aggregated weights for each attribute would sum up to the corresponding weights in the corresponding task template given in Table 6.13.

The chosen weights for the evaluation criteria are presented in Tables 6.14, 6.15, 6.16, 6.17, and 6.18

Criterion	**Test case**	**Weight (Points)**			
		$Task_1$	$Task_2$	$Task_3$	$Task_4$
C_8 (small)		**0.11(5)**	**0.1 (7)**	**0.05 (5)**	**0.08 (8)**
	e_1	0.6 (3)	0.3 (2)	0.6 (3)	1 (8)
	e_2	0.4 (2)	0.7 (5)	0.4 (2)	
C_9 (correctness)		**0.33 (15)**	**0.35 (24)**	**0.35 (25)**	**0.46 (46)**
	e_1	0.133 (2)	0.6 (14)	0.48 (12)	0.5 (23)
	e_2	0.133 (2)	0.2 (5)	0.52 (13)	0.5 (23)
	e_3	0.133 (2)	0.2 (5)		
	e_4	0.133 (2)			
	e_5	0.133 (2)			
	e_6	0.133 (2)			
	e_7	0.202 (3)			
C_{10} (efficiency)		**0.56 (25)**	**0.55 (39)**	**0.6 (40)**	**0.46 (46)**
	e_1	0.12 (3)	0.15 (5)	0.16 (8)	0.5(23)
	e_2	0.12 (3)	0.15 (6)	0.42 (16)	0.5(23)
	e_3	0.12 (3)	0.35 (14)	0.42 (16)	
	e_4	0.16 (4)	0.35 (14)		
	e_5	0.16 (4)			
	e_6	0.16 (4)			
	e_7	0.16 (4)			
SUBTOTAL		**1(45)**	**1(70)**	**1(70)**	**1(100)**

Table 6.16: Weights of criteria for the attribute *Performance of the compiled code*. The criteria are taken from Table 6.2. The points derived from the weights were rounded.

Criterion	**Weight** w_j	**Points**
C_{13} (consistency)	0.33	9.9
C_{14} (layout)	0.2	6
C_{15} (naming)	0.2	6
C_{16} (substructures)	0.2	6
C_{17} (magic)	0.07	2.1
SUBTOTAL	1	$w_{style}^{pt} = 30$

Table 6.17: Weights of criteria for the attribute *Quality of programming style*. The criteria are taken from Table 6.3. The jury decided not to reveal the scores for separate criteria to the contestants, therefore there was no need to round up the points.

Criterion	Weight w_j	Points
C_{19} (validity)	0.2	5
C_{20} (category)	0.4	10
C_{21} (completeness)	0.4	10
SUBTOTAL	1	$w_{tests}^{pt} = 25$

Table 6.18: Weights of criteria for the attribute *Quality of a set of test cases.* The criteria are taken from Table 6.4.

6.2.1.9 Partial Value Functions for the Evaluation Criteria

Quality of reasoning on design. We have defined the partial value function for each criterion from C_1 to C_6. Therefore, further we assume that $j = 1, \cdots, 6$ (the indices of criteria from C_1 to C_6), $i = 1, \cdots, 14$ (the indexes of submissions A_1 to A_{14}), and $k = 1, \cdots, 3$, where k is the index of the jury member.

The performance of submissions in terms of all the criteria from C_1 to C_6, denoted as $x_j^k(A_i)$, and measured using the linguistic scale taken from Table 3.1. These are the inputs to the partial value function which should perform the following:

1. Convert each linguistic term $x_j^k(A_i)$ into a fuzzy number using Table 3.1.
2. Convert each fuzzy number into a crisp number using Table 3.2.
3. Calculate the aggregated crisp score $v_j^{aggr}(A_i)$ (i.e., taking into account the evaluations of each involved jury member) using Formula 3.15.
4. In order to relate the scores of each criterion to the score of criteria *correctness of design decisions* (C_5), apply the following inter-criterion formula to $v_j^{aggr}(A_i)$, $j \neq 5$:

$$v_j(A_i) = \frac{v_j^{aggr}(A_i) \times v_5^{aggr}(A_i)}{v_{max}} \tag{6.2}$$

   where v_{max} is the maximum possible value of $v_5^{aggr}(A_i)$. v_{max} is equal to the maximum $\mu_T(\widetilde{A})$ value in Table 3.2, i.e. 0.92. For $j = 5$: $v_5(A_i) = v_5^{aggr}(A_i)$.

The score for the whole attribute is calculated using Formula 6.3 and rescaled:

$$v_{reasoning}(A_i) = \frac{\sum_{j=1}^{6} v_j(A_i) w_j}{v_{max}} \times w_{reasoning}^{pt} \tag{6.3}$$

Performance of the compiled code. The performance of submissions in terms of the three criteria from C_8 to C_{10} is measured objectively and the jury is not involved in measuring directly (i.e. only in deciding the weights).

We defined the partial value function for each criterion from C_8 to C_{10}. Therefore, further we assume that $j = 8, \cdots, 10$ (the indices of criteria from C_8 to C_{10}), $i = 1, \cdots, 14$ (the indexes of submissions A_1 to A_{14}).

The performance in terms of each criterion is measured by one or more test cases, and each test case consists of one or more tests. We assume that $e = 1, \cdots E_j$ (E_j is the number of test cases for the criterion j), and $r = 1, \cdots R_e$ (R_e is the number of tests for test case e).

The performance of submissions in terms of all the criteria from C_8 to C_{10}, denoted as $x_j^{er}(A_i)$ is measured using the Boolean scale, i.e., $x_j^{er}(A_i) = 0$ or 1. These are the inputs to the partial value function.

The partial value function should perform the following:

1. Calculate the Boolean score for each test case e:

$$x_j^e(A_i) = \prod_{r=1}^{R_e} x_j^{er}(A_i) \tag{6.4}$$

2. Calculate the rescaled score for each test case e (that is required, because the scores for each test case are made available to the contestants:

$$x_j^{e\_scaled}(A_i) = x_j^e(A_i) \times w_j^{e\_pt} \tag{6.5}$$

3. Calculate the value of the partial value function:

$$v_j(A_i) = \sum_{e=1}^{E_j} x_j^{e\_scaled}(A_i) \tag{6.6}$$

Then the score for the whole attribute is calculated:

$$v_{performance}(A_i) = \sum_{j=8}^{10} v_j(A_i) \tag{6.7}$$

Note that, in this case, the classical WSM was applied, because neither GDM nor linguistic variables were involved.

Quality of the programming style. We defined the partial value function for each criterion from C_{13} to C_{17}. Therefore further we assume that $j = 13, \cdots, 17$ (the indices of criteria from C_{13} to C_{17}), $i = 1, \cdots, 14$ (the indices of submissions A_1 to A_{14}), and $k = 1, \cdots, 3$, where k is the index of a jury member.

The performance of submissions in terms of all the criteria from C_{13} to C_{17}, denoted as $x_j^k(A_i)$, and measured using the linguistic scale, is shown in Table 3.1. These are the inputs to the partial value function which should perform the following:

1. Apply the inter-atribute function defined by Formula 6.1:

$$v_j(A_i) = 0, \text{ if } v_8(A_i) < v_8^{max}, \tag{6.8}$$

   where v_8^{max} is the maximum score for the criterion C_8 (*small correctness tests focusing on the basic categories of input data*).

   The inter-attribute function is applied in the first step, so that the jury would not spend their time on evaluating the quality of programming style if the score is intended to be cancelled.

2. Convert each linguistic term $x_j^k(A_i)$ into a fuzzy number using Table 3.1 for those i values that were not processed by Formula 6.8.

3. Convert each fuzzy number into a crisp number using Table 3.2.

4. Calculate the aggregated crisp score $v_j(A_i)$ (i.e. taking into account the evaluations of each involved jury member) using Formula 3.15.

The score for the whole attribute is calculated and rescaled:

$$v_{style}(A_i) = \frac{\sum_{j=13}^{17} v_j(A_i) w_j}{v_{max}} \times w_{style}^{pt} \tag{6.9}$$

where v_{max} is equal to the maximum $\mu_T(\widetilde{A})$ value in Table 3.2, i.e. $v_{max} = 0.92$.

Quality of a set of test cases. Further we assume that $j = 19, \cdots, 21$ (the indices of criteria from C_{19} to C_{21}), $i = 1, \cdots, 14$ (the indices of submissions from A_1 to A_{14}), $k = 1, \cdots, 3$, $t = 1, \cdots, T_i$, where t is the index of the submitted test and T_i is the total number of tests submitted by A_i.

We define partial value functions for each of the three criteria separately.

- *Validity of a set of test cases* (C_{19})

$$v_{19}(A_i) = \frac{\sum_{t=1}^{T_i} x_t(A_i)}{T_i} \times w_{19} \tag{6.10}$$

  where $x_t(A_i)$ represents the validity of test t submitted by A_i and its value is either 0 or 1. T_i is the total number of test cases submitted by A_i.

- *Belonging to a certain test category* (C_{20})

$$v_{20}(A_i) = \frac{\sum_{t=1}^{T_i} y_t(A_i)}{\times} w_{20} \tag{6.11}$$

  where $y_t(A_i) = 0$ or 1, and it shows whether the test belongs to the test category indicated by the contestant, or not. T_i is the total number of test cases submitted by A_i.

Note that the decision, whether a test belongs to a certain category or not, is made by jury. Nevertheless, we do not apply GDM or linguistic variables. The reason is that we require a Boolean decision and believe that the jury should be unanimous on that.

- Completeness of the whole set of tests (C_{21})

$$v_{21}(A_i) = \frac{g(A_i)}{g_{jury}} \times w_{21} \tag{6.12}$$

where $g(A_i)$ is the number of different test categories provided by a contestant, and g_{jury} is the number of different test categories suggested by the jury.

The aggregated score for the attribute is calculated in the following way:

$$v_{tests}(A_i) = \sum_{j=19}^{21} v_j(A_i) \tag{6.13}$$

The attribute is new and the contestants are not used to it. Therefore we elaborated sample partial value functions which we considered to be good for the first piloting, but which should be gradually updated when the contestants start getting used to the attribute.

6.2.2 Evaluation of Submissions to the Pilot Tasks

For piloting we chose a small contest which was held in the training camp for IOI'2010. 14 contestants and 3 jury members took part in the contest. All the tasks were solved in the same day. Table 6.19 presents data about the submissions that were received for each task.

Quality of reasoning on design. This attribute was evaluated for *Task 1.* We received five submissions to this task and two of them contained reasoning on design. In Table 6.20, we present the results of evaluation in linguistic terms and the corresponding crisp scores which were obtained after using the partial scoring function, defined in Subsection 6.2.1.9

Performance of the compiled code. This attribute was evaluated for all four tasks. In Table 6.21, we present the scores for each of the three criteria and the total score for this attribute for each task.

Quality of the programming style. This attribute was evaluated for $Task_2$ and $Task_3$. We received 23 submissions to those tasks and the style of 5 submissions was not evaluated, because they did not score enough points for the criterion C_8.

In Table 6.22, we present the partial scores for separate criteria and the aggregated scores for the whole attribute of $Task_2$. The scores were obtained after using the partial scoring function, defined in Subsection 6.2.1.9. The scores for $Task_3$ were calculated in a similar way, therefore for this task, we only provide the aggregated score for the whole attribute.

Contestant ID	$Task_1$	$Task_2$	$Task_3$	$Task_4$
A_1		+	+	
A_2		+	+	
A_3			+	+
A_4			+	
A_5		+	+	
A_6		+		+
A_7	+	+	+	+
A_8		+	+	
A_9	+	+	+	+
A_{10}	+		+	+
A_{11}		+	+	+
A_{12}	+	+	+	+
A_{13}	+	+	+	
A_{14}			+	
TOTAL (33)	4	10	13	6

Table 6.19: Data about the submissions for the piloted tasks.

Criteria	A_{12}				A_{13}			
	M_2	M_3	M_{13}	$v_j(A_{12})$	M_2	M_3	M_{13}	$v_j(A_{13})$
C_1 (clarity)	P	P	P	0.10	F	VG	G	0.37
C_2 (story)	P	BFG	P	0.15	G	VG	VG	0.43
C_3 (knowledge)	BPF	F	VP	0.14	F	VG	G	0.37
C_4 (convincingness)	BPV	BPF	P	0.11	BFG	F	BFG	0.29
C_5 (correctness)	BPF	BPF	BPV	0.35	*F*	*P*	*BFG*	0.45
C_6 (efficiency)	BPF	BFG	BPF	0.18	BFG	G	BPF	0.29

Table 6.20: Evaluation of reasoning on design. The columns contain the evaluations of three jury members: M_2, M_3, and M_{13}. The aggregated values calculated at the attribute level are: $v_{reasoning}(A_{12}) = 5.58$, and $v_{reasoning}(A_{13}) = 12$. The opinions of jury members differ significantly on $v_5(A_{13})$ (written in italics). In general, such submissions are reviewed and discussed to find out why the opinions of jury members differ so much.

Quality of a set of test cases. This attribute was evaluated for *Task 1.* We received 4 submissions to those tasks and two submissions contained sets of test cases. In none of the submissions comments were provided about what kind of tests were created. Consequently, we could only apply the criterion *validity of a set of test cases.* The scores are given in Table 6.23.

The final ranking Table 6.24 contains the final ranking of the contestants based on the scores presented and discussed above.

Contes-tant ID	$Task_1$	$Task_2$	$Task_3$	$Task_4$
A_1		7+24+25=56	5+12+0=17	
A_2		7+24+25=56	5+25+24=54	
A_3			5+12+0=17	8+0+0=8
A_4			0+0+0=0	
A_5		7+24+0=31	5+25+0=30	
A_6		7+10+19=36		8+0+0=8
A_7	3+0+0=3	7+24+6=37	5+25+40=70	8+23+0=31
A_8		2+0+0=2	3+0+0=3	
A_9	5+2+3=10	7+0+0=7	5+25+8=38	8+0+0=8
A_{10}			5+12+0=17	8+0+0=8
A_{11}		7+24+39=70	5+25+40=70	8+23+0=31
A_{12}	3+0+0=3	7+0+0=7	3+0+0=3	8+0+0=8
A_{13}	3+0+0=3	7+19+0=26	5+12+8=25	
A_{14}			3+0+0=3	

Table 6.21: Evaluation of performance of the compiled code. The columns contain the evaluations of the three attributes (simple cases, correctness, efficiency) and the aggregated score for the performance of the compiled code.

	$Task_2$						$Task_3$
	$v_{13}(A_i)$	$v_{14}(A_i)$	$v_{15}(A_i)$	$v_{16}(A_i)$	$v_{17}(A_i)$	$v_{style}(A_i)$	$v_{style}(A_i)$
A_1	0.65	0.55	0.78	0.37	0.92	**20.34**	**27.30**
A_2	0.85	0.85	0.70	0.85	0.92	**27.00**	**23.34**
A_3							**23.83**
A_5	0.74	0.76	0.78	0.85	0.92	**25.82**	**30.00**
A_6	0.85	0.55	0.55	0.59	0.50	**21.56**	
A_7	0.59	0.65	0.62	0.74	0.66	**21.13**	**26.62**
A_9	0.48	0.65	0.48	0.75	0.66	**19.01**	**28.89**
A_{10}							**17.85**
A_{11}	0.79	0.79	0.75	0.85	0.92	**26.19**	**23.62**
A_{12}	0.42	0.50	0.56	0.85	0.55	**18.33**	
A_{13}	0.75	0.82	0.75	0.85	0.50	**25.02**	**25.93**

Table 6.22: Evaluation of the quality of programming style. The columns contain partial scores for separate criteria of Task 2, and the aggregated score for the whole attribute of Tasks 2 and 3. Spaces are left if the submission was not delivered

6.2.3 Feedback About the Piloted Evaluation Scheme

The overall conclusion is that piloting of the proposed scheme was successful and can be applied in LitIO. Below we provide the feedback of the jury members who participated in the piloting.

Criteria	A_{10}	A_{12}
No of submitted tests	9	7
No of valid tests	5	6
C_{19} (validity)	3	4
C_{20} (category)	-	-
C_{21} (completeness)	-	-
TOTAL	3	4

Table 6.23: Evaluation of set of test cases. The submitted sets of test cases were not eligible to be evaluated in terms of criteria C_{20} and C_{21}.

No.	**Contestant ID**	$Task_1$	$Task_2$	$Task_3$	$Task_4$	**Total**
1	A_{11}		96.19	93.62	31	**220.81**
2	A_7	3	58.13	96.62	31	**188.75**
3	A_2		83	77.34		**160.34**
4	A_1		76.34	44.3		**120.64**
5	A_{13}	15	51	50.93		**116.93**
6	A_5		56.82	60		**116.82**
7	A_9	10	26.01	66.89	8	**110.9**
8	A_6		57.56		8	**65.56**
9	A_{12}	12.58	25.33	3	8	**48.91**
10	A_3			40.38	8	**48.38**
11	A_{10}	3		34.85	8	**45.85**
12	A_8		2	3		**5**
13	A_{14}			3		**3**
14	A_4			0		**0**

Table 6.24: Aggregate scores of the piloted tasks.

- Even though the description of partial value functions seems to be complicated, in practice the values are calculated automatically and do not add any additional burden to the jury.

- It would be better to have two scales for expressing the scores of manual linguistic evaluation. The first scale should be small in size and used for criteria of a small scope (e.g., C_{17} *absence of magic numbers*). The second scale should be larger and used for criteria of a larger scope (e.g. C_{12}, *Level of quality of programming style*).

- It would be more convenient to use a conversion scale (i.e., a scale that associates linguistic variables with fuzzy numbers) for manual linguistic evaluations that provides full and zero scores. Currently, if all the jury members assign the highest linguistic score, the corresponding crisp value is 0.92, but not 1.

- It is convenient to use submission and attribute templates.
- During piloting (as well as in LitIO) the tasks are assigned equal weights. However, for the tasks where submission includes both an algorithm-code complex and a set of test cases, a higher weight should be considered.
- It is good to apply the GDM method proposed for deciding weights of attributes, important criteria, important inter-criteria partial value functions, because the GDM method ensures that the opinion of each decision maker is taken into account. However, for smaller issues (e.g., weights of test cases within the criterion *small correctness tests* C_8) there is no need to apply MCDA algorithms. The weights can be decided by a few jury members involved.
- The evaluation time increases a lot if many criteria requiring manual evaluation are included into the evaluation scheme. That should be taken into account when composing the evaluation scheme for a concrete task set.
- It might take several years till the contestants and their coaches get used to the new element of a submission: a set of test cases. The partial value function for this attribute will have to be reconsidered for several times, until we arrive at the satisfactory one.

6.2.4 Sensitivity Analysis of the Scores of the Piloted Tasks

In Subsection 3.9, the formula were provided how to calculate the most critical criteria and the most critical measure of performance. The calculated data of sensitivity measures are provided in the following two subsections.

6.2.4.1 The Most Critical Criterion

During the experiment, the contestants solved four tasks. We decided to calculate not only the most critical criterion for each task, but also the most critical criterion for the whole set of tasks. The data are presented in Table 6.25.

	$Task_1$	$Task_2$	$Task_3$	$Task_4$	*Task set*
PT	C_{19}	C_{10}	C_{10}	C_9	C_{10} ($task_2$)
$\delta'_{ji_1i_2}$	-58.96	94.92	18.75	50	97.64
w_j in points	5	39	40	46	39
PA	C_{19}	C_{10}	C_{10}	C_9	C_9 ($task_3$)
$\delta'_{ji_1i_2}$	-58.96	3.47	-86.17	50	-0.248
w_j in points	5	39	40	46	25

Table 6.25: The most critical criterion measures. $\delta'_{ji_1i_2}$ values presented in the table are the signed $|\delta'_{ji_1i_2}|$ value which corresponds to PT and PA indices.

It can be concluded from the data that the best alternative for each task and the best overall alternatives are not very sensitive to the changes of weights of the criteria. The smallest relative change is 18.75 for the weight of the criterion C_{10} in $Task_3$.

	$Task_1$		$Task_2$		$Task_3$		$Task_4$	
	D'_k	$sens(C_k)$	D'_k	$sens(C_k)$	D'_k	$sens(C_k)$	D'_k	$sens(C_k)$
C_8	129.89	0.0076	100	0.01	100	0.01		0
C_9	129.89	0.0076	4.67	0.2141	65.59	0.0152	50	0.02
C_{10}	86.59	0.0115	3.47	0.2882	18.75	0.0539		0
C_{13}			23.38	0.0428	436.83	0.023		
C_{14}			48.17	0.0208	1009.84	0.0010		
C_{15}			43.98	0.0227	598.42	0.0017		
C_{16}			38.91	0.0257	812.21	0.0012		
C_{17}			68.82	0.0145		0		
C_{19}	58.96	0.0169						

Table 6.26: The criticality degree and sensitivity coefficient values for each criterion. The sensitivity coefficient values for non-feasible values of D'_k are equal to 0.

The data about the criticality degree and the sensitivity coefficient of each criterion are presented in Table 6.26. The most sensitive criterion is C_{10} criterion of $Task_2$. Its sensitivity coefficient is 0.2882. In this case, this is the criterion with the highest weight. The same holds for the most sensitive criteria of the other tasks.

It can be concluded that in this pilot contest, the criteria related to the performance of the compiled code are the most important ones. In particular, C_{10} (efficiency tests sorted into groups to benchmark the solutions of different efficiency) and C_9 (correctness tests focusing on different categories of input data).

6.2.4.2 The Most Critical Measure of Performance

The goal of calculating the most critical measure of performance is to analyse how easy it is to change the ranking of alternatives by changing the performances of alternatives. The data about the first and the second tasks are given in Tables 6.27, 6.28. Extensive data about the other two tasks are not provided for the lack of space. Note that the alternatives in each table are ranked in the order of preference *to a particular task*, i.e., $v(F_1) \geq v(F_2) \geq \cdots \geq v(F_n)$. Therefore different notation (F_i instead of A_i) is used.

The most sensitive alternative in $Task_1$ is F_3, the most sensitive alternative in $Task_2$ is F_4. The most sensitive alternative in $Task_3$ is F_1 with $\Delta'_{12} = 7.502$ and $sens(v_{10}(A_1)) = 0.1333$. There are no sensitive alternatives in $Task_4$, i.e., there is no way to change ranking by relatively decreasing or increasing the values of partial value functions. The alternatives that perform equally with all the criteria are not taken into account.

	$Task_1$											
	C_1			C_2			C_3			C_4		
F_2	434	(0.0023)	F_1	289.33	(0.0034)	F_1	310	(0.003)	F_1	394.54	(0.0025)	F_1
	C_5			C_6			C_8			C_9		
F_1	96.44	(0.0103)	F_2				78.62	(0.0127)	F_2			
F_2	124	(0.0080)	F_1	241.11	(0.0041)	F_1	78.62	(0.0127)	F_1			
F_3							51.95	(0.0192)	F_2	129.89	(0.0076)	F_2
	C_{10}			C_{19}								
F_2				58.96	(0.0169)	F_1						
F_3	86.59	(0.011)	F_2									

Table 6.27: The criticality degree Δ'_{ij} (%), the sensitivity coefficients $sens(v_j(A_i))$ (in parentheses), and the corresponding alternative are given for each $v_j(A_i)$ performance measure of $Task_1$. Here F_1, F_2, F_3 represent the alternatives sorted in the order of preference, i.e., $v(F_1) \geq v(F_2) \geq v(F_3)$. Non-feasible values are not represented.

6.3 Conclusions

The goal of this chapter was to reconsider the evaluation scheme, using the MCDA approaches. To accomplish the required stage of the MCDA process, problem structuring, we followed the Goal/Question/Metric framework and invited a group of Lithuanian and international experts to discuss and the evaluation scheme for LitIO. We provided the experts with a description of the Olympiad structure, scope and resources as well as a three-level (submission, attributes, metrics) hierarchical evaluation model pattern to be discussed and filled.

The experts represented two trends that can be distinguished in informatics contests. Some of them associate the quality of a submission with the performance of implementation only, while the others think that a high-quality submission should have both reasoning on design and test data to check the implementation.

The educational goals of LitIO were the key factor in deciding in favour of the latter concept of quality of a solution. Two significant changes were introduced into the current model. The verbal algorithm description which was not related to the implementation and emphasized the correctness and efficiency was replaced by reasoning on the design, which should be related to the implementation and shifted the focus to design issues. The other new aspect was introducing a set of tests as part of a solution.

In the proposed model the concept of a submission is altered. The attributes that need to be measured about the submission are identified, as well as metrics to measure each of the attributes.

During the model analysis stage, the evaluation scheme proposed was piloted in a small contest. A procedure was elaborated and demonstrated how to adjust the evaluation scheme to concrete tasks and how to apply it in practice. The piloting

	$Task_2$											
	C_8			C_9			C_{10}			C_{13}		
F_1				55.37	(0.0181)	F_2	34.07	(0.0293)	F_2			
F_2	95.96	(0.0104)	F_3	27.99	(0.0357)	F_3	26.87	(0.0372)	F_3	73.44	(0.0136)	F_3
F_3	95.6	(0.0104)	F_2	27.99	(0.0357)	F_2	26.87	(0.0372)	F_2	96.04	(0.0104)	F_2
F_4	9.35	(0.0170)	F_5	2.73	(0.3668)	F_5	10.91	(0.0917)	F_5	10.31	(0.0970)	F_5
F_5	9.35	(0.0170)	F_4	6.54	(0.1528)	F_4	3.44	(0.2904)	F_4	7.15	(0.1398)	F_4
F_6	9.43	(0.1061)	F_5	2.75	(0.3638)	F_5				8.29	(0.1207)	F_5
F_7	80.79	(0.0124)	F_6	29.77	(0.336)	F_6				70.07	(0.0143)	F_6
F_8	10.02	(0.0998)	F_9							13.57	(0.0737)	F_9
F_9	10.02	(0.0998)	F_8							15.51	(0.0645)	F_8
F_{10}	1161.58	(0.009)	F_9									

	C_{14}			C_{15}			C_{16}			C_{17}		
F_1												
F_2	239.73	(0.0042)	F_1	291.10	(0.0034)	F_1	239.73	(0.0042)	F_1	632.82	(0.0016)	F_1
F_3	187.27	(0.0053)	F_2	132.05	(0.0076)	F_2	278.38	(0.0036)	F_2	319.88	(0.0031)	F_2
F_4	15.44	(0.0648)	F_5	16.18	(0.0618)	F_5	13.56	(0.0738)	F_5	43.43	(0.0230)	F_5
F_5	18.24	(0.0548)	F_4	18.24	(0.0548)	F_4	17.01	(0.0588)	F_4	57.33	(0.0174)	F_4
F_6	13.31	(0.0751)	F_5	12.97	(0.0771)	F_5	11.90	(0.0840)	F_5	31.42	(0.0318)	F_5
F_7	105.75	(0.0095)	F_6	115.62	(0.0086)	F_6	102.02	(0.0098)	F_6	495.52	(0.0020)	F_6
F_8	16.54	(0.0605)	F_9	22.40	(0.0447)	F_9	14.33	(0.0698)	F_9	46.54	(0.0215)	F_9
F_9	21.50	(0.0465)	F_8	19.20	(0.0521)	F_8						
F_{10}	12.65	(0.0791)	F_8	55.84	(0.0179)	F_8						

Table 6.28: The criticality degree Δ'_{ij} (%), the sensitivity coefficients $sens(v_j(A_i))$ (in the parentheses) and the corresponding alternatives are given to each $v_j(A_i)$ performance measure for $Task_1$. Here F_1, F_2, $\cdots$, F_{10} represent the alternatives sorted in the order of preference, i.e., $v(F_1) \geq v(F_2) \geq \cdots \geq v(F_{10})$. Non-feasible values are not represented.

confirmed that the proposed evaluation scheme is suitable to be applied in LitIO. The feedback with suggestions for improvement, especially about the scheme adjustment process was received. We have calculated the sensitivity measures. They have showed that the winner is strong, i.e., the score of the contest winner was not very sensitive either in terms of criteria weights or in terms of performances of other contestants. In general, the sensitivity coefficient values were not high, which is positive and implies that the ranking could not be easily changed.

7 Conclusions

1. The evalutation scheme currently applied in LitIO should be improved.

   The practice of evaluating the solutions to programming assignments in undergraduate courses in many cases is not applicable in informatics contests, because of different goals, different task complexity, and different evaluation methods.

2. The tasks with graphs are suitable for a semi-automated evaluation conducted by applying visualisation of the graphs, implemented in the algorithm-code complexes.

3. The attributes that describe the quality of an algorithm-code complex and are included into the current evaluation scheme are valid in terms of the ISO-9126-1 quality model. The precision of measurements of the quality of algorithm-code complexes, obtained by using black-box testing, is 80% or more depending upon the scoring function.

4. The evaluation scheme, developed applying the MCDA methods is suitable for LitIO.

A Appendices

A.1 *Nescafé Algebra*, Sample of a Batch Task

The whole text of problem *Nescafé Algebra* was presented in the first exam session in the finals of LitIO'2008 for senior division contestants. Constraints on this task are given in the input section.

Task story. Linas is studying informatics and even though he says he enjoys mornings, most of his work he completes at the night time. To avoid falling asleep during the night, he... drinks «coffee».

As many IT people, Linas is rather lazy and therefore he got used to buying instant "Nescafé" coffee, because it is very easy to prepare it. There are different sorts of "Nescafé". For example, the «2 in 1» packet contains coffee and milk powder while the «3 in 1» packet there includes sugar as well.

"«2 in 1», «3 in 1»... Hmm, if you mix them both, would you get «5 in 2» coffee?", – enjoyed Linas! And being full of inspiration he continued reasoning in a similar way:

"Suppose we have coffee mixes «p in r» and «s in t». If we mix those two, the resulting super-mix will be «$(p+s)$ in $(r+t)$». Which of the following N packages of coffee «a_1 in 1», «a_2 in 1», $\cdots$ «a_N in 1» we need to mix in order to get a super-mix «b in c»?"

Task. Write a program to solve the task.

Input. The input data are presented in the file *kava.in*. The first line contains three integers b, c, and N. The following N lines contain numbers a_1, a_2, ..., a_N that describe the corresponding coffee mixes. The numbers are written one number per line. All the input numbers are positive.

The following constraints are valid: $a_i \le 1,000$, $b \le 10,000$, $c \le N \le 100$.

Output. Your program should output a solution to the file *kava.out*. The solution consists of the numbers of mixes that have to be mixed in order to get a super-mix «b in c». The numbers of mixes can be presented in any order, one number per line. Each mix can be used only once for obtaining the super-mix.

If there is no way to make a super-mix «b in c», then the only line of the output file should contain zero (0). In the case of several possible solutions, the program should output any of them.

Examples are provided in Table A.1.

Input	Output	Comments

5 2 3
2
3
1
	1
	2
		If we mix the content of the first and the second packet, we will get a super-mix «5 in 2» The solution could be written in the inverse order: first 2 and then 1.

6 2 3
2
3
1
	0	It is only possible to get a super-mix «6 in 3», but there is no way to get a super-mix «6 in 2».

12 3 6
6
7
5
2
3
2
	2
	4
	5
		Other possible solution would be to mix the content of the second, fifth and the sixth packages because the sixth and the fourth packages are identical. The program can output any solution

Table A.1: Sample tests for *Nescafé Algebra*

A.2 Material and Questionnaire Distributed to the Experts

I am inviting you to be one of the experts in the investigation, as part of my PhD research. The research concerns grading in the (Lithuanian) Informatics Olympiads, which are contests in algorithmic problem solving for high school students described below.

The contest has several objectives. However, in this research, the most important contest goals are the challenge in algorithmic problem solving and the educational aspect of the contest, i.e., to forward the message to the contestants that the solutions should be designed in conformance with the academic standards (as much as it is possible under the contest pressure).

The purpose of my research is to develop a framework for producing motivated grading schemes for contest tasks. Such a grading scheme is applied to the contestants' submissions in order to determine a score that reflects their abilities. The framework defines the aspects that are taken into account and how they are measured.

The current framework for grading schemes (described below) lacks transparency and a scientific foundation.

The main work I am asking you to do is to propose the list of aspects to be measured and the corresponding measures.

Further I explain a few concepts used in the research.

Submission. [1] Currently, the submission consists of a verbal description of the algorithm and the implementation of the algorithm (not necessarily the same as in the verbal description) presented in the form of a compilable source code (in Pascal/C/C++).

The concept of submission is not status quo and can be altered, if there is a motivation for that (e.g., if an important aspect that can be used to determine a score requires alterations in the understanding of a submission)

Quality of a submission. The quality is understood as a constructed notion, i.e., determined by the decisions of experts to choose metrics.

Metric. A metric consists of:

1. The domain (input set) of the metric; the broadest possible domain is a submission; if the metric applies to some part of a submission, it should be indicated which part is that.
2. The range (output set) of the metric (a scale). The scale is expected to be either a ratio scale where the measurement is expressed as an integer from the interval [0 to 100], or an ordinal scale.
3. The description of the corresponding output for a valid input that describes how to determine the metric's value for a given submission (this could be given as an algorithm, i.e. recipe)
4. A motivation for why 1, 2 and 3 are a valid way of measuring this attribute.

I would like to ask you as an expert to answer the following questions:

- What attributes of a submission are most relevant for determining a score and can be objectively measured? You can restrict yourself to what you consider the not more than five most relevant attributes.
- What metrics would you propose to measure these attributes (more than one metrics could be used to measure each attribute). Define each metric as precisely as you can.
- How would you suggest to implement each metric (taking into account the resources and limitations described below). The metrics can be implemented by a manual measurement procedure, or by an automated procedure, or by their combination. Describe each procedure as precisely as you can.

[1] The concept of *submission* (i.e. material presented for evaluation) is unambiguously understood in the community of informatics contests and there was no need to repeat it here. The purpose of explaining it here was to provide the components of a submission in LitIO.

- How would you suggest to integrate the separate metrics to get one score (for one submission).

DESCRIPTION OF THE CONTEST *(Lithuanian Informatics Olympiad at the national level) and its resources*
There are two divisions (junior and senior) in LitIO. The national contest consists of two parts. The first part is an on-line round which lasts 5 hours. The second part is a face-to-face contest with two consecutive contest days, 4-5 hours each round.

During one 4-5 hour round the contestants have to solve two-three tasks in each division. Altogether, the organizers have to provide from 10 to 15 contest tasks. Some of the tasks may overlap.

The number of contestants varies from 50 (on-site contest) to 300 (on-line contest). On average there are 5-8 jury members available for task design and grading. They agree to spend up to 3-5 working days for preparing one task for senior students and up to 2-3 days for preparing one task for junior students. Those who have more time available are preparing more than one task.

The time available for grading is from 4-5 hours (on-site contest) to two working days (on-line contest). In case of on-line contest, the scores have to be produced within two weeks. However, no member of the jury will agree to spend more than a couple of full working days on grading.

The technical resources are 6-8 grading machines for the on-line contest (more are available on request) and one grading machine for the final round.

DESCRIPTION OF THE CURRENT GRADING PROCEDURE

Each algorithmic task is assigned the same number of points (maximum 100) independently of its difficulty. The points are distributed over the verbal description of an algorithm (0-20% of points), testing results (70-100%) and the programming style (0-10%).

The algorithm description is to be written in natural language and is graded independently whether the program is provided or not. There is no requirement for the description to match the implementation. The program performance (correctness and efficiency) is tested by running the program with set of tests designed in advance and not disclosed to the contestants until after the contest. Tests are grouped, each group targeting a specific goal. The points for the tests in a group are only given if all the tests in the group are passed successfully.

The programming style is graded only if the program scores 50% or more points for the tests.

References

(1990). *IEEE Standard Glossary of Software Engineering Terminology.* Report IEEE Std 610.12-1990, IEEE. 105

(2001). *ISO 9126-1:2001, Software engineering - Product Quality, Part 1: Quality model.* ISO, International Organization for Standardization. `http://www.sqa.net/iso9126.html`. 89

(2002). IOI'2002 manual. `http://www.ioi2002.or.kr/eng/PracticeCompetitionMaterial/ContestSystemManual.pdf`. 17, 29

(2002). *Value Tree Analysis.* Multiple Criteria Decision Analysis E-learning site created in the EU project ORWorld by System Analysis Laboratory of Helsinki University of Technology. `http://www.mcda.hut.fi/value_tree/theory/`. 48, 50, 53, 54, 55

(2004). IOI'2004 material. `http://olympiads.win.tue.nl/ioi/ioi2004/surveys/contestants.html`. 76

(2006). Perspectives on computer science competitions for (high school) students. In *Workshop*, Dagstuhl, Germany. `http://www.bwinf.de/competition-workshop/`. 8

(2007). 2007 m. informacinių technologijų valstybinio brandos egzamino rezultatų analizė. `http://www.nec.lt/failai/339_rez_analize_2007_VBE_statistine_IT.pdf`. Nacionalinis egzaminų centras. 96

(2007). *Software Quality ISO Standards.* Arisa - Controlling Software Quality. `http://www.arisa.se/compendium/node6.html`. 89

(2007). SWOT analysis. Strategic management. `http://www.quickmba.com/strategy/swot/`. 55

(2008). IOI regulations. `http://ioinformatics.org/rules/reg08.pdf`. 12

(2008). ISO/IEC 12207:2008 Software life cycle processes. In *Software Engineering Process technology (SEPT).* `http://www.12207.com`. 86

(2008). Regulations of Lithuanian Informatics Olympiad. `http://www.lmitkc.lt/lt/informatikos`. (in Lithuanian). 12

(2009). Waterfall lifecycle model, softdevteam. `http://www.softdevteam.com/waterfall-lifecycle.asp`. 86

(2010a). ACM-ICPC International Collegiate Programming Contest. `http://cm.baylor.edu`. xiii, 9, 40, 119

(2010b). The ACM-ICPC World Finals 2010. `http://www.cs.helsinki.fi/en/news/761`. 10

(2010). Bundeswettbewerb informatik. `http://www.bwinf.de/`. 44, 116

(2010). *CourseMarker. Automatic Marking and Feedback for Students and Teachers.* School of Computer Science and IT, The University of Nottingham, UK. `http://www.cs.nott.ac.uk/~cmp/cm_com/index.html`. 28

(2010). Dr. Juozo Petro Kazicko moksleivių kompiuterininkų forumas. `http://forumas.ktu.lt`. 13

(2010). IOI – International Olympiad in Informatics. `http://www.ioinformatics.org`. xiv, 9

(2010). ISO 9126 Software Quality Characteristics. `http://www.sqa.net/iso9126.html`. 89, 105

(2010). *Item analysis.* `http://fcit.usf.edu/assessment/selected/responsec.html`. 13

(2010). *Item discrimination.* Website of University of Wisconsin Oshkosh, USA. `http://www.uwosh.edu/testing/facultyinfo/itemdiscrimone.php`. 96

(2010). Lithuanian Olympiads in Informatics. `http://ims.mii.lt/olimp`. (in Lithuanian). xiv, 125

(2010). Scientific Committee of Lithuanian Informatics Olympiads. 51

(2010). SWOT analysis: Lesson. `http://marketingteacher.com/Lessons/lesson_swot.htm`. 55

(2010). TopCoder algorithm competition. `http://www.topcoder.com/tc`. 10, 114

Ahoniemi, T. and Reinikainen, T. (2006). ALOHA - a grading tool for semi-automatic assessment of mass programming courses. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 139–140, Uppsala, Sweden. 9, 45, 46

Ala-Mutka, K. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102. 21, 28, 31, 36

Ala-Mutka, K., Uimonen, T., and Jarvinen, H. M. (2004). Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3:245–262. 33, 35

Allowatt, A. and Edwards, S. (2005). IDE support for test-driven development and automated grading in both Java and C++. In *OOPSLA'05 Eclipse Technology exchange (ETX) Workshop*, San Diego, California, USA. 36

Amelung, M., Piotrowski, M., and Rosner, D. (2006). EduComponents: experiences in e-assessment in computer science education. *ACM SIGCSE Bulletin*, 38(3):1–5. 30

Andrianoff, S. K. and Hunkins, D. R. (2004). Adding objects to the traditional ACM programming contest. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. 38, 105

Anido, R. O. and Menderico, R. M. (2007). Brazilian olympiad in informatics. *Olympiads in Informatics*, 1:5–14. 9

Astrachan, O. L., Khera, V., and Klotz, D. (1993). The internet programming contest: Report and philosophy. *ACM SIGCSE Bulletin*, 25(1):48–52. 12

Basili, V. R., Caldiera, C., and Rombach, H. D. (1994). Goal Question Metric paradigm. *Encyclopaedia of Software Engineering, John Wiley & Sons*, 1:528–532. 56

Basili, V. R. and Weiss, D. M. (1984). A methodology for collecting valid software engineering data. *IEEE Transactions, Software Engineering*, SE-10(6):728–738. 56

Belton, V. and Stewart, T. J. (2003). *Multiple Criteria Decision Analysis: An Integrated Approach*. Kluwer Academic Publishing, Boston. 48, 52, 53, 54, 57, 58, 59, 61

Benford, S. D., Burke, E. K., Foxley, E., and Higgins, C. A. (1995). The Ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd annual on Southeast regional conference*, pages 176–182. 28, 34

Benson, M. (1985). Machine assisted marking of programming assignments. *ACM SIGCSE Bulletin*, 17(3):24–25. 29, 30

Berry, R. E. and Meekings, B. A. E. (1985). A style analysis of C programs. *Communications of the ACM*, 28(1):80–88. 34

Blonskis, J. and Dagienė, V. (2006). Evolution of informatics maturity exams and challenge for learning programming. *Informatics Education – The Bridge between Using and Understanding Computers (Editor: R. T.Mittermeir), Lecture Notes in Computer Science*, 4226:220–229. 115

Blonskis, J. and Dagienė, V. (2008). Analysis of students' developed programs at the maturity exams in information technologies. In Mittermeir, R. T. and Syslo, M. M., editors, *Lecture Notes in Computer Science, Informatics Education – Supporting Computational Thinking*, volume 5090, pages 204–215. 12

Boersen, R. and Phillips, M. (2006). Programming contests: two innovative models from new Zealand. In *Perspectives on Computer Science Competitions for (High School) Students, Workshop, Dagstuhl, Germany*. `http://www.bwinf.de/competition-workshop/Submissions/1_BoersenPhillipps.pdf`. 14, 41

Bowring, J. F. (2008). A new paradigm for programming competitions. *ACM SIGCSE Bulletin*, 24(1):87–91. 13, 38

Brassard, G. and Bratley, P. (1996). *Fundamentals of algorithms*. Prentice Hall. 99

Bryson, D. and Roth, R. W. (1981). Programming contest. topics. In *Computer Education for Elementary and Secondary Schools*, pages 60–65. Joint Issue Education Board of ACM. 8

Burn, O. (2003). CheckStyle. *SourceForge.net*. `http://checkstyle.sourceforge.net/`. 34

Burton, B. (2007). Informatics olympiads: Approaching mathematics through code. *To appear in Mathematics Competitions*. 18, 19

Burton, B. A. (2008). Breaking the routine: Events to complement informatics olympiad training. *Olympiads in Informatics*, 2:5–15. 38

Califf, M. E. and Goodwin, M. (2002). Testing skills and knowledge: introducing a laboratory exam in CS1. *ACM SIGCSE Bulletin*, 34(1):217–221. 31

Carlsson, C. and Fullér, R. (1996). Fuzzy multiple criteria decision making: recent developments. *Fuzzy Sets and Systems*, 78(2):139–153. 59, 70

Carter, J., Ala-Mutka, K., Fuller, U., Dick, M., English, J., Fone, W., and Sheard, J. (2003). How shall we assess this? *ACM SIGCSE Bulletin*, 35(4):107–123. 30

Chen, P. (2004). An automated feedback system for computer organization projects. *IEEE Transactions on Education*, 47:232–240. 36

Chen, S. J., Hwang, C. L., and Hwang, F. P. (1992). Fuzzy multiple attribute decision making: Methods and applications. In *Lecture Notes in Economics and Mathematical Systems*, volume 375. Springer-Verlag, Berlin, Germany. 59, 66, 67, 68, 69, 71, 126, 127

Chou, S. Y., Chang, Y. H., and Shen, C. Y. (2007). A fuzzy simple additive weighting system under group decision-making for facility location selection with objective/subjective attributes. *European Journal of Operational Research*. 125

Colton, D., Fife, L., and Thompson, A. (2006). A web-based automatic program grader. In *Proceedings of ISECON, the Conference for Information Systems Educators*, volume 23, pages 1–9, Dallas, USA. 28, 32, 38

Comer, J. R., Wier, J. R., and Rinewalt, J. R. (1983). Programming contests. In *Proceedings of the fourteenth SIGCSE technical symposium on Computer science education*, pages 241–244. 8, 9

Cormack, G. (2006). Random factors in IOI 2005 test case scoring. *Informatics in Education*, 5(1):5–14. 42, 114

Cormack, G., Kemkes, G., Munro, I., and Vasiga, T. (2006). Structure, scoring and purpose of computing competitions. *Informatics in Education*, 5(1):15–36. 8, 12, 13, 14, 17, 41, 42, 44

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (1992). *Introduction to Algorithms*. MIT Press and McGraw-Hill. xiii, xiv, 10

Crosby, P. B. (1979). *Quality is free: the art of making quality certain*. McGraw-Hill, New York. 20

Csáki, P., Rapcsák, T., Turchányi, P., and Vermes, M. (1995). Research and development for group decision aid in Hungary by WINGDSS, a Microsoft Windows based group decision support system. *Decision Support Systems*, 14:205–221. 71

Dagienė, V. (2008). Teaching information technology and elements of informatics in lower secondary schools: curricula, didactic provision and implementation. In Mittermeir, R. T. and Syslo, M. M., editors, *Lecture Notes in Computer Science, Informatics Education – Supporting Computational Thinking*, volume 5090, pages 293–304. 12

Dagienė, V. and Skūpienė, J. (2003). Analysis of tasks in Lithuanian Informatics Olympiad by type of solution and level of difficulty. *Lithuanian Mathematical Journal*, 43:209–214. (In Lithuanian). 19, 76

Dagienė, V. and Skūpienė, J. (2007). Contests in programming: Quarter century of Lithuanian experience. *Olympiads in informatics*, 1:37–49. 16, 49, 95

Daly, C. and Waldron, J. (2004). Assessing the assessment of programming ability. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 210–213. 35

Daniels, M., Berglund, A., Pears, A., and Fincher, S. (2005). Five myths of assessment. In *ACE'04: Proceedings of the sixth conference on Australasian computing education*, pages 57–61, Darlinghurst, Australia. 45

Deimel, L. (1984). 1984 ACM International Scholastic Programming Contest. *ACM SIGCSE Bulletin*, 6(3):7–12. 8, 17

Deimel, L. (1988). Problems from the 12'th Annual ACM Scholastic Programming Contest. *ACM SIGCSE Bulletin*, 20(4):19–28. 8, 37

Demetrescu, C., Finocchi, I., and Stasko, J. (2002). Specifying algorithm visualizations: Interesting events or state mapping? In *Software Visualization*, pages 16–30. Springer-Verlag, Berlin, Heidelberg. 77

Deming, W. E. (1988). *Out of the crisis: quality, productivity and competitive position.* Cambridge Univ. Press. 20

Diehl, S., Görg, C., and Kerren, A. (2002). Animating algorithms live and post mortem. *Software Visualization*, LNCS 2269:46–57. 81

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM, ACM Turing Lecture*, 15(10):859–866. 31, 37, 39

Diks, K., Kubica, M., and Stencel, K. (2007). Polish olympiads in informatics – 14 years of experience. *Olympiads in Informatics*, 1:50–56. 37, 43

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *ACM Journal of Educational Resources in Computing*, 5(3):1–13. 9, 28, 29, 30, 33

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering*, 21:146–162. 34

Eden, C. (1988). Cognitive mapping: a review. *European Journal of Operational Research*, 36:1–13. 55

Eden, C. and Ackermann, F. (1998). Making strategy: the journey of strategic management. In *SAGE Publications*, London. 55

Eden, C. and Simpson, P. (1989). *SODA and cognitive mapping in practice.* Rosenhead. 55

Edwards, S. H. (2003). Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 148–155, Anaheim, CA, USA. 29, 36

Ernst, F., Moelands, J., and Pieterse, S. (2000). Teamwork in programming contests: 3*1=4. *Crossroads, The ACM student magazine.* 37

Fisher, M. and Cox, A. (2006). Gender and programming contests: Mitigating exclusionary practices. *Informatics in Education*, 5(1):47–62. 41

Fitzgerald, S. and Hines, M. L. (1996). The computer science fair: an alternative to the computer programming contest. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 368–372. 38

Forišek, M. (2006). On the suitability of tasks for automated evaluation. *Informatics in Education*, 5(1):63–76. 37, 38, 39, 40, 42, 49, 93, 99, 107

Forsythe, G. E. and Wirth, N. (1965). Automatic grading programs. *Communications of the ACM*, 8(5):275–278. 28

Foxley, E., Higgins, C., Hrgazy, T., Symeonidis, P., and Tsintisfas, A. (2004). *The CourseMaster CBA System: Improvements over Ceilidh.* The University of Nottingham, School of Computer Science and IT. `www.cs.nott.ac.uk/~cah/pdf/EIT_ImprovementsOverCeilidh_Paper.pdf`. 9, 32

French, S. (1988). *Decision Theory: an Introduction to the Mathematics of Rationality.* Ellis Horwood, Chichester. 60

Grigas, G. (1995). Investigation of the relationship between program correctness and programming style. *Informatica*, 6(3):265–276. 33, 35, 107, 108, 116

Hansen, H. and Ruuska, M. (2003). Assessing time-efficiency in a course on data structures and algorithms. In *Koli Calling 2003, 3rd Annual Finnish/Baltic Sea Conference on Computer Science Education*, pages 86–93. 32

Harris, J. A., Adams, E. S., and Harris, N. L. (2004). Making program grading easier: but not totally automatic. *Journal of Computing Sciences in Colleges*, 20(1):248–261. 9

Hellmann, M. (2001). *Fuzzy Logic Introduction.* The University of Rennes, France. `http://epsilon.nought.de/`. 63

Helmick, M. T. (2007). Interfacebased programming assignments and automatic grading of java programs. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 63–67, Dundee, Scotland. 9, 42

Hext, J. B. and Winings, J. W. (1969). An automatic grading scheme for simple programming exercises. *Communications of the ACM*, 12(5):272–275. 28

Hill, T. and Westbrook, R. (1998). SWOT analysis: It's time for a product recall. *Long Range Planning*, 30(1):46–52. 55

Hirch, B. and Heines, J. M. (2005). Automated evaluation of source code documentation: Interim report. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 1–5, S. Louis, Missouri, USA. 33, 35

Hollingsworth, J. (1960). Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529. 28

Horvath, G. and Verhoeff, T. (2002). Finding the Median under IOI conditions. *Informatics in Education*, 1(1):73–92. 104

Hoyer, R. W. and Hoyer, B. B. I. (2001). What is quality? *Quality Progress*, 7:52–62. 19, 20

Jackson, D. (2000). A semi-automated approach to online assessment. In *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 164–167, Helsinki, Finland. 21, 33

Jackson, D. and Usher, M. (1997). Grading student programs using ASSYST. *ACM SIGCSE Bulletin*, 29(1):335–339. 32, 34, 36

Joy, M. and Luck, M. (1995). On-line submission and testing of programming assignments. In *Innovations in Computing Teaching, SEDA*, London. 28

Kahraman, C. (2008). *Fuzzy Multi Criteria Decision Making. Theory and Applications with Recent Developments.*, volume 16 of *Optimization and its applications.* Springer. 59

Kauffman, B., Bettge, T., Buja, L., Craig, T., DeLuca, C., Eaton, B., Hecht, M., Kluzek, E., Rosinski, J., and Vertenstein, M. (2001). Community climate system model. software developer's guide. `http://www.ccsm.ucar.edu/working_groups/Software/dev_guide/dev_guide/dev_guide.html`. 86

Kearse, I. B. and Hardnett, C. R. (2008). Computer science olympiad: Exploring computer science through competition. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 92–96, Portland, USA. 12

Keeney, R. L. and Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs.* John Wiley & Sons. 48, 55, 59, 60

Keeney, R. R. (1992). *Value-Focused Thinking: A Path to Creative Decision making.* Harvard University Press, Cambridge, Massachusetts, London, England. 55

Kemkes, G., Cormack, G., Munro, I., and Vasiga, T. (2007). New task types at the Canadian Computing Competition. *Olympiads in Informatics*, 1:79–89. 19

Kemkes, G., Vasiga, T., and Cormack, G. (2006). Objective scoring for computing competition tasks. In *Proceedings of International Conference in Informatics in Secondary Schools – Evolution and Perspectives*, Lecture Notes in Computer Science, pages 230–241. Springer-Verlag. 13, 28, 36, 37, 38, 39, 40, 41, 42, 43, 97

Kernighan, B. W. and Pike, R. (1999). *The Practice of Programming.* Addison-Wesley. 27, 33, 110

Kerren, A. and Stasko, J. T. (2002). Algorithm animation. *Software Visualization*, LNCS 2269:1–15. 77

Khuri, S. and Holapfel, K. (2001). EVEGA: An Educational Visualization Environment for Graph Algorithms. In *Proceedings of the 6th Annual Conference on Innovaton and Technology in Computer Science Education.* ACM Press. 78

Kiryukhin, V. M. (2007). The modern contents of the Russian National Olympiads in Informatics. *Olympiads in Informatics*, 1:90–104. 17

Kolstad, R. and Piele, D. (2007). USA Computing Olympiad (USACO). *Olympiads in Informatics*, 1:105–111. 9, 12

Laarhoven, P. J. M. and Pedrycz, W. (1983). A fuzzy extension of saaty;s priority theory. *Fuzzy Sets and Systems*, 11:229–241. 65

Land, R. (2002). Measurements of software maintainability. In *Proceedings of ARTES (A network for Real-Time research and graduate Education in Sweden)'2002 Conference*, pages 23–27. http://www.artes.uu.se/events/gsconf02/papers/ARTESproceeding2002.pdf. 105

Leal, P. J. and Moreira, N. (1998). Automatic grading of programming exercises. Technical report series: Dcc-98-4, University of Porto, Portugal, Department of Computer Science. www.ncc.up.pt/~nam/publica/dcc-98-4.ps.gz. 31, 33, 34, 37

Lee, K. H. (2005). *First Course on Fuzzy Theory and Applications*. Springer. 63, 64

Leeuwen, W. T. V. (2005). A critical analysis of the ioi grading process with an application of algorithm taxonomies. *Master's Thesis, Technische Universiteit Eindhoven, Faculty of Mathematics and Computing Science*. http://alexandria.tue.nl/extra1/afstversl/wsk-i/leeuwen2005.pdf. 37, 38, 92, 104

Lewis, S. and Davies, P. (2004). The automated peer-assisted assessment of programming skills. In *Proceedings of the 8th JAVA & The Internet in the Computing Curriculum Conference JICC8*, pages 1–10. 30

Li, D. F. and Yang, J. B. (2004). Fuzzy linear programming technique for multiattribute group decision making in fuzzy environments. *Information Sciences*, 158:263–275. 48

Lootsma, F. A. (1997). *Fuzzy logic for planning and decision making*. Kluwer. 62

Lu, J., Zhang, G., and Ruan, D. (2007). *Multi-Objective Group Decision Making: Methods, Software and Application with Fuzzy Set Techniques*. Series in Electrical and Computer Engineering. 70, 71, 126, 127, 132

Lundberg, L., Mattson, M., and Wohlin, C. (2005). *Software Quality Attributes and Trade-offs*. BESQ - Blekinge Engineering Software Qualities, Blekinge Institute of Technology. http://sea-mist.se/tek/besq.nsf/pages/017bd879b7c9165dc12570680047aae2!OpenDocument. 20

Malmi, L., Korhonen, A., and Saikkonen., R. (2002). Experiences in automatic assessment on mass courses and issues for designing virtual courses. *ACM SIGCSE Bulletin*, 34(3):55–59. 30

Manev, K. (2008). Tasks on graphs. *Olympiads in Informatics*, 2:90–104. 76

Mareš, M. (2007). Perspectives on grading systems. *Olympiads in Informatics*, 1:124–130. 17, 29

McCall, J. A., Richards, P. K., and Walters, G. F. (1977). Factors in software quality. *Vol. I-III, Rome Air Development Center, Italy*. 89

Metzner, J. R. (1983). Proportional advancement from regional programming contests. *ACM SIGCSE Bulletin*, 15(3):27–30. 8

Miara, R. J., Musselman, J. A., Navarro, J. A., and Shneiderman, B. (1983). Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867. 27

Miller, C. A. (1956). The magic number seven plus or minus two: some limits on our capacity for processing information. *Psychological review*, 13:81–97. 126

Mohan, A. and Gold, N. (2004). Programming style changes in evolving source code. In *IEEE Proceedings of the 12th International Workshop on Program Comprehension*, pages 236–240, Italy. 33, 106

Myers, D. and Null, L. (1986). Design and implementation of a programming contest for high school students. In *Proceedings of the seventeenth SIGCSE technical symposium on Computer science education*, pages 307–312. 13, 41, 119

Oberti, P. (2004). Décision publique et recherche procédurale: illustration d'une démarche multicritère à la localisation participative d'un parc éolien en région corse. In *Actes des journées de l'Association Française de Science Economique, Economie: aide à la dècision publique*. Université de Rennes. http://crereg.eco.univ-rennes1.fr/afse/TEXTES-PAR-SESS/A2/OBERTI.P.75.pdf. 53

Oman, P. and Cook, C. (1990). A taxonomy for programming style. In *18th ACM Computer Science Conference Proceedings*, pages 244–250. 34, 106

Opmanis, M. (2006). Some ways to improve olympiads in informatics. *Informatics in Education*, 5(1):113–124. 42, 93, 114

Pankov, P. S. and Okruskulov, T. R. (2007). Tasks at Kyrgysztani olympiads in informatics: Experience and proposals. *Olympiads in Informatics*, 1:131–140. 44

Pardo, A. (2002). A multi-agent platform for automatic assignment management. *ACM SIGCSE Bulletin*, 34(3):60–64. 30

Patterson, A. F. (2005). Reflections on a programming olympiad. *Communications of the ACM*, 48(7):15–16. 9, 12

Pohl, W. (2004). National computer science contests. `http://www.bwinf.de/olympiade/national-contests.pdf`. 44

Pohl, W. (2006). Computer science contests for secondary school students: Approaches for classification. *Informatics in Education*, 5(1):125–132. 8, 14, 16

Pohl, W. (2007). Computer science contests in Germany. *Olympiads in Informatics*, 1:141–148. 19, 44

Pohl, W. (2008). Manual grading in an informatics contest. *Olympiads in Informatics*, 2:122–130. 37, 38, 39, 44, 46, 114

Pohl, W. and Polley, T. (2006). Experiences with graduated difficulty in programming contest problems. In *Information Technologies at School. Proceedings of the Second International Conference Informatics in Secondary Schools: Evolution and Perspectives*, pages 499–508, Vilnius. TEV. 12, 14

Poranen, T., Dagiene, V., Eldhuset, A., Hyro, H., Kubica, M., Laaksonen, A., Opmanis, M., Pohl, W., Skupiene, J., Soderhjelm, P., and Truu, A. (2009). Baltic Olympiads in Informatics: Challenges for training together. *Olympiads in Informatics*, 3:112–131. 19, 76

Rahman, K. A., Ahmad, S., and Nordin, J. (2007). Interfacebased programming assignments and automatic grading of java programs. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 1–10, Dundee, Scotland. 9

Rao, R. V. (2007). *Decision making in the manufactoring environment*. Springer. 67, 70, 72, 125

Redish, K. A. and Smyth, W. F. (1986). Program style analysis: a natural by-product of program compilation. *Communications of the ACM*, 29(2):126–133. 34

Rees, M. J. (1982). Automatic assessment aids for Pascal programs. *ACM SIGPLAN Notices*, 17(10):33–42. ix, 34, 35

Revilla, M. A., Manzoor, S., and Liu, R. (2008). Competetive learning in informatics: The Uva Online Judge experience. *Olympiads in Informatics*, 2:131–148. 8, 42

Ribeiro, P. and Guerreiro, P. (2009). Improving the automatic evaluation of problem solutions in programming contests. *Olympiads in Informatics*, 3:132–143. 32

Roberts, F. S. (1979). *Measurement Theory with Applications to Decision Making, Utility and the Social Sciences*. Addison-Wesley, London. 60

Roberts, G. B. and Verbyla, J. L. M. (2002). An online programming assessment tool. In *Proceedings of Australasian Computing Education Conference (ACE2003)*, volume 20 of *Conferences in Research and Practice in Information Technology*, pages 69–75, Adelaide, Australia. 29

Roy, B. (1996). *Multicriteria methodology for decision aiding*. Kluwer Academic Publishers, Dordrecht. 48, 52, 59

Royce, W. W. (1987). Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Montery, CA, US. 86, 87

Ryan, J. P. and Deimel, E. L. (1985). Contest problems from the 1985 ACM Scholastic Programming Contest. *ACM SIGCSE Bulletin*, 6(3):7–12. 8

Saaty, T. (1980). *The Analytic Hierarchy Process*. McGraw-Hill, New York. 132

Saghafian, S. and Hejazi, S. R. (2005). Multicriteria group decision making using a modified fuzzy *topsis* procedure. In *Computational Intelligence for Modeling, Control and Automation*, volume IEEE Proceedings. 62

Saikkonen, R., Malmi, L., and Korhonen, A. (2001). Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 133–136, Canterbury, United Kingdom. 30, 32

Salniek, P. and Naylor, J. (1988). Professional skills assessment in programming competitions. *ACM SIGCSE Bulletin*, 20(4):9–14. 8, 21

Schorsch, T. (1995). CAP: An automatic self-assessment tool to check pascal programs for syntax, logic and style errors. In *Proceedings of the 26th SIGCSE technical symposium on Computer science education*, pages 168–172, USA. 33

Sherrel, L. and McCauley, L. (2004). A programming competition for high school students emphasizing process. In *ACM International Conference Proceeding Series*, volume 61, pages 173–182. 13, 38

Shilov, N. V. and Kwangkeun, Y. (2002). Engaging students with theory through ACM Collegiate Programming Contests. *Communications of the ACM*, 45(9):98–101. 12

Simon, H. A. (1976). *Administrative Behavior*. The Free Press, New York. 59

Skienna, S. and Revilla, M. (2003). *Programming Challenges - the Programming Contest Training Manual*. Springer-Verlag, New York. 8, 17, 37, 114

Skūpas, B. and Dagienė, V. (2008). Is automatic evaluation useful for the maturity programming exam. In *Proceedings of 8th International Conference on Computing Education Research, Koli Calling'2008*, pages 117–118. 123

Skūpienė, J. (2004). Testing in informatics olympiads (in Lithuanian). In *Information Technologies Conference Proceedings*, pages 37–41, Kaunas. Technologija. 29

Spacco, J., Strecker, J., Hovemeyer, D., and Pugh, W. (2005). Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5. 9, 33, 45

Stevens, S. S. (1946). On the thory of scales of measurement. *Science*, 103(2684):677–680. xvi, 49

Struble, G. (1991). Experience hosting a high school level programming contest. *ACM SIGCSE Bulletin*, 23(2):36–38. 9, 33, 38

Sule, D. R. (2001). *Logistics of facility location and allocation*. Marcel Dekker, New York, Basel. 66, 67, 71, 126

Sumner, N., Banu, D., and Dershem, H. (2003). JSAVE: Simple and Automated Algorithm Visualization using the Java collection framework. In *Proceedings of the tenth annual Consortium for Computing Sciences in Colleges*. 77

Triantaphyllou, E. (2000). *Multi-Criteria Decision Making Methods: a Comparative Study*. Kluwer Academic Publishers. 48, 59, 60, 61, 62, 63, 64, 66, 68, 69, 72, 73, 74

Trotman, A. and Handley, C. (2006). Programming contest strategy. *Computers & Education*, 50(6):821–837. 8, 12, 16, 17

van der Wegt, W. (2009). Using subtasks. *Olympiads in Informatics*, 3:144–148. 43

van Solingen, R. and Berghout, E. (1999). *The Goal/Question/Metric Method - A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill Publishing Company, Maidenhead, England. 56

Vasiga, T., Cormack, G., and Kemkes, G. (2008). What do olympiads tasks measure? *Olympiads in Infomatics*, 2:181–191. 8, 20, 21, 38, 39

Verhoeff, T. (2002). The 43rd International Mathematical Olympiad: A reflective report on IMO 2002. *Computing Science Report 02-11, Faculty of Mathematics and Computing Science, Eindhoven University of Technology*. `http://www.win.tue.nl/~wstomv/publications/imo2002report.pdf`. 45, 106, 120

Verhoeff, T. (2004). Concepts, terminology, and notations for IOI competition tasks. Document presented at IOI'2004 in Athens. `http://scienceolympiads.org/ioi/sc/documents/terminology.pdf`. 78

Verhoeff, T. (2006). The IOI is (not) a science olympiad. *Informatics in Education*, 5(1):147–158. 13, 19, 31, 37, 38, 40, 41, 44, 45, 92, 106, 107, 114

Verhoeff, T. (2009). 20 years of IOI competition tasks. *Olympiads in Informatics*, 3:149–166. xiv, 8, 19, 76, 95

Verhoeff, T., Horvath, G., Diks, K., and Cormack, G. (2006). A proposal for IOI syllabus. *Teaching Mathematics and Computer Science*, 4(1):193–216. 18

von Matt, U. (1994). Kassandra: the automatic grading system. *ACM SIGCUE Outlook*, 22(1):26–40. 9, 28

Wang, H., Yin, B., and Li, W. (2007). Development and exploration of Chinese National Olympiad in Informatics (CNOI). *Olympiads in Informatics*, 1:165–174. 9

Williams, L. (2006). *Testing Overview and Black-Box Testing Techniques*. North Carolina State University Department of Computer Science. `http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf`. xiii, 12, 36, 121

Winston, W. L. (1991). *Operations Research, Applications and Algorithms*. PWS-Kent Publishing, New York. 72

Woit, D. M. and Mason, D. V. (1998). Lessons from on-line programming examinations. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, pages 257–259, Ireland. Dublin City University. 9, 30

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*. 63, 64

Zadeh, L. A. (1968). Fuzzy algorithms. *Information and Control*, 12:94–102. 63

Zadeh, L. A. (1975a). The concept of a linguistic variable and its application to approximate reasoning - I. *Information Sciences*, 8(3):199–249. 66

Zadeh, L. A. (1975b). The concept of a linguistic variable and its application to approximate reasoning - II. *Information Sciences*, 8(43):301–357. 66

Zadeh, L. A. (1975c). The concept of a linguistic variable and its application to approximate reasoning - III. *Information Sciences*, 9(1):43–80. 66

Zeleny, M. (1982). *Multiple Criteria Decision Making*. McGraw-Hill Book Company: New York. 48

Zhang, W. (2004). Handover decision using fuzzy MADM in heterogeneous networks. In *Wireless Communications and Networking Conference, WCNC, IEEE*, volume 2, pages 653–658. 66, 67

Zhu, Q. and Lee, E. S. (1991). Comparison and ranking of fuzzy nnumbers. *Fuzzy Regression Analysis*, pages 23–32. 66

Jūratė SKŪPIENĖ

EVALUATION OF ALGORITHM-CODE COMPLEXES IN INFORMATICS CONTESTS

Doctoral Dissertation
Physical Sciences (P 000)
Informatics (09 P)
Informatics, Systems Theory (P 175)

Jūratė SKŪPIENĖ

ALGORITMŲ IR JUOS REALIZUOJANČIŲ PROGRAMŲ VERTINIMAS INFORMATIKOS VARŽYBOSE

Daktaro disertacija
Fiziniai mokslai (P 000)
Informatika (09 P)
Informatika, sistemų teorija (P 175)

2010 09 27, 11 sp. l. Tiražas 20 egz.
Parengė spaudai ir išleido Vilniaus universiteto Matematikos ir informatikos institutas
Akademijos g. 4, LT-08663 Vilnius
Interneto svetainė: http://www.mii.lt
Spausdino „Kauno technologijos universiteto spaustuvė“
Studentų g. 54, LT-51424 Kaunas